UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Ingegneria Elettrica
Elettronica Informatica

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Relazione Progetto
Industrial Informatics

Server OPC UA – PubSub

Anno accademico 2019-2020

Massimo Gollo O55000403

Alessandro Spallina O55000439

INDICE

Introduzione

Il progetto, presente al seguente link, consiste nella realizzazione di un Server per esporre dati custom, ai quali processi terzi potrebbero essere interessati, sia attraverso il modello di comunicazione Client/Server che attraverso modello PubSub. Il Server è conforme allo standard OPC Unified Architeture, definito in IEC 62541 e sviluppato da OPC Foundation. Lo standard è composto da 14 parti, ognuno delle quali descrive una particolare specifica:

1) Concepts
2) Security Model
3) Address Space Model
4) Services
5) Information Model
6) Mappings
7) Profiles
8) Data Access
9) Alarms and Conditions
10) Programs
11) Historical Access
12) Discovery and Global Service
13) Aggregates
14) PubSub

È rilevante notare che le specifiche dello standard non impongono vincoli a livello applicazione, essendo indipendente dalla piattaforma, ma descrivono invece meccanismi astratti, gestiti e concretizzati attraverso la particolare realizzazione dello stack di comunicazione. Per il presente progetto sono state maggiormente consultate le parti 1) 3) 5) con particolare riferimento a 14).

Per lo sviluppo del server è stata utilizzata la libreria open source in C (99) Open62541 release 1.1, basata su IEC 62541. Tra i vantaggi si evidenziano:

Portabilità - scritta in C99, la libreria esegue su diverse piattaforme e sistemi embedded avendo componenti client/server leggeri, con dimensioni persino minori a 100 kb.

Scalabilità - architettura event-based, single/multi thread.

Flessibilità: - possibilità di modificare l'information model a runtime e generazione di tipi di dato custom.

La documentazione è disponibile al seguente link Open62541. Per l'installazione e la configurazione della libreria e per l'esecuzione del progetto, fare riferimento al readme del progetto.

Idealmente, il server contiene istanze di stazioni meteo per raccogliere misurazioni di temperatura e umidità prelevati da diverse località e aggiornate a runtime. Un Client ha la possibilità di instaurare una connessione con il server utilizzando i diversi endpoints che espone, in base ai sistemi di sicurezza che supporta. Esso può inoltre leggere le informazioni presenti nell'AddressSpace e ricevere quelle che il server pubblica agendo da Publisher Brokerless UDP UADP. Quest'ultima funzionalità è quella che è stata maggiormente attenzionata in questo progetto.

L'ambiente di sviluppo utilizzato è Microsoft Visual Studio 2019 su sistema operativo Windows10. Il Client utilizzato per verificare e ispezionare le funzionalità del Server è UaExpert.

1. PubSub

Introduzione

Il meccanismo PubSub consente la distribuzione di dati ed eventi da una sorgente generica, verso osservatori interessati. Lo standard definisce due ruoli principali che sono assunti dai protagonisti della comunicazione. Si distinguono il ruolo di ***Publisher*** e il ruolo di ***Subscriber***. Questi sono disaccoppiati e indipendenti; ciò rende questo meccanismo particolarmente utile in applicazioni dove è richiesta *scalability* e *location independence*. Risulta vantaggioso rispetto al modello Client/Server in quanto i dati sono automaticamente pubblicati senza che ne sia esplicitamente richiesta la distribuzione. Lo standard, inoltre, non definisce un particolare sistema di messaggistica né ne impone uno. Il meccanismo PubSub è quindi flessibile e agnostico rispetto allo specifico protocollo utilizzato.

I *Publisher*, sorgenti delle informazioni, sono in grado di trasmetterle a diversi *Subscriber* distinti, essendone consumatori, attraverso un ***Message Oriented Middleware***. Il *Message Oriented Middleware* può essere un componente software o un'infrastruttura hardware che consente la distribuzione di messaggi tra applicazioni.

Middleware

Lo standard distingue due tipologie di *Message Oriented Middleware*:

- ***Broker-less***: il *middleware* è un'infrastruttura di rete capace di veicolare messaggi *datagram-based*. Un tipico protocollo utilizzato è UDP.
- ***Broker-based***: il middleware è un broker; publisher e subscriber utilizzano un protocollo di messaggistica standard come AMQP o MQTT per comunicare con il broker. Tutti i messaggi sono pubblicati su code specifiche che il broker espone ai subscriber in ascolto.

