

Haute Ecole Arc - Ingénierie

Guide développeur Connected Factory

Travail d'Automne 2016 – Projet 15DLM-TA655

Superviseur : Huber Droz

1. Table des matières

1.	Table des matières	3
2.	Introduction.....	4
3.	Client.....	4
3.1.	Handshake	4
3.2.	Lecture et modification de variables.....	5
3.3.	Abonnement.....	6
3.4.	Appel d'une fonction serveur	8
3.5.	Historique d'une valeur	8
4.	Server.....	8
4.1.	Configuration générique	8
4.2.	Instanciations d'un nœud.....	9
4.3.	Méthode.....	9
4.4.	Événement.....	10
4.5.	Historique	11
5.	Conseils.....	11

2. Introduction

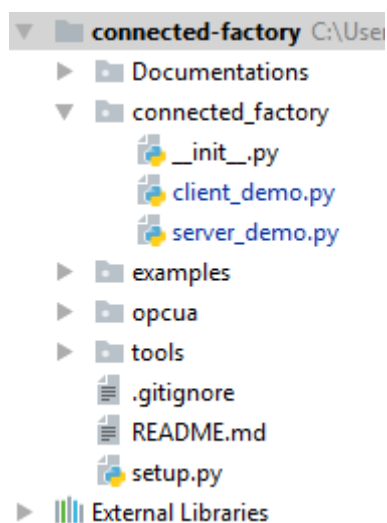
Ce guide détaille l'implémentation de la démonstration liée au projet Connected Factory. Le but est de faciliter l'accès à la spécification OPC UA par l'étudiant qui continuera ce projet. Le langage n'a pas un grand impact sur la spécification donc ce document reste pertinent même si l'utilisateur futur souhaite utiliser une implémentation avec un autre langage. L'implémentation de base utilisée ici est celle du projet FreeOpcUa¹.

Voici la structure du projet. Le dossier connected-factory contient les deux fichiers que nous allons explorer dans ce guide.

Le dossier exemples regroupent les exemples officiels du projet FreeOpcUa.

Le dossier opcua contient l'implémentation de la spécification OPC UA.

Le dossier tools contient des fonctions qui sont utilisables depuis l'interpréteur en ligne de commande.



La démonstration est composée de deux fichiers : client_demo.py et server_demo.py.

La première partie de ce guide détaille ces deux fichiers.

La deuxième partie donne des astuces liées à l'utilisation de spécifique de l'implémentation Python.

3. Client

Le client est instancié au travers d'une classe générique `Client` qui permet de centraliser et de faire abstraction de toutes les opérations liées au réseau.

3.1. Handshake

Premièrement le client doit se connecter au serveur. Le constructeur de la classe `Client` prend l'adresse http du serveur en paramètre.

¹ <https://github.com/FreeOpcUa/python-opcua>

```
client = Client("opc.tcp://localhost:4840/connected-factory/server/")
client.connect()
```

3.2. Lecture et modification de variables

Le client peut ensuite récupérer le nœud racine du serveur. A partir de là tout est possible car le nœud racine permet d'accéder à tous les objets et variables du serveur.

```
root = client.get_root_node()
```

Dans mon exemple je veux accéder à l'objet robot. Le chemin relatif de l'objet à partir du nœud racine sera 0 :Objects puis 2 :Robot1.

DisplayName	BrowseName	NodeId
Root	0:Root	i=84
Objects	0:Objects	i=85
Server	0:Server	i=2253
Robot1	2:Robot1	ns=2;i=1
Arm X coordinate	2:Arm X coordinate	ns=2;i=3
Arm Y coordinate	2:Arm Y coordinate	ns=2;i=4
Arm model	2:Arm model	ns=2;i=6
Arm speed	2:Arm speed	ns=2;i=5
TempSensor	2:TempSensor	ns=2;i=2
Trigger Event	2:Trigger Event	ns=2;i=12
(x) move_arm	2:move_arm	ns=2;i=7
Types	0:Types	i=86
Views	0:Views	i=87

Figure 1 Structure du serveur de démonstration sur le client graphique

```
robot = root.get_child(["0:Objects", "2:Robot1"])
```

La fonction `Node.get_child()` permet de récupérer la référence sur un nœud spécifique. La référence `robot` chez le client pointe sur l'instance de l'objet `Robot1` sur le serveur. Le client peut donc directement modifier l'instance `Robot1`. Si le serveur modifie l'instance sans prévenir le client, le récupérera automatiquement les nouvelles valeurs. En effet, les méthodes d'accès à la valeur des variables et propriétés de l'objet récupère leur valeur sur le serveur et non pas dans le cache du client.

