UNIVERSITÀ DEGLI STUDI DI CATANIA

OPC-UA Aggregation Server

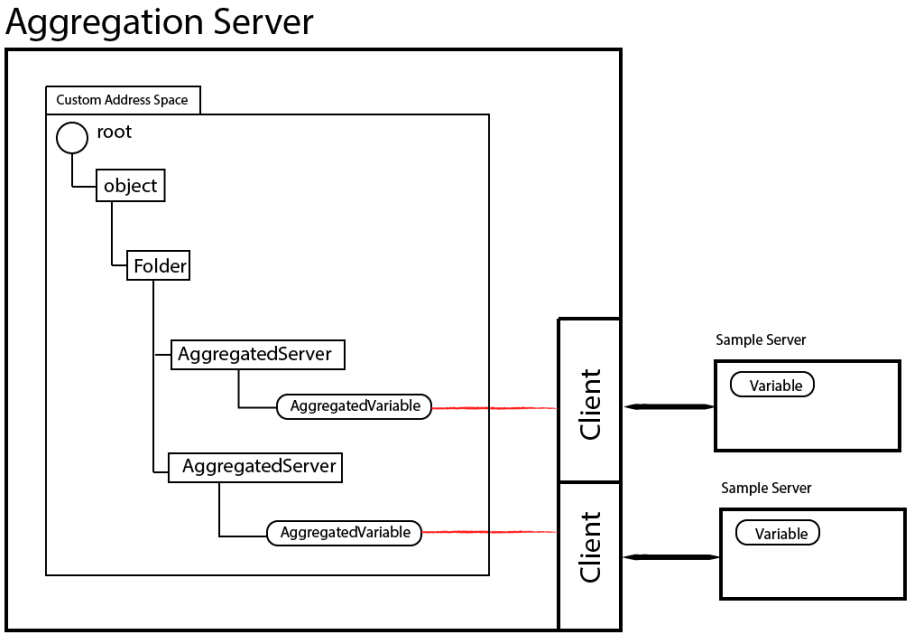
Industrial Informatics a.a 2019/2020

Raiti Mario O55000434
Nardo Gabriele Salvatore O55000430

L'obiettivo di questa tesina di fine corso è la realizzazione di un Aggregation Server utilizzando la versione in python dello stack OPC-UA , disponibile gratuitamente su github al seguente link (https://github.com/FreeOpcUa/python-opcua).

Sommario

Aggregation Server – Architettura

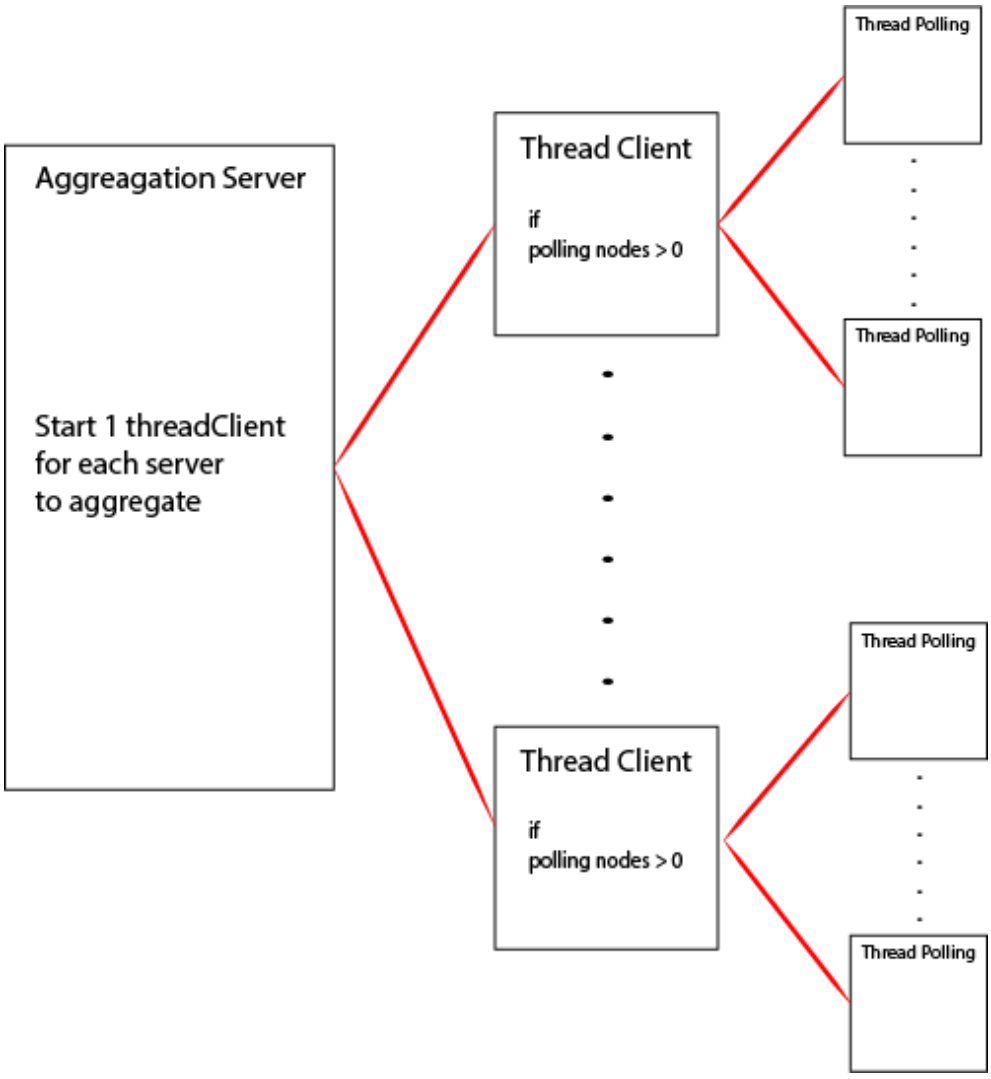


La figura in alto mostra l'architettura di base dell'elaborato. L'elemento Aggregation server sarà un Server OPC-UA. L'address space è stato customizzato creando un nuovo

namespace specifico per l'applicazione e ai suoi componenti di base è stato aggiunto un Node di tipo folder che avrà lo scopo di raccogliere e organizzare gli oggetti AggregatedServer. Tali oggetti modellano i sample server da aggregare A questo scopo è stato creato un nuovo Object Type custom chiamato AggratedServer, al quale è stato aggiunto un set di variabili che modellano i valori di cui si vuole tener traccia.