Il meccanismo di PubSub può essere integrato e puo' coesistere con il meccanismo Client/Server (request-response). Nello scenario più comune un Server OPC UA assume il ruolo di Publisher, mentre un Client OPC UA funge da Subscriber.

L'assegnazione dei ruoli pero' è arbitraria e lo standard non impone nessun vincolo, quindi potrebbero presentarsi scenari in cui un Server OPC UA è esso stesso Subscriber per un altro server; viceversa potrebbe risultare Subscriber anche per un Client che potrebbe assumere il ruolo di Publisher.

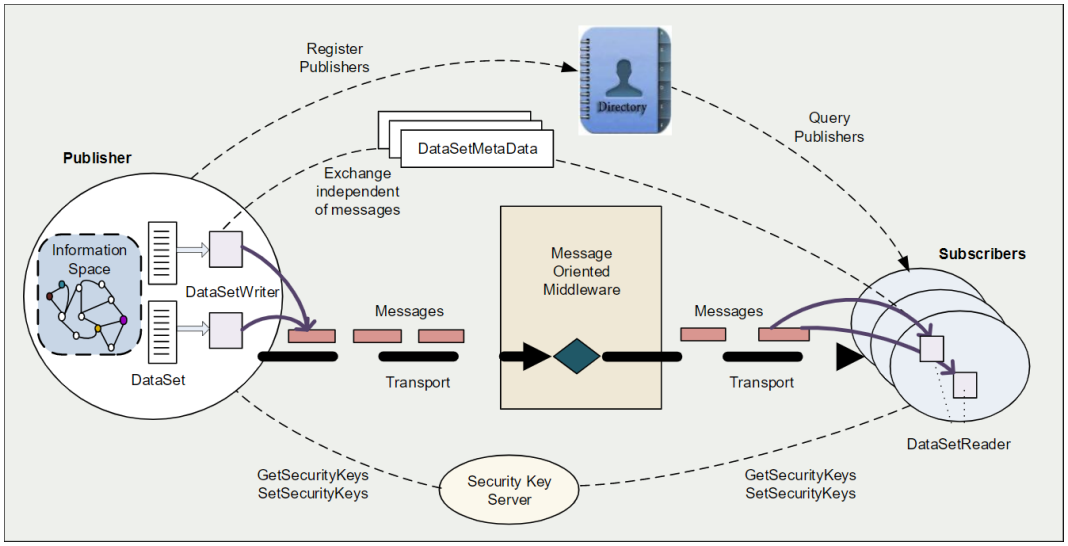


Figura: Meccanismo di funzionamento generale

Concetti generali

Lo standard definisce ***DataSet*** una lista di elementi chiave-valore che rappresentano eventi o campioni di una o più variabili; i campi di un DataSet, ***DataSetField***, possono essere informazioni interne al Publisher oppure dati che esso riceve da altri Publisher o Server. Prima di essere trasmesso, un DataSet viene codificato in un ***DataSetMessage***; uno o più DataSetMessage compongono il payload del ***NetworkMessage***. Questo è il nome del pacchetto effettivamente inviato dal Publisher e ricevuto dai Subscriber in ascolto.

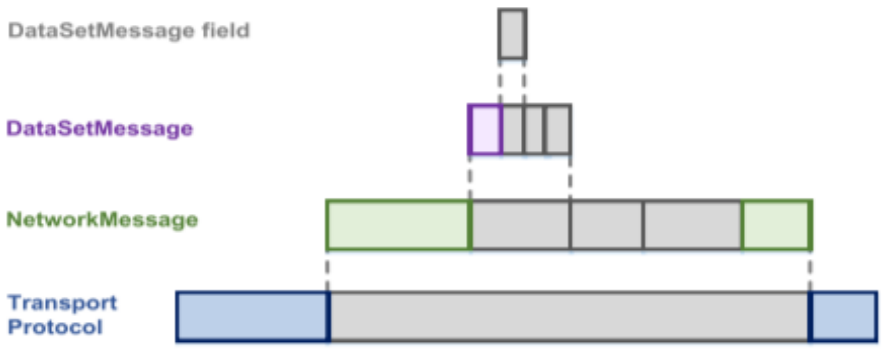


Figura: Composizione dei diversi message

Il NetworkMessage possiede un header e il payload. L'header include informazioni di identificazione del mittente e dati relativi alla sicurezza mentre il payload può contenere uno o più DataSetMessage. I DataSetMessage, che sono creati da un DataSet, possono essere cifrati o firmati qualora le applicazioni supportino tali meccanismi di sicurezza; in questo caso, un ***Security Key Server*** sarà responsabile della distribuzione delle chiavi necessarie e dei meccanismi di sicurezza per la codifica dei messaggi.

