Haute Ecole Arc - Ingénierie

Guide d’utilisateur  
Connected Factory

Travail d’Automne 2016 – Projet 15DLM-TA655

Superviseur : Huber Droz

# Table des matières

[1. Table des matières 3](#_Toc473703878)

[2. Introduction 4](#_Toc473703879)

[3. Client 4](#_Toc473703880)

[3.1. Handshake 4](#_Toc473703881)

[3.2. Lecture et modification de variables 4](#_Toc473703882)

[3.3. Abonnement 5](#_Toc473703883)

[3.4. Appel d’une fonction serveur 7](#_Toc473703884)

[3.5. Historique d’une valeur 7](#_Toc473703885)

[4. Server 8](#_Toc473703886)

[4.1. Configuration générique 8](#_Toc473703887)

[4.2. Instanciations d’un nœud 8](#_Toc473703888)

[4.3. Méthode 9](#_Toc473703889)

[4.4. Evénement 10](#_Toc473703890)

[4.5. Historique 11](#_Toc473703891)

[5. Conseils 11](#_Toc473703892)

# Introduction

Ce guide détaille l’implémentation de la démonstration liée au projet Connected Factory. Le but est de facilité 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 FreeOpcUa[[1]](#footnote-1).

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.

# 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.

## Handshake

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

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

## 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.

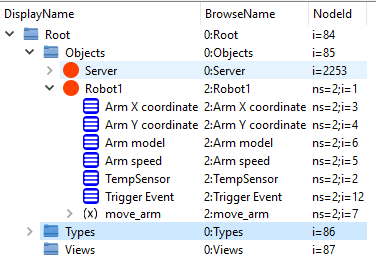


Figure 1 Structure du serveur de démonstration sur le client graphique[[2]](#endnote-1)

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.

## 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 comprends les fonctions la fonction :

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\_subscritpion(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 TempSonsor.  
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).

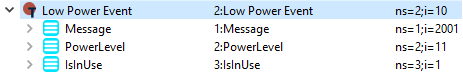


Figure 3 Evénement personnalisé comprenant trois propriétés.

## 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

## 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))

# Server

Le server 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.

## Configuration générique

Il faut attribuer une adresse IP au serveur ainsi qu’une URI[[3]](#footnote-2). 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.

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**)**

## 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 protéger 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.

## Méthode

L’exemple ci-dessous illustre la définition exhaustive des arguments[[4]](#footnote-3) 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.**

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], [])*

## Evénement

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 3)

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**()**

## 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**)**

# 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 fonctionnel 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 FreeOpcUa[[5]](#endnote-2)
  + Documentation de Unified Automation[[6]](#endnote-3)
  + Documentation des Nœuds[[7]](#endnote-4)
  + Exemples (pas toujours fonctionnels !)[[8]](#endnote-5)

1. <https://github.com/FreeOpcUa/python-opcua> [↑](#footnote-ref-1)
2. [↑](#endnote-ref-1)
3. [Identifiant une ressource sur un réseau.](https://fr.wikipedia.org/wiki/Uniform_Resource_Identifier) [↑](#footnote-ref-2)
4. [Data Types : Argument](http://documentation.unified-automation.com/uasdkhp/1.0.0/html/_doc__opc_ua__base_model__data_types.html#Doc_OpcUa_Argument) [↑](#footnote-ref-3)
5. <http://python-opcua.readthedocs.io/en/latest/index.html> [↑](#endnote-ref-2)
6. <http://documentation.unified-automation.com/uasdkhp/1.0.0/html/_l1_opc_ua_fundamentals.html> [↑](#endnote-ref-3)
7. <http://python-opcua.readthedocs.io/en/latest/opcua.common.html> [↑](#endnote-ref-4)
8. <https://github.com/FreeOpcUa/python-opcua/tree/master/examples> [↑](#endnote-ref-5)