All'interno dell'aggregation server è previsto uno o più moduli client (uno per ogni server che si vuole contattare) che avranno il compito di stabilire le connessioni con i sample server al fine di leggere e scrivere le variabili aggregate. Le informazioni relative ai sample server da aggregare, e che quindi il modulo client deve raggiungere, sono contenuti all'interno di un file di configurazione in formato *json* (tale file verrà discusso in dettaglio in seguito) in cui sono anche indicati i nodeId delle informazioni da recuperare e le modalità di recupero: subscription o polling (read/write).

I valori prelevati dal modulo client devono essere sincronizzati con le copie locali dell'aggregation server, cioè le variabili degli AggregatedServer. Tali variabili devono quindi mantenere come source timestamp quello del sample server.



File di Configurazione

In questa sezione verrà descritto il file di configurazione.

Config.json

Questo file contiene un array sample, tale array avrà un elemento per ogni server che si vuole aggregare. Per ogni elemento sono previsti dei campi da configurare opportunamente per settare le informazioni relative al server e ai dati da tracciare, e per far si che l'applicativo esegua correttamente. Di seguito vengono descritti tali campi e per ognuno di essi sarà presentato in basso un esempio con dei valori ammissibili:

- **serverName**: stringa che indica il nome server da aggragare che verrà visualizzato collegandosi all'aggreagtion server;
- **endpoint**: deve contenere l'url del server che si vuole aggregare;
- **security_policy**: deve contenere una stringa che rappresenti l'algoritmo utilizzato per le operazioni di sicurezza, ove previste, in accordo al campo security mode;
- **security_mode**: deve contenere una stringa che rappresenti la modalità di sicurezza per le operazioni, ove previste, e determina il riempimento del campo security_policy opportunamente. I valori ammissibili per la policy sono 'None' se scegliamo come mode 'None', 'Basic256' se scegliamo come mode 'Sign' o 'Basic256Sha256'/'Basic128Rsa15' se scegliamo come mode SignAndEncrypt;

Il campo **sub_infos** è anch'esso un array, ogni suo elemento indicherà le caratteristiche di una singola subscription, ogni elemento presenta i seguenti campi :

- **requested_publish_interval**: definisce il publish interval della subscription in millisecondi;
- **requested_lifetime_count**: intero che indica quante volte il publishing interval può trascorrere senza che sia monitorata alcuna attività da parte del client. Passato questo lasso di tempo, il server cancella la Subscription e libera le risorse. ***NOTA BENE: questo parametro dev'essere grande almeno tre volte il keep alive count***;
- **requested_max_keepalive_timer**: intero che indica quante volte il publishing interval deve trascorrere senza che sia disponibile alcuna Notifications da inviare al Client, perché il server mandi un keep-alive messagge al client in grado di comunicargli che quella particolare Subscription è ancora attiva;
- **max_notif_per_publish**: intero che indica il numero massimo di notifiche per ogni publish;
- **publishing_enabled**: booleano che abilità la pubblicazione dei messaggi prodotti dai monitored item;
- **priority**: intero che indica la priorità associata alla sottoscrizione;

Il campo **monitoring_info** è un array, ogni su elemento andrà ad indicare un oggetto che si vuole monitorare. Sono possibili due tipi di elementi in base al tipo di monitoring richiesto la discriminazione avviene tramite il campo **monitoringMode**, che può assumere due valori : *monitored_item* in questo caso l'elemento costituirà il monitored

item che verrà creato nel meccanismo di sottoscrizione, *polling* se si vuole monitorare il nodo con letture ad intervalli regolari.

Nel caso in venga settato **monitoringMode** con *monitored_item* seguiranno i seguenti campi :

- **displayName** : stringa che indica il nome della variabile da monitorare che verrà visualizzato collegandosi all'aggreagtion server;
- **client_handle** : intero che verrà utilizzato per associare il monitored item alla variabile nell'aggregation server e permetterne l'aggiornamento. Tale valore deve essere diverso per ogni monitored item e preferibilmente progressivo;
- **subIndex** : indice dell'array sub_info corrispondente alla sottoscrizione a cui si vuole associare il monitored item;
- **nodeTomonotor** : stringa contenente il nodeid del nodo da monitorare,
- **sampling_interval**: intero che indica l'intervallo di tempo con la quale vengono prodotte le notifiche dai monitored item e poste nella coda dei messaggi;
- **queue_size**: intero che indica la dimensione della coda dei monitored items;
- **discard_oldest**: booleano che definisce la politica di gestione dei messaggi quando la coda del monitored item è piena;
- **deadbandval**: valore da inserie in accordo al tipo di deadband voluta: se si sceglie la deadband percentuale bisogna inserire un numero compreso tra 0 e 100; se si sceglie invece la deadband assoluta bisogna mettere il valore che andrà a costituire la soglia;
- **deadbandtype**: permette di settare il tipo di dead band che verrà presa in considerazione dal datachange filter, come valore ammissibile ha **1** se vogliamo una dead band assoluta, **2** se vogliamo una deadband percentuale e **0** se vogliamo aggiornare continuamente la variabile ad ogni cambiamento di valore. Scegliere una dead band percentuale solo se si vuole monitorare un node analog.