L'accès et la modification d'une variable se fait de la manière suivante :

```
arm_speed = robot.get_child(["2:Arm speed"])
new_speed = 15
arm_speed.set_value(new_speed)
```

Remarque : si la variable est protégée en écriture l'exception « `BadUserAccessDenied` » sera levée.

3.3. Abonnement

Afin de suivre la modification d'une variable, il faut s'abonner à celle-ci. Le serveur enverra une notification pour chaque modification de la variable pendant le temps de l'abonnement.

Premièrement il faut déclarer un `handler` qui est une classe implémente l'interface `SubHandler` qui comprend les fonctions :

```
datachange_notification(node, val, data)
event_notification(event)
status_change_notification(status)
```

La fonction `datachange_notification` est appelée quand la variable à laquelle le `handler` est abonnée est modifiée.

L'argument `node` est une référence au nœud dont la valeur a été modifiée.

L'argument `val` est la nouvelle valeur du nœud.

L'exemple suivant montre un `handler` qui affiche un message dans la console lorsque la température est modifiée.

```
class TemperatureSubHandler(object):

    @staticmethod
    def datachange_notification(node, val, data):
        print("The Temperature of the robot is now
{0:3.1f}".format(val))
```

La fonction `event_notification` est appelée lorsque que le serveur auquel le `handler` est abonné génère un événement.

L'argument `event` permet de récupérer le nom de l'événement ainsi que ses propriétés.

L'exemple ci-dessous affiche un message dans la console qui dépend de la valeur d'une propriété de cet événement.

```
class ServerEventSubHandler(object):

    @staticmethod
    def event_notification(event):
        if event.IsInUse:
            print("Event received from {0}: {1} Current power level is {2}% and
                  is being used."
                  .format(event.SourceName, event.Message, event.PowerLevel))
```

La fonction `status_change_notification` est utilisée pour suivre les changements d'état du serveur.

Maintenant, nous devons instancier un abonnement qui utilise une de nos classes handler. Pour ce faire, on utilise la fonction `Client.create_subscription(periode, handler)`. L'argument `periode` n'a pas d'effet à ma connaissance. Finalement, l'objet `sub` est lié à la variable qu'elle doit monitorer.

```
handler = TemperatureSubHandler()
sub = client.create_subscription(1, handler)
handle = sub.subscribe_data_change(root.get_child(["0:Objects", "2:Robot1",
"2:TempSensor"]))
sub.unsubscribe(handle)
```

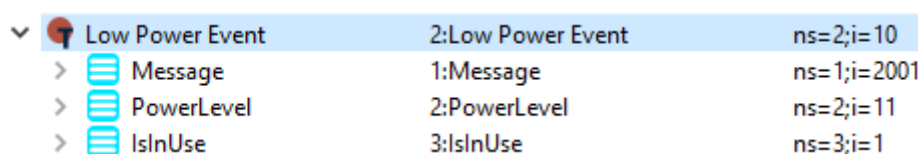
Cette exemple illustre l'abonnement à la variable `TempSensor`.

La variable `handle` est uniquement utilisée pour se désabonner d'une variable.

L'abonnement aux événements du serveur se fait de façon similaire.

```
low_power_event = root.get_child(["0:Types", "0:EventTypes",
"0:BaseEventType", "2:Low Power Event"])
handler = ServerEventSubHandler()
sub = client.create_subscription(500, handler)
handle = sub.subscribe_events(robot, low_power_event)
```

La fonction `subscribe_events(sourcenode, evtype)` prend en paramètre le nœud qui a généré l'événement. L'argument `evtype` permet de spécifier le type de l'événement. Le type de l'événement est en réalité la même information que son emplacement dans l'adresse de nom du serveur (voir Figure 1).



▼ Low Power Event	2:Low Power Event	ns=2;i=10
> Message	1:Message	ns=1;i=2001
> PowerLevel	2:PowerLevel	ns=2;i=11
> IsInUse	3:IsInUse	ns=3;i=1

Figure 2 Événement personnalisé comprenant trois propriétés.

