UNIVERSITÀ DEGLI STUDI DI CATANIA

OPC-UA Aggregation Server

Industrial Informatics a.a 2019/2020

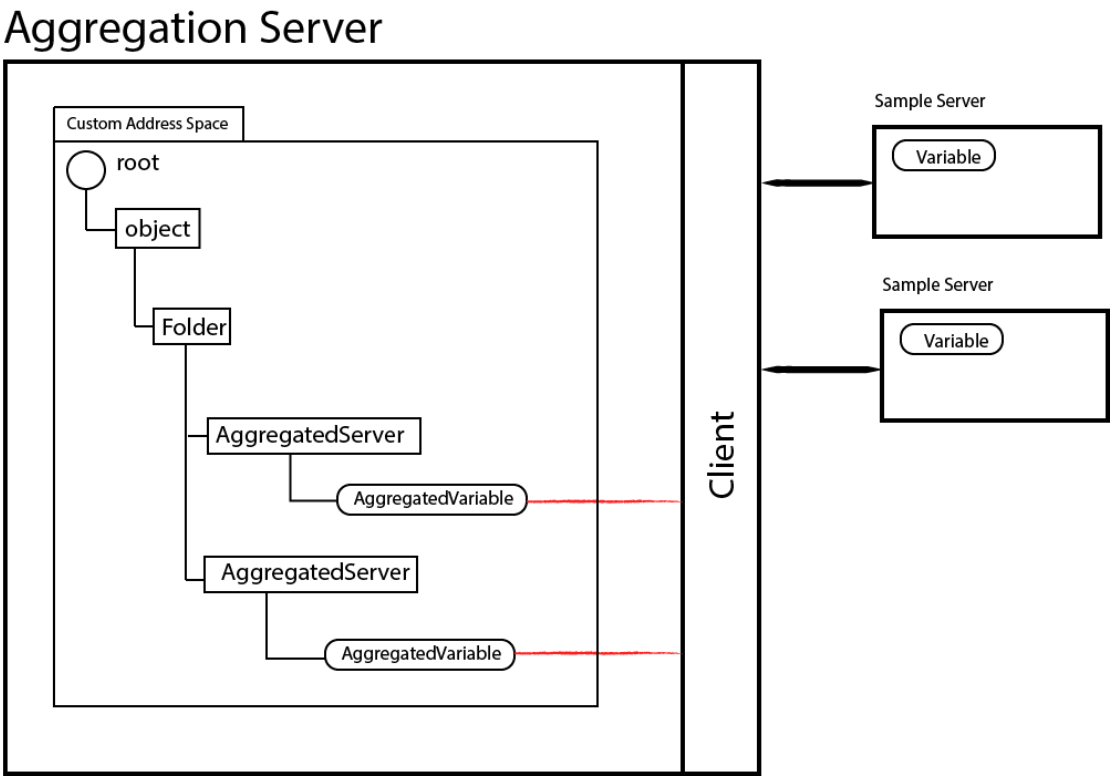
Raiti Mario O55000434
Nardo Gabriele Salvatore O55000430

Lo scopo 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).

Il codice sorgente dell'elaborato è disponibile su github al corrispondente indirizzo (aggiungere link).

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 tal proposito 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 un modulo client che avrà 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 cioè tramite 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 verranno descritti i file di configurazione json utilizzati per il passaggio delle informazioni di configurazione e per la creazione dei certificati x509v3.

Config.json

Questo file contiene un elemento sample server per ogni server che si vuole aggregare. Per ogni elemento sono previsti sei campi da configurare opportunamente per settare le informazioni relative al server e ai dati da tracciare. Di seguito vengono descritti tali campi e per ognuno di essi sarà presentato in basso un esempio di valore ammissibile:

- **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 contente la modalità di sicurezza richiesta; i valori ammissibili sono: None, Sign e SignAndEncrypt;
- **node_id**: deve contenere il node id della variabile di cui si vogliono ottenere i valori sotto forma di stringa formattata nel seguente modo: 'ns=valore;i=valore';
- **variable_type**: deve contenere il tipo della variabile da leggere;
- **service_req**: definisce il tipo di servizio per otternere i dati; i valori ammissibili sono tre: read, write, subscribe;

Se viene richiesto come servizio la *write* bisogna settare il campo **new_value** presente in **write_info** che sarà il nuovo valore che verrà attribuito al nodo del sample server e della copia locale nel aggregation server.

Se viene richiesto come servizio la *subscribe* bisognare settare i sottocampi di **sub_info** che corrispondo alle caratteristiche della subscription e dei relativi monitored item ad essa associati.