Nel caso in venga settato **monitoringMode** con *polling* seguiranno i seguenti campi :

- **nodeTomonotor** : stringa contenente il nodeid del nodo da monitorare,
- **refreshing_interval** : intero che rappresenta l'intervallo tempo in secondi che indica la periodicità della lettura e aggiornamento dei valori

In basso viene riportato un esempio di come riempire i campi del suddetto file.

```
{

   "servers" : [{

        "serverName" : "Sample Server 1",

        "endpoint":"opc.tcp://pc-mario:51210/UA/SampleServer",

        "security_policy":"Basic128Rsa15",

        "security_mode":"SignAndEncrypt",

        "sub_infos": [
```

```
    {
        "requested_publish_interval": 2000,
        "requested_lifetime_count": 30000,
        "requested_max_keepalive_timer": 5000,
        "max_notif_per_publish": 2147483647,
        "publishing_enabled": true,
        "priority": 0
    },
    {
        "requested_publish_interval": 3000,
        "requested_lifetime_count": 30000,
        "requested_max_keepalive_timer": 5000,
        "max_notif_per_publish": 2147483647,
        "publishing_enabled": true,
        "priority": 0
    }
],
"monitoring_info":[
    {
        "displayName” : “Int64Value",
        "nodeTomonitor": "ns=2;i=11206",
        "monitoringMode": "polling",
        "refreshing_interval": 2
    },
    {
       "displayName” : “Uint64Value",
        “client_handle” : 1,
        "subIndex": 0,
        "nodeTomonitor": "ns=2;i=11212",
        "monitoringMode": "monitored_item",
        "sampling_interval": 2000,
```

```
                "queue_size": 1,
                "discard_oldest": true,
                "deadbandval": 40,
                "deadbandtype": 2
            }
        ]
    }
    ]
}
```

Implementazione

In questa sezione saranno mostrati i dettagli implementativi, discusse le scelte progettuali e le funzionalità sviluppate nei file *aggregationServer.py*, *Client.py*, *Thread_client.py*, *Thread_Polling*.

L'elaborato è stato sviluppato in ambiente Windows (Windows 10 Professional) e per tale motivo le scelte implementative sono mirate all'esecuzione su tale piattaforma (gestione dei path). Da ciò ne consegue che potrebbero incorrere errori durante l'esecuzione su altre piattaforme diverse da quella presa in considerazione.

Il codice è stato sviluppato utilizzando l'ultima versione di python, cioè la 3.8. Come editor è stato utilizzato *VScod*e.

aggregationServer.py

Questo file rappresenta il cuore dell'elaborato implementando il server OPC-UA che si occuperà di aggregare i sample server.

Vengono importate come dipendenze i moduli *ua* e *Server* dello stack *opcua*, *json* per la gestione dei file di configurazione, *time* e la classe ThreadClient presente nel file Thread_client.py.

Inizialmente viene gestito il caricamento delle informazioni di configurazione presenti nei file della directory config. In particolare, viene creata la lista *aggr_servers* che conterrà in ogni sua entry un elemento 'sample_server' che conterrà tutte informazioni dei singoli server passati in configurazione.

Una volta caricate le informazioni di configurazione viene inizializzato il server istanziando la classe Server dello stack. Viene settato l'endpoint, tramite il quale il server sarà interrogabile, attraverso lo specifico metodo *set_endpoint()* e le policy di sicurezza supportate dal server attraverso *set_security_policy()*.

Attraverso i metodi load_certificate() e load_private_key() vengono caricati il certificato di sicurezza del server e la sua chiave privata al fine di poter garantire le policy di sicurezza settate. Di seguito un'immagine che mostra la sequenza delle istruzioni usate.

```
# Server Setup endpoint and security policy settings
server = Server()
server.name = "AggregationServer"
server.set_endpoint("opc.tcp://127.0.0.1:8000/AggregationServer/")

server.set_security_policy([ua.SecurityPolicyType.NoSecurity,
                            ua.SecurityPolicyType.Basic128Rsa15_Sign,
                            ua.SecurityPolicyType.Basic128Rsa15_SignAndEncrypt,
                            ua.SecurityPolicyType.Basic256_Sign,
                            ua.SecurityPolicyType.Basic256_SignAndEncrypt,
                            ua.SecurityPolicyType.Basic256Sha256_SignAndEncrypt,
                            ua.SecurityPolicyType.Basic256Sha256_Sign])

# load server certificate and private key. This enables endpoints
server.load_certificate(certificate_path + "server_certificate.der")
server.load_private_key(certificate_path + "server_private_key.pem")
```

Concluse le operazioni relative alla sicurezza si è proceduto alla creazione e alla popolazione dell'address space custom, realizzato appositamente per l'aggregation server. La creazione del name space è stata realizzata attraverso la funzione *register_namespace()* che restituisce come valore di ritorno l'indice del name space all'interno del namespace array. Creato il namespace si è ottenuto il nodo attraverso la funzione *get_objects_node()*. A tale elemento è stato poi aggiunto un nodo folder chiamato Aggregator che sarà il contenitore degli oggetti che rappresenteranno i server aggregati.

Prima di eseguire la creazione dinamica degli oggetti, si è creato un objectType specifico per rappresentare i server aggregati. Tale oggetto è stato derivato da un BaseObjectType, ed aggiunto al namespace custom col nome di AggregatedServerType.

A seguito di ciò sono stati istanziati gli oggetti AggregatedServer in maniera dinamica, uno per ogni sample server contenuto nel file di configurazione. Inoltre, se nel file di configurazione è stata passata una lista di nodeid, vengono aggiunte all'oggetto n variabili ed n properties al fine di avere una variable per ogni nodo che si vuole monitorare. Questo permette di avere una corrispondenza di un oggetto per ogni sample server e tale oggetto conterrà un numero di variabili uguali al numero di nodi da monitorare nel sample server. Le n property vengono utilizzate per creare una corrispondenza univoca tra una varabile dell'Aggregation Server e la variabile che vogliamo monitorare nel sample server. A tale scopo si setta come valore della property una stringa con il nodeid passato in configurazione. Inoltre ad ogni Aggregated server è stata associata una property che contiene l'indirizzo del server remoto a cui corrisponde.

Durante le operazioni di popolamento dell'address space, sono stati creati due dizionari utilizzati di traduzione, utilizzati per verificare le corrispondenze tra variabili locali e remote al momento dell'aggiornamento del valore :

- **handle_dict** : dizionario che tiene traccia della corrispondenza tra il nodeid delle variabili dell'aggregation server relative ai nodi da monitorare attraverso sottoscrizione e il *client_handle* del monitored item passato in configurazione.
- **polling_dict** : dizionario che tiene traccia della corrispondenza tra il nodeid delle variabili dell'aggregation server e il nodeid del server remoto che si vuole monitorare in modalità polling.