3.4. Appel d'une fonction serveur

L'appel d'une fonction se fait à l'aide de la fonction `Node.call_method(methodid, *args)`. Elle prend en argument l'identifiant de la méthode et les arguments de la fonction.

```
robot.call_method("2:move_arm", x, y)
```

Code 1 Appel la fonction move_arm du nœud Robot1

3.5. Historique d'une valeur

La fonction `Node.read_raw_history(starttime=None, endtime=None, numvalues=0)` permet de récupérer l'historique des modifications du nœud courant. Il est possible de choisir le début et la fin de l'échantillonnage ainsi que le nombre de mesures. Les valeurs par défaut permettent de récupérer tous les enregistrements de cette variable par le serveur jusqu'au moment de l'appel. Elle retourne une liste d'objet possédant les propriétés suivantes : `Encoding`, `ServerPicoSeconds`, `ServerTimestamp`, `SourcePicoSeconds`, `SourceTimestamp`, `StatusCode`, `Value`. Le code illustre l'utilisation de certaines de ces propriétés.

```
history = temp_sensor.read_raw_history(numvalues=5)
for temp in reversed(history):
    print("Temperature at {0} was
{1:3.1f}°.".format(str(temp.SourceTimestamp), temp.Value.Value))
```

4. Server

Le serveur est représenté par la classe `Server`. Elle fait abstraction de toutes les fonctionnalités réseaux et permet d'accéder à l'espace d'adressage du serveur. Celui-ci contient l'arborescence des nœuds.

4.1. Configuration générique

Il faut attribuer une adresse IP au serveur ainsi qu'une URI². L'URI n'est pas nécessaire au fonctionnement du serveur mais fait partie de la spécification et permet d'avoir accès à une page internet donnant des informations sur le serveur.

² [Identifiant une ressource sur un réseau.](#)

```
server = Server()
server.set_endpoint("opc.tcp://0.0.0.0:4840/connected-factory/server/")

uri = "https://github.com/thegazou/connected-factory"
idx = server.register_namespace(uri)
```

4.2. Instanciations d'un nœud

Tout d'abord il nous faut récupérer le nœud `Objet` de base puis nous allons y ajouter des variables, objets ou propriétés. Ci-dessous nous allons instancier un objet `Robot1`. Cet objet nous permettra d'encapsuler toutes les données et fonctions relatives à ce robot. Nous allons lui ajouter un capteur de température dont la valeur initiale est 6.7.

```
objects = server.get_objects_node()
robot = objects.add_object(idx, "Robot1")
temp_sensor = robot.add_variable(idx, "TempSensor", 6.7)
```

Un nœud est par défaut protégé en écriture. Notre variable `TempSensor` ne peut pas être modifiée par le client. Sa valeur est modifiée par le serveur qui reçoit les données directement depuis le capteur physique. Nous allons maintenant ajouter à ce robot une variable définissant la vitesse de déplacement de son bras mécanique. Cette variable pourra être modifiée à tout moment par le client.

```
arm_speed = robot.add_variable(idx, "Arm speed", 45)
arm_speed.set_writable()
```

Il est possible d'ajouter des variables et des objets à une variables. L'élément ajouté deviendra un enfant de l'élément de base.

4.3. Méthode

L'exemple ci-dessous illustre la définition exhaustive des arguments³ d'une fonction dont le nom du nœud est `move_arm` et pointe vers la fonction `move_arm_v` qui prend un argument de type `float` et renvoie un booléen.

Remarque : chaque type de variable est interprété comme si c'était une liste donc la fonction `move_arm_v` accepte une liste de variable de type `float`.

³ [Data Types : Argument](#)


```

inargx = ua.Argument()
inargx.Name = "coordonnées"
inargx.DataType = ua.NodeId(ua.ObjectIds.Double)
inargx.ValueRank = -1
inargx.ArrayDimensions = []
inargx.Description = ua.LocalizedText("List contenant les coordonnées a cible")
robot.add_method(idx, "move_arm", move_arm_v,
                  [inargx], [ua.VariantType.Boolean])

```

Nous devons maintenant implémenter la fonction du côté serveur. Pour ce faire, il faut ajouter le décorateur `@uamethod`.

```

@uamethod
def move_arm_v(parent, coord):
    # Do work
    return True

```

L'argument `parent` est un pointeur vers le nœud parent ; Dans notre cas, il s'agit de l'identifiant numérique de l'objet `Robot1`.