- **publish_interval** : definisce il publish interval della subscription in millisecondi;
- **queue_size** : intero che indica la dimensione della coda dei monitored items;
- **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 e **2** se vogliamo una deadband percentuale. Scegliere una dead band percentuale solo se si vuole monitorare un node analog.

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

```
{
   "sample_server1" : {
        "endpoint":"opc.tcp://desktop-v6n8m9j:51210/UA/SampleServer",
        "security_policy":"None",
        "security_mode":"None",
        "node_id":"ns=2;i=11212",
        "variable_type":"DataValue",
        "service_req":"subscribe",
        "write_info":{
            "new_value":""
        },
        "sub_info": {
            "publish_interval": 500,
            "queue_size": 1,
            "deadbandval": 40,
            "deadbandtype": 1
        }
    }
}
```

Openssl_conf.json

In questo file sono presenti due campi da settare opportunamente per la corretta esecuzione del programma, utilizzati per la creazione dei certificati. Anche questo file è in formato *json* ed è composto da due campi:

- **ssl_installation_path:** in questo campo va inserito il proprio path di installazione di openssl.
- **ssl_confing_file_name:** va inserito il nome del file di configurazione di openssl.

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

```
{
    "ssl_installation_path":"C:\Program Files\OpenSSL-Win64\bin\openssl.cfg",
    "ssl_confing_file_name":"openssl.cfg"
}
```

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*.

L'elaborato è stato sviluppato in ambiente Windows (Windows 10 Professional), 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 *VScode*.

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
server = Server()
server.set_endpoint("opc.tcp://127.0.0.1:8000/AggregationServer/")

server.set_security_policy([ua.SecurityPolicyType.NoSecurity,
                            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 creato 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.

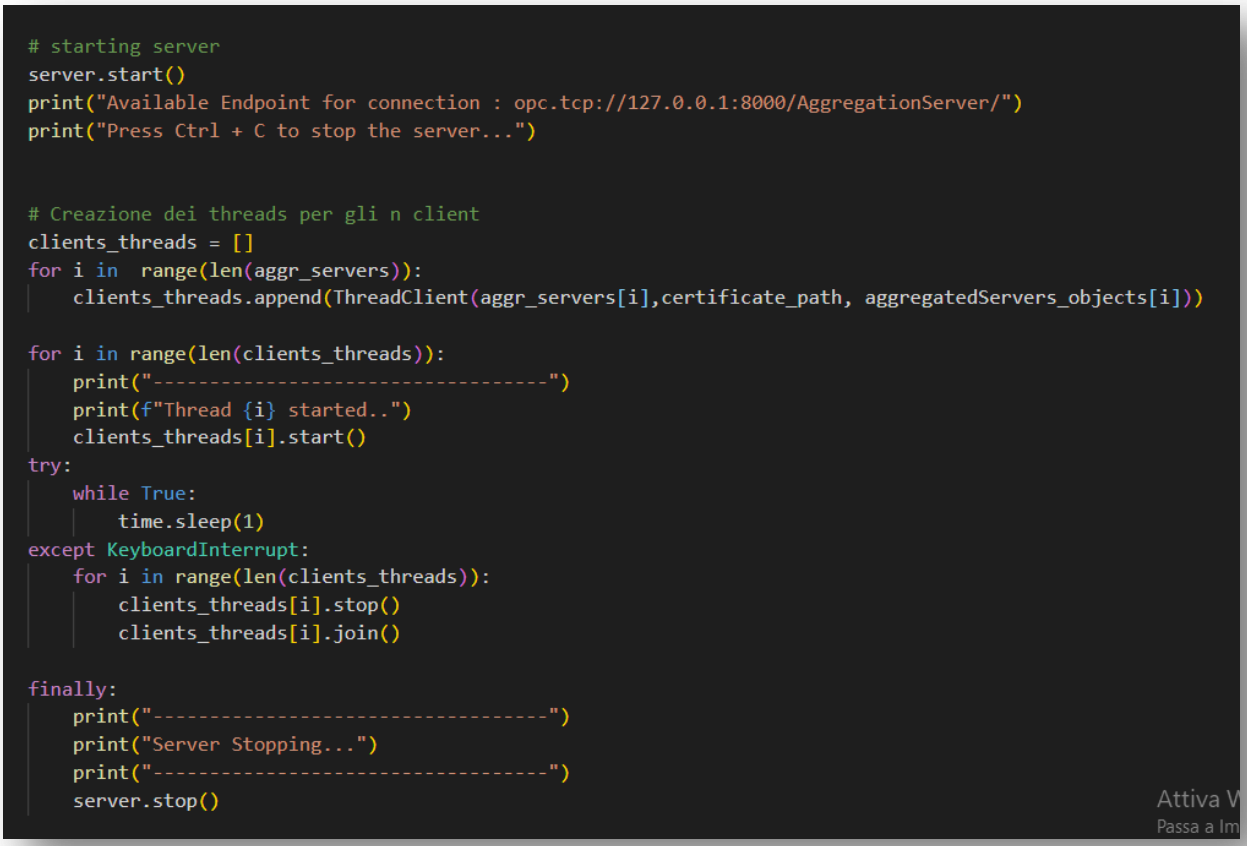
Prima di eseguire la creazione dinamica degli oggetti, si è creato un objectType specifico per rappresentare i server aggregati. Tale oggetto è stato derivato da un BaseObjectType, ed aggiunto al namespace custom col nome di AggregatedServerType, di default a tale tipo è stata aggiunta una variable chiamata AggregatedVariable.