Di seguito un'immagine che mostra la sequenza delle istruzioni usate.

```
# Setup our namespace
uri = "http://Aggregation.Server.opcua"
#Getting the index of our nasmespace
idx = server.register_namespace(uri)

# get Objects node. This is where we should put our custom stuff
objects = server.get_objects_node()

# populate our namespace with the aggreagated element and their variables
aggregator = objects.add_folder(idx, "Aggregator")

# definition of our custom object type -> AggregatedServer
types = server.get_node(ua.ObjectIds.BaseObjectType)
mycustomobj_type = types.add_object_type(idx, "AggregatedServerType")
mycustomobj_type.set_modelling_rule(True)
```

```python
# each element of the node_ids list contains the node ids of the nodes that we want to monitor on a single sample server
#This list is useful when populating object tree
c_handles = [[] for i in range(len(aggr_servers))]
node_ids = [[] for i in range(len(aggr_servers))]
for i in range(len(aggr_servers)):
    for j in range(len(aggr_servers[i]["monitoring_info"])):
        node_ids[i].append(aggr_servers[i]["monitoring_info"][j]["nodeTomonitor"])
        if(aggr_servers[i]["monitoring_info"][j]["monitoringMode"] == "monitored_item"):
            c_handles[i].append(aggr_servers[i]["monitoring_info"][j]["client_handle"])

#Populating the object tree with our custom object type and variables that will contain the values that we want to monitor.
#Added also a property for each variable, that will contain the node id of the node in the sample server that we want to obtain values ->
# -> relationship between our Aggregated server and sample servers
var_node_ids = [[] for i in range(len(aggr_servers))]
polling_dict = [{} for i in range(len(aggr_servers))]
aggregatedServers_objects = [] #aggregated servers objects list
for i in range(len(aggr_servers)):
    obj = aggregator.add_object(idx,"AggregatedServer_"+str(i+1), mycustomobj_type.nodeid)
    obj.add_property(idx, "Remote URL Server", aggr_servers[i]["endpoint"])
    for j in range(len(node_ids[i])):
        var = obj.add_variable(idx,"AggregatedVariable_"+str(j+1), 0)
        if(aggr_servers[i]["monitoring_info"][j]["monitoringMode"] == "monitored_item"):
            var_node_ids[i].append(var.nodeid)
        if(aggr_servers[i]["monitoring_info"][j]["monitoringMode"] == "polling"):
            polling_dict[i][str(var.nodeid)] = (node_ids[i][j])
        var.add_property(idx, "Remote NodeId", node_ids[i][j])
        var.set_writable()
    aggregatedServers_objects.append(obj)


#Creating dictionary of client_handles and node_ids
handle_dict = [{} for i in range(len(aggr_servers))]
for i in range(len(aggr_servers)):
    for j in range(len(c_handles[i])):
            handle_dict[i][str((var_node_ids[i][j]))] = (c_handles[i][j])
```


Una volta popolato l'address space, viene avviato il server chiamando il metodo *start()* sull'oggetto server creato inizialmente. A seguito dello start, attraverso un print viene mostrato l'url a cui si può contattare il server e il comando per lo spegnimento del server.

```python
# starting server
server.start()
print("Available Endpoint for connection : opc.tcp://127.0.0.1:8000/AggregationServer/")
print("Press Ctrl + C to stop the server...")


# Creazione dei threads per gli n client
clients_threads = []
for i in  range(len(aggr_servers)):
    clients_threads.append(ThreadClient(aggr_servers[i],certificate_path, aggregatedServers_objects[i]))

for i in range(len(clients_threads)):
    print("-----------------------------------")
    print(f"Thread {i} started..")
    clients_threads[i].start()
try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    for i in range(len(clients_threads)):
        clients_threads[i].stop()
        clients_threads[i].join()

finally:
    print("-----------------------------------")
    print("Server Stopping...")
    print("-----------------------------------")
    server.stop()
```

La figura in alto mostra l'ultima parte del codice sorgente. In questa porzione vengono creati e successivamente avviati gli n thread che si occuperanno di istanziare n client che permettono di prelevare le informazioni dai sample server e aggiornare l'adress space del nostro server, così da tenere allineati i valori delle variabili.

I thread vengono creati come istanza della classe *ThreadClient* contenuta nel file *Thread_client.py* (che verrà discusso in seguito) e salvati in una lista.

Alla creazione vengono passate al thread la lista delle informazioni relative ad un singolo sample server, il percorso contenente i certificati di sicurezza e l'oggetto del name space che corrisponde al sample server ad esso assegnato, così da permettere l'aggiornamento dei valori. Ogni thread si occuperà di istanziare un client ed eseguire il servizio richiesto nel file di configurazione.

Infine, è presente un costrutto *try/except/finally*. Nella clausola try è presente un while loop infinto per non far concluder l'esecuzione del server. La clausola except è stata associata ad un *KeybordInterrupt, scatenato* premendo crtl+c. Il controllo di questo interrupt si occupa di stoppare i thread (quindi invocare il metodo *stop()* su tutti i thread avviati) e successivamente attendere la loro conclusione attraverso metodo *join()*. La clausola finally si occupa di stoppare il server per concludere l'esecuzione del modulo.

Client.py

Questo file contiene l'implementazione della classe *Client_opc()* , e la classe *SubHandler()*. Vengono importate come dipendenze i moduli *ua* e la classe *Client* dello stack *opcua*.

La classe ***SubHandler()*** è necessaria per l'implementazione del servizio di sottoscrizione, in particolare il suo metodo *datachange_notification()* viene richiamato ogni volta che un nuovo message notification è disponibile. In particolare, tale metodo si occupa di aggiornare le variabili dell'oggetto AggrObject passato come parametro, con i nuovi valori ottenuti dal servizio di sottoscrizione. Ogni volta che da*tachange_notification()* viene richiamato, recupera le variabili dell'oggetto richiamando su AggrObject il metodo dello stack *get_variables()*. La verifica di quale variabile deve essere aggiornata viene realizzata nel seguente modo. Ciclando su tutte le variabili ottenute e su tutte le chiavi di handle_dict si verifica che per quella specifica chiave il client_handle ricevuto dal messaggio di notifica sia quello associato all'elemento di handle_dict che ha la chiave considerata e che il nodeid dell'iesima variabile sia uguale alla chiave.