Un DataSet, per essere pubblicato, deve essere prima opportunamente codificato in un DataSetMessage. Per la codifica, sicurezza e trasporto, si ricorre ad un componente all'interno del Publisher chiamato ***DataSetWriter***. Esso genera una sequenza continua di DataSetMessage e contiene le impostazioni per la codifica e il trasporto. La gran parte di queste impostazioni dipende dallo specifico Message Oriented Middleware utilizzato.

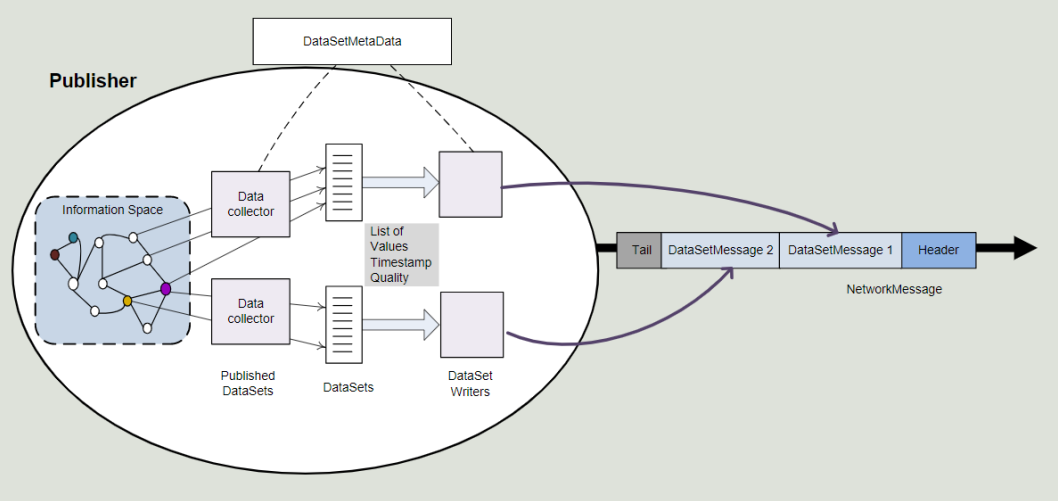


Figura: Composizione tipica di un publisher

L'entità DataSetWriter fa riferimento a uno specifico gruppo chiamato ***WriterGroup***, identificato in maniera univoca da un ID. Esso contiene le informazioni necessarie alla composizione e produzione del NetworkMessage, come ***PublisherId***, ***PublishingInterval***, ***KeepAliveTime*** e ***Security Settings***.

La sintassi e la semantica di un DataSet è descritta dal ***DataSetMetaData***. La definizione dei metadati include una descrizione del DataSet insieme a nome e tipo di ogni DataSetField. L'ordine dei campi nel DataSetMetaData dovrebbe

corrispondere con l'ordine dei campi ricevuti da un Subscriber nel DataSetMessage. I Subscriber utilizzano questi metadati per decodificare i campi contenuti nel pacchetto ricevuto. Ogni DataSetMessage include la versione del DataSetMetaData al quale fa riferimento; ciò consente al subscriber di verificare se è in grado di comprendere il contenuto del pacchetto e se i metadata che possiede sono allineati con quelli del publisher.

```
Name:     "Temperature-Sensor Measurement"
Fields:   [1] Name=DeviceName, Type=String
          [2] Name=Temperature, Type=Float, Unit=Celsius, Range={1,100}
```

Figura: Esempio DataSetMetaData

Lo standard specifica diverse possibilità per far sì che il Subscriber ottenga i DataSetMetaData:

- Il Subscriber è in grado di ottenere le configurazioni necessarie accedendo al nodo PubSub istanziato sul server attraverso la browse dell'AddressSpace del Server
- Il Subscriber riceve i DataSetMetaData come NetworkMessage dal Publisher
- Il Subscriber ha già le configurazioni necessarie (DataSetMetaData hard coded)

I parametri che specificano come acquisire i dati di un DataSet sono racchiusi nel componente logico chiamato ***PublishedDataSet***. Un DataSetMetaData può essere specifico per un singolo PublishedDataSet o lo stesso per più PublishedDataSet configurati su base di una ***DataSetClass***, che altro non è che un template del contenuto del DataSet. Una DataSetClass è identificata univocamente da un ID globale.