A seguito di ciò dinamicamente 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 aggiuntive 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 contiene n variabili quanti sono i nodeid dai nodi da monitorare.

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

```python
# Setup our namespace
uri = "http://Aggregation.Server.opcua"
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 adn 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")
var = mycustomobj_type.add_variable(idx, "AggregaterdVariable1", 0)
var.set_writable()
var.set_modelling_rule(True)

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)
    for j in range(len(aggr_servers[i]['node_id'].split(",")) -1):
        obj.add_variable(idx,"AggregatedVariable"+str(j+2), 0).set_writable()
    aggregatedServers_objects.append(obj)
```


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 con tali valori, 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 viene passato 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* , questo si scatena 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 il metodo *join()* per attendere la loro conclusione. Infine la clausola finally si occupa di stoppare il server e così conclude l'esecuzione del modulo.

Client.py

(Descrivere in maniera dettagliata il file Client.py)

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) e il modulo threading. 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 ***)*** : 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.
- ***stop***() : questo metodo semplicemente 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 cioè come 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 utilizzando i metodi di tale classe crea il canale sicuro e si connette al server indicato nel file di configurazione. Creata la connessione verifica quale servizio viene specificato dalla informazioni di configurazione ed invoca la rispettiva funzione di Client_opc per eseguire il servizio ed aggiornare le copie locali dell'address space. 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 presenti subscription vengono distrutte, il client si disconnette dal sample server e poi viene eseguito il return terminando così l'esecuzione.

```
def run(self):
        client = Client.Client_opc(self.cert_path , self.sample_server_conf['endpoint'], self.sample_server_conf['security_p
        client.client_instantiate()
        client.secure_channel_and_session_connection()

        if (self.sample_server_conf['service_req'] == "read"):
            client.readData(self.sample_server_conf['node_id'])


        if (self.sample_server_conf['service_req'] == "subscribe"):
            sub, handle = client.subscribe(self.sample_server_conf['node_id'],self.sample_server_conf['sub_info'])

        if (self.sample_server_conf['service_req'] == "write"):
            client.writeData(self.sample_server_conf['node_id'],self.sample_server_conf['write_info']['new_value'])


        while True:
            if self.stopped():
                print("Client Stopping...")
                if (self.sample_server_conf['service_req'] == "subscribe"):
                    client.unsubscribe(sub, handle)
                    client.delete_sub(sub)
                client.disconnect()
                return
```

Risultati

(Qui potremmo discutere i risultati delle esecuzioni con screen)

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 ci 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 *deadband_monitor* (questa funzione permette di settare la tipologia di datachange filter , raltivo valore e la dimensione della coda dei monitored item)*,* chiamata sulla sottoscrizione appena creata che appartiene alla classe Subscription dello stack. *create_subscription* ha come parametri di ingresso il publish_interval e un handler che viene istanziato tramite la classe SubHandler creata nel nostro progetto, ma che deve possedere le funzioni *datachange_notifications* ed *event_notifications* specificate dallo stack. Esse verranno chiamate per notificare un cambiamento dei valori delle variabili. *subscribe_data_change* ha come parametro di ingresso una lista di variabili, ottenute tramite la lista dei node id di partenza. Questa funzione creerà per noi un montored item per ogni variabile della lista e ci tornerà una lista contenente i monitored items id dei monitored items appena creati.

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 *unsubscribe* 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.

Altri Dettagli

I dettagli relativi alla struttura del progetto e all'avvio dell'applicativo sono contenuti all'interno del file ***README*** contenuto nel repository del progetto linkata in alto.