L'aggiornamento viene realizzato chiamando sulle variabili la *set_value(),* passando come argomento della funzione il valore ottenuto dalla subscription.

```python
class SubHandler(object):

    #We pass Aggr Object to updated our variables whan a datachange_notification is called
    def __init__(self, AggrObject, handle_dict):
        self.AggrObject = AggrObject
        self.handle_dict = handle_dict

    def datachange_notification(self, node, val, data):
        threadLock = threading.Lock()

        print("Subscription Service: New data change event:", data.monitored_item.ClientHandle, val, node)
        #Getting node id string to compare with properties of aggregated variables
        AggrVar = self.AggrObject.get_variables()
        for var in AggrVar:
            threadLock = threading.Lock()
            for key in self.handle_dict:
                if(self.handle_dict[key] == data.monitored_item.ClientHandle and str(var.nodeid) == key):
                    threadLock.acquire()
                    var.set_value(data.monitored_item.Value)
                    threadLock.release()
```

La classe ***Client_opc()*** è una classe controller, che implementa lo specifico client che si occupa di comunicare col singolo sample server assegnatogli in fase di configurazione. Tali oggetti vengono poi istanziati dai singoli thread. I metodi forniti da tale classe utilizzano le funzioni offerte dallo stack.

Sono presenti i seguenti metodi in ***Client_opc()*** :

- ***__init__(****self, cert_path, server_path, policy, mode, AggrObj*): si occupa di associare agli attributi della classe, i parametri forniti alla creazione dell'oggetto. I parametri passati al costruttore sono:
    - *cert_path*: percorso contenente i certificati di sicurezza del client;
    - *server_path*: url del sample server da contattare:
    - *policy*: stinga che definisce la politica di sicurezza selezionata in accordo alla mode;
    - *mode*: stringa che identifica la modalità di sicurezza richiesta nell'interazione tra il client e il sample server,
    - *AggrObj:* oggetto Aggreated Server presente in aggregated server associato al server da monitorare;
- ***client_istantiate***(*self*): si occupa di istanziare l'oggetto della classe Client, e se richiesta una modalità di sicurezza diversa da None, viene settata la security policy voluta sull'oggetto istanziato;
- ***secure_channel_and_session_activation***(*self*): setta i timeout richiesti e richiama il metodo dello stack per instaurare la connessione;
- ***disconnect***(*self*): si occupa di richiamare il metodo dello stack per eseguire la disconnessione;
- ***readData***(*self, node_id, polling_dict*): richiede come parametro il nodeid del nodo da leggere e il dizionario di traduzione ed implementa il servizio di lettura e aggiornamento delle variabili di AggrObject;

- ***set_datachange_filter***(*self, deadband_val, deadbandtype*): utilizzata per implementare il servizio di sottoscrizione, prende in ingresso i parametri per la creazione del DataChange filter, passati tramite il config file, e si occupa di creare ed inizializzare tale filtro;
- ***create_motired_item***(*self, subscription, sub_node , sampling_interval, client_handle , filter=None, queuesize = 0, discard_oldest = True, attr=ua.AttributeIds.Value*): utilizzata per implementare il servizio di sottoscrizione, si occupa di creare i monitored item con le caratteristiche specificate dai parametri di ingresso che corrispondono alle informazioni sui monitored items passati dal file di configurazione;
- ***make_monitored_item_request***(*self, subscription, node, attr, sampling_interval, client_handle , filter, queuesize, discard_oldest*) : utilizzata per implementare il servizio di sottoscrizione, viene richiamata dalla create_monitored_item per effettuare la richiesta di creazione dei monitored item, settando i parametri di tale richiesta in base a quelli ottenuti dal file di configurazione;
- ***subscribe***(*self, node_id, sub_infos, sub_info_ids , monitored_item_infos , monitored_item_info_ids*): richiede come parametri tutte le informazioni contenute nel file di configurazione relative alle caratteristiche delle sottoscrizioni da creare e ai monitored item ad essa associata; implementa il servizio di sottoscrizione.
- ***delete_monit_items***(*self, sub, handle*): richiede come parametro gli elementi sub e handle che vengono ritornati dalla funzione di subscribe. Sub rappresenta la lista di oggetti sottoscrizione mentre handle rappresenta la lista dei monitored item id. Per ogni elemento di handle richiama il metodo dello stack per realizzare l'unsubscribe (e quindi cancellazione dei monitored items).
- ***delete_sub***(*self, sub):* richiede come parametro l'elemento sub (lista di sottoscrizioni) e per ogni elemento, chiama la funzione delete dello stack per eliminare le sottoscrizioni.

Menzione particolare merita la funzione 'subscribe', che esegue sia la creazione delle subscriptions, che la creazione dei monitored items (chiamando create_monitored_item al suo interno) in base alle info passate in ingresso. Inoltre, per implementare questo metodo, si è scelto di non utilizzare la funzione *deadband_monitor* fornita dallo stack per la creazione dei monitored items con successive chiamate innestate di funzioni dello stack. Questo per poter avere maggiore liberta nella personalizzazione dei parametri dei monitored item. Richiamando infatti solamente la funzione di *deadband_monitor,* tutti i parametri tranne la queue size vengono settati staticamente senza possibilità di variazione, dovuto alla chiamata di varie funzioni innestate che creano i parametri con valori statici.

Nell'implementazione della subscribe è stato usato lo stesso flusso logico della soluzione fornita dallo stack però utilizzando delle funzioni custom per la creazione dei datachange filter e dei monitored item. Per la gestione delle notifiche viene sempre settato un evento di tipo DataChangeTrigger che andrà a chiamare la *datachange_notification* implementata nell'handler, ogni volta che un nuovo notification message è disponibile.