Remarque : Il n'est pas possible de surcharger une méthode avec le décorateur `@uamethod` et on ne peut pas avoir instancier deux méthodes avec le même nom dans l'espace d'adressage.

L'ajout d'une méthode acceptant deux paramètres de type `float` en entrée et aucun paramètre de retour s'écrit ainsi :

```

robot.add_method(idx, "grab_object", grab_object,
                  [ua.VariantType.Double, ua.VariantType.Double], [])

```

4.4. Événement

L'instanciation d'un événement diffère de celui d'un autre nœud dans le sens où on n'instancie pas un type de nœud possédant un type de donnée mais un type de nœud possédant une liste de propriété. L'exemple suivant instancie un événement qui servira à prévenir le client lorsque le niveau d'énergie du robot est en dessous d'un certain seuil. Les informations reçues par le client sont : le nom de l'événement, un message, le niveau d'énergie actuel et un booléen indiquant si le robot est actuellement en fonction. (Voir Figure 2)

```
power_event = server.create_custom_event_type(2, 'Low Power Event',
                                              ua.ObjectIds.BaseEventType)
power_event.add_property(1, 'Message', ua.Variant("Power is low!",
                                                  ua.VariantType.String))
power_event.add_property(2, 'PowerLevel', ua.Variant(15, ua.VariantType.Int32))
power_event.add_property(3, 'IsInUse', ua.Variant(True, ua.VariantType.Boolean))
```

Afin de déclencher l'événement il faut encore le lier à un générateur d'événement. Celui-ci fait le lien entre l'objet d'où vient l'événement et l'événement lui-même. La fonction `trigger()` envoie une notification de l'événement à tous les clients abonné aux événements du serveur.

```
power_event_generator = server.get_event_generator(power_event, robot)

power_event_generator.event.Message = "Power is low!"
power_event_generator.event.IsInUse = True
power_event_generator.event.PowerLevel = 20
power_event_generator.trigger()
```

4.5. Historique

Afin de pouvoir conserver l'historique d'un nœud, il faut définir une méthode de stockage et le nom du fichier dans lequel les données seront écrites. Puis, le serveur doit s'abonner aux modifications d'un nœud. Le service d'abonnement requière l'utilisation d'un serveur actif donc il faut d'abord activer le serveur. L'argument `count` spécifie le nombre de valeur à sauvegarder dans le fichier. Lorsque cette limite est dépassée, les valeurs les plus anciennes sont remplacées par les nouvelles.

```
server.iserver.history_manager.set_storage(HistorySQLite("temp_sensor_history.sql"))

server.start()
server.historize_node_data_change(temp_sensor, period=None, count=100)
```

5. Conseils

Dans cette section je vais vous donner quelque conseil sur l'utilisation du projet FreeOpcUa et plus spécifiquement sur le langage Python :

- L'auto complétion n'est pas toujours fonctionnelle car cette fonctionnalité utilise la réflexion des types qui est possible qu'un fois que le code est compilé. Python est un langage qui n'est pas compilé mais interprété. Cela signifie que l'auto complétion fonctionne dans l'interpréteur après avoir exécuter du code. Afin de contourner ce problème, il est possible d'utiliser la commande `dir(objet)` qui renvoie une liste de tous les attributs de cet objet. Une autre technique consiste à exécuter le code et d'utiliser la fonction d'auto complétion dans l'interpréteur.

- La documentation n'est pas décentralisée et est lacunaire. Voici une liste de site utiles :
 - Documentation de FreeOpcUaⁱⁱ
 - Documentation de Unified Automationⁱⁱⁱ
 - Documentation des Nœuds^{iv}
 - Exemples (pas toujours fonctionnels !)^v

ⁱⁱ <http://python-opcua.readthedocs.io/en/latest/index.html>

ⁱⁱⁱ http://documentation.unified-automation.com/uasdkhp/1.0.0/html/_l1_opc_ua_fundamentals.html

^{iv} <http://python-opcua.readthedocs.io/en/latest/opcua.common.html>

^v <https://github.com/FreeOpcUa/python-opcua/tree/master/examples>