Le specifiche funzioni dello stack utilizzate dai metodi sopra elencati sono discusse alla sezione NOTE SULLE FUNZIONI DELLO STACK.

Thread_client.py

In questo file viene implementata la classe ***ThreadClient*** in cui viene definito il comportamento dei thread che vengono creati e lanciati dal file *aggregationServer.py*.

Vengono importate come dipendenze il file Client.py (discusso precedentemente) il modulo threading e la classe *PollingService* presente in Thread_polling.py. In particolare, verrà utilizzata la classe Thread di threading che verrà estesa da ***ThreadClient***.

Sono presenti 4 metodi in ***ThreadClient***:

- ***__init__*** **(**self, sample_server_conf, certh_path, AggrObject, handel_dict , polling_dict**)**: il costruttore della classe si occupa di invocare il costruttore della classe padre e di associare agli attributi della classe i parametri passati. Inoltre, associa un evento all'attributo stopper (tale attributo verrà utilizzato per implementare lo stop del thread). I parametri passati al costruttore sono:
  - *sample_server_conf*: lista delle informazioni relative ad un singolo sample server;
  - *certh_path*: percorso contenente i certificati di sicurezza del client da istanziare;
  - *AggrObject*: oggetto Aggreated Server presente in aggregated_server associato al server da monitorare;
  - *Handle_dict* : dizionario di traduzione per il servizio di sottoscrizione;
  - *Polling_dict* : dizionario di traduzione per il servizio di polling;
- ***stop***(): questo metodo setta a true il valore dell'evento associato all'attributo *stopper,*
- ***stopped***(): questo metodo ritorna un booleano che corrisponde al valore dell'evento su stopped.

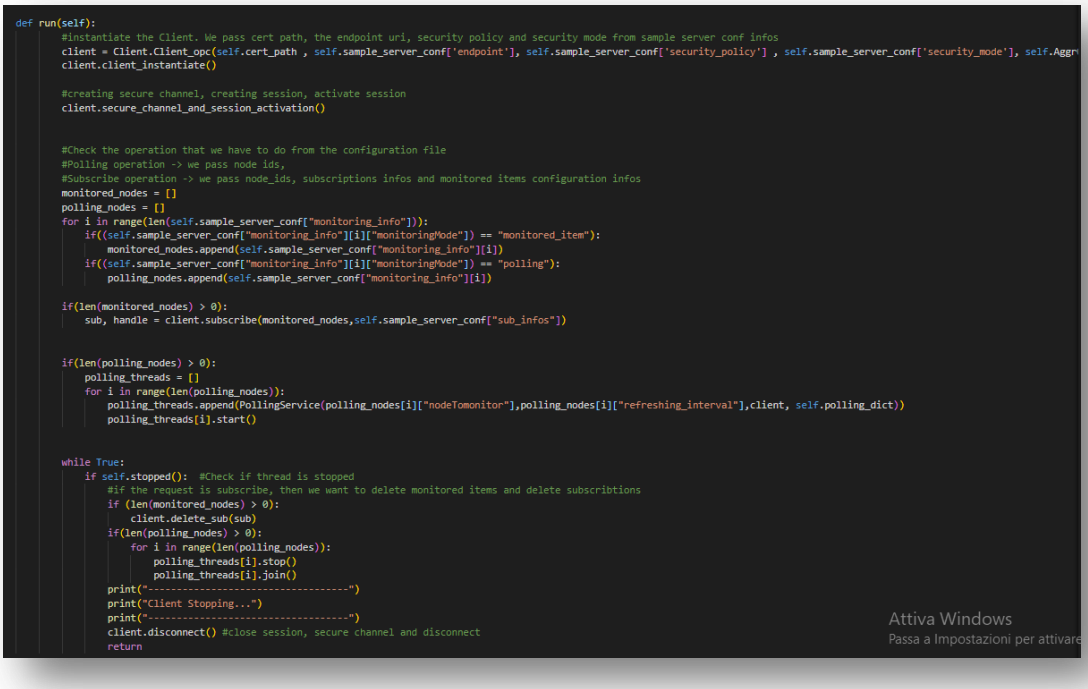
```
def __init__(self , sample_server_conf , cert_path, AggrObject):
    threading.Thread.__init__(self)
    self._stopper = threading.Event()
    self.sample_server_conf = sample_server_conf
    self.cert_path = cert_path
    self.AggrObject = AggrObject

def stop(self):
    self._stopper.set()

def stopped(self):
    return self._stopper.isSet()
```

I metodi stop e stopped vengono usati insieme per implementare un meccanismo di stop del thread; forzano il metodo run() ad eseguire un return e quindi interrompersi.

- ***run***() : implementando questo metodo si definisce il comportamento del thread, perché tale metodo è quello che viene invocando alla chiamata del metodo start sull'istanza della classe. Questo metodo si occupa di istanziare un oggetto Client della classe Client_opc definita nel file Client.py. Una volta istanziato, crea il canale sicuro, crea la sessione e la attiva, connettendosi al server indicato nel file di configurazione. Creata la connessione, verifica quale servizio viene specificato dalle informazioni di configurazione ed invoca la rispettiva funzione di Client_opc per eseguire il servizio ed aggiornare le copie locali dell'address space. In particolare nel caso del servizio di polling si occupa di lanciare n thread che implementano il servizio di polling quanti sono i nodi da monitorare nel seguente modo. Viene anche qui definito un while loop in cui viene sempre verificato se è stato settato l'evento di stop; appena questo si verifica, se sono presenti subscription, vengono distrutte, se sono stati lanciati thread di polling questi vengono stoppati ,il client si disconnette dal sample server e viene eseguito il return terminando così l'esecuzione.



Thread_polling.py

In questo file viene implementata la classe ***PollingService*** in cui viene definito il comportamento dei thread che vengono creati e lanciati dal file *Thread_client.py*.

Vengono importate come dipendenze time e il modulo threading. In particolare, verrà utilizzata la classe Thread di threading che verrà estesa da ***PollingService*** per implementare il servizio di lettura periodica dei dati.

Sono presenti 4 metodi in ***PollingService***:

- ***__init__ (****self, node_id, red_int, client, polling_dict****)*** : il costruttore della classe si occupa di invocare il costruttore della classe padre e di associare agli attributi della classe i parametri passati. Inoltre, associa un evento all'attributo stopper (tale attributo verrà utilizzato per implementare lo stop del thread). I parametri passati al costruttore sono:
  - *node_id* : il nodeid del nodo da leggere;
  - *red_int* : periodo di ripetizione dell'operazione di lettura;
  - *client* : il client istanziato da Thread_client su cui invocare l'operazione di lettura;
  - *Polling_dict* : dizionario di traduzione.
- ***stop***() : questo metodo setta a true il valore dell'evento associato all'attributo *stopper;*
- ***stopped***() : questo metodo ritorna un booleano che corrisponde al valore dell'evento su stopped;
- ***run()*** : implementando questo metodo si definisce il comportamento del thread, perché tale metodo è quello che viene invocando alla chiamata del metodo start sull'istanza della classe. Contiene un implementazione molto semplice, costituita da un ciclo while attivo fin quando il thread non viene stoppato, l'operazione di lettura è realizzata invocando il metodo *readData()* dell'oggetto client, a seguito della lettura il thread viene messo in sleep per una frazione di tempo pari a *red_int*.

```
class PollingService(threading.Thread):
    def __init__(self, nodeid , ref_int , client, polling_dict):
        threading.Thread.__init__(self)
        self._stopper = threading.Event()  #used ih thread stopping
        self.ref_int = ref_int
        self.nodeid = nodeid
        self.client = client
        self.polling_dict = polling_dict

    def stop(self):
        self._stopper.set()

    def stopped(self):
        return self._stopper.isSet()

    def run(self):
        while not(self.stopped()):
            self.client.readData(self.nodeid, self.polling_dict)
            time.sleep(self.ref_int)
```

Note sulle funzioni dello stack

Il modulo Client viene istanziato tramite la chiamata al costruttore della classe 'Client' importata dallo stack opc ua per python utilizzato in questo progetto. La chiamata effettuata è Client(server_path), dove *server_path* è l'url del sample server alla quale vogliamo che si colleghi il nostro modulo Client.

Dopo aver istanziato il Client, dobbiamo caricare il certificato, la private key e settare eventuali security policy e security mode: questo viene fatto tramite la funzione *set_security_string* della classe Client dello stack, che prende come parametro una stringa che ha tutte le informazioni sopra citate. Questo permetterà di connetterci all'endpoint consono con le politiche di sicurezza scelte.

Per connetterci all'endpoint, viene usata la funzione *connect* dello stack, che maschera le varie sequenze di azioni da intraprendere per connettere un client ad un endpoint. Infatti, questa funzione dello stack, chiamerà al suo interno altre funzioni innestate per la *creazione del canale sicuro*, *creazione della sessione* e *attivazione* della stessa.

Come controparte della *connect*, la funzione *disconnect* dello stack, maschera le varie sequenze di azioni per disconnettere un client ad un endpoint.

Per leggere un qualsiasi valore dato un node id, ciò che bisogna fare è ottenere innanzi tutto il nodo tramite il node id ed in seguito leggerne il valore. Queste due azioni vengono eseguite tramite la funzione *get_node* della classe Client e la funzione *get_data_value* della classe Node dello stack opc ua.

Per scrivere un valore dato un node id, la sequenza di azioni è identica della read, eccetto che viene utilizzata la funzione *set_value* della classe Node dello stack piuttosto che la get_data_value.

Per effettuare una sottoscrizione e creare i monitored items data una lista di node id, vengono utilizzate due funzioni dello stack: *create_subscription,* chiamata sulla classe Client e *create_monitored_items* chiamata sulla sottoscrizione all'interno della nostra funzione custom *create_ monitored*. Il metodo *create_subscription* ha come parametri di ingresso il singolo publish_interval oppure una struttura dati di tipo *ua.CreateSubscriptionParameters* se si vogliono passare tutti i parametri e un handler che viene istanziato tramite la classe SubHandler creata nel nostro progetto. Tale classe deve possedere le funzioni *datachange_notifications* ed *event_notifications* specificate dallo stack. Esse verranno chiamate per notificare un cambiamento dei valori delle variabili.

Se non siamo più interessati ad essere aggiornati sui cambiamenti di valore di una variabile, possiamo passare il monitored item id associato alla variabile come parametro di ingresso alla funzione dello stack *unsubscribe,* chiamata sulla sottoscrizione di interesse. Possiamo iterare questo comportamento per tutte le variabili non più di interesse: ciò viene fatto tramite la funzione *delete_monit_items* del modulo Client creato da noi, che prende in ingresso una lista di monitored item ids e chiama per ognuno di essi la funzione *unsubscribe* dello stack.

Per eliminare una sottoscrizione, basterà chiamare la funzione *delete* dello stack sulla sottoscrizione che si vuole eliminare. Tale funzione eliminerà anche tutti i monitored items presenti. È preferibile però chiamare la funzione di *unsubscribe* appena citata su tutti i monitored items prima di effettuare la cancellazione della sottoscrizione.

Creazione dei Certificati

Per il corretto funzionamento dell'elaborato e per utilizzare i meccanismi di sicurezza offerti da OPC-UA è necessario creare i certificati di sicurezza x509v3 da posizionare all'interno della directory *certificates* del progetto. Per la creazione dei certificati è necessario aver installato openssl sulla propria macchina.

Bisogna posizionarsi all'interno del path di installazione di Openssl nella directory bin. Una volta situati in questo path "*C:\Program Files\OpenSSL-Win64\bin*", bisogna creare un file di configurazione per openssl con la seguente struttura:

```
[ req ]

default_bits = 2048

default_md = sha256

distinguished_name = subject

req_extensions = req_ext

x509_extensions = req_ext

string_mask = utf8only

prompt = no

[ req_ext ]

basicConstraints = CA:FALSE

nsCertType = client, server

keyUsage = nonRepudiation, digitalSignature, keyEncipherment, dataEncipherment, keyCertSign

extendedKeyUsage= serverAuth, clientAuth

nsComment = "OpenSSL Generated Certificat"

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid,issuer

subjectAltName = DNS: DESKTOP-V6N8M9J, IP: 127.0.0.1

[ subject ]

countryName = IT

stateOrProvinceName = CT

localityName = CT

organizationName = MyOrg
```

Importante inserire all'interno del campo `DNS` di `subjectAltName` l'hostname. Se non si conosce l'hostname della propria macchina basta digitare il comando *hostname* sul terminale.

Per questo esempio il file di configurazione creato verrà chiamato *my_config.conf*.

Partendo quindi dalla directory *C:\Program Files\OpenSSL-Win64\bin,* lanciare il file exe di openssl e inserire i seguenti comandi:

```
[Server Certificates]

req -x509 -nodes -days 355 -newkey rsa:2048 -keyout server_private_key.pem -out
server_certificate.pem -config my_config.conf

x509 -outform der -in server_certificate.pem -out server_certificate.der

[Client Certificates]

req -x509 -nodes -days 355 -newkey rsa:2048 -keyout client_private_key.pem -out
client_certificate.pem -config my_config.conf

x509 -outform der -in client_certificate.pem -out client_certificate.der
```

Una volta creati i certificati, vanno posti all'interno della directory certificates.

Tool di Supporto Utilizzati

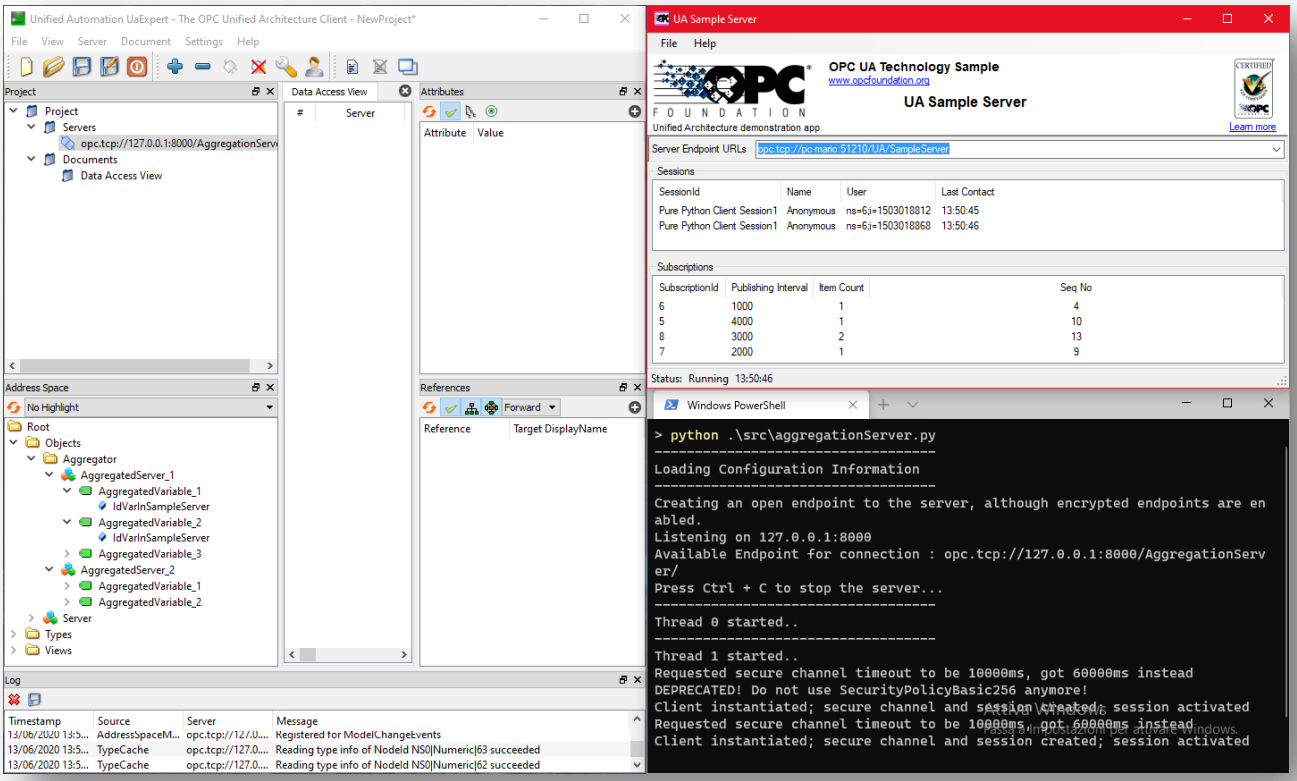
Per verificare il corretto funzionamento dell'Aggregation Server sono stati realizzati dei test utilizzando i seguenti tool:

- **UaExpert** come client per esplorare ed interrogare l'AggregationServer;
- **Sample Server** fornito da OPC foundation come server da aggregare; i node id utilizzati negli esempi presenti sul file di configurazione sono stati presi dall'address space di tale server.

Guida All'utilizzo

Per testare/utilizzare il software è necessario seguire i seguenti step:

- Compilare opportunamente il file di configurazione config.json presente nella directory *config* della repository;
- Avviare il Sample Server (o in alternativa il/i server che si vogliono aggregare);
- Avviare l'aggregation server posizionandosi all'interno della directory principale del progetto AggregationServer, e da terminale lanciare il seguente comando `python .\src\aggregationServer.py`;
- Collegarsi all'aggregation server attraverso UaExpert.



La figura in alto mostra uno screen realizzato durante l'esecuzione di un test. Sulla sinistra è possibile vedere un'istanza di UaExpert collegato all'aggreagation server, infatti in basso a sinistra è possibile vedere l'organizzazione dell'address space con gli AggreagtedServer e le variabili e properties ad essi associate. In alto a destra invece è presente una istanza del Sample Server che mostra che le sessione attive sono due, dato che in configurazione erano stati passati due sample server; il sample server mostra anche le sottoscrizioni attive con le relative caratteristiche che corrispondono alle informazioni passate tramite i campi di sub_infos nel file di configurazione. In basso a destra invece è visibile il terminale con l'aggregation server in esecuzione , tale finestra mostra i messaggi prodotti dal server durante l'esecuzione.