Un DataSetMessage consiste di un header e un payload contenente i campi del DataSet codificati. Un DataSetMessage può contenere dettagli differenti; in funzione della ***DataSetMessageContentMask*** si definisce quali campi dovrebbero essere presenti nell'header. Un DataSetMessage non contiene informazioni riguardanti le modalità di acquisizione dei dati o informazioni sulla sorgente,

essendo queste delegate al *PublishedDataset*. Le informazioni presenti nell'header possono essere:

- *DataSetWriterId*: identifica il DataSetWriter e indirettamente il PublishedDataSet
- *Sequence Number*: numero incrementato per ogni DataSetMessage.uò essere utilizzato per verificare l'ordinamento dei messaggi e trovare eventuali pacchetti mancanti
- *Timestamp*: descrive l'istante in cui il dato in questo pacchetto è stato ottenuto
- *Versione*: versione del DataSetMetaData
- *Status*: informazioni sullo stato del dato nel pacchetto
- *Keep Alive*: utilizzato per notificare al Subscriber che il Publisher è ancora in esecuzione e che non ha nuovi dati da mandare

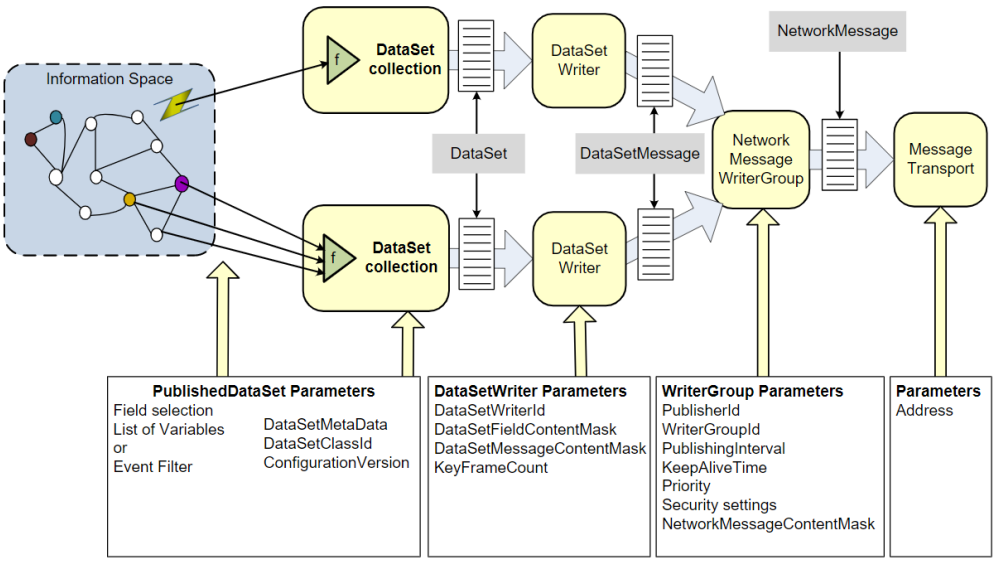


Figura: Publisher nel dettaglio

Subscriber

I Subscriber sono i consumatori dei NetworkMessage provenienti dal Message Oriented Middleware. Per determinare a quali DataSetMessage siano interessati e a quale Middleware sottoscriversi, essi devono essere opportunamente configurati, oppure utilizzare un meccanismo di discovery per la configurazione

automatica. E' necessario che siano tolleranti alla ricezione di messaggi non comprensibili o di poca rilevanza. Ogni NetworkMessage fornisce campi nell'header, non cifrati, atti all'identificazione e filtraggio, in funzione di Publishers, DataSetMessages, DataSetClasses o altre informazioni rilevanti. Se il NetworkMessage è firmato o firmato e cifrato, il subscriber necessita delle chiavi di sicurezza per verificare la firma e decifrare il DataSetMessage che esso contiene.

Filtrato il DataSetMessage ritenuto rilevante e di interesse, esso viene inoltrato al componente logico ***DataSetReader***, corrispettivo speculare del DataSetWriter, per la decodifica in un DataSet. L'informazione risultante può essere a questo punto processata.

Il processo di ricezione, decodifica e inoltro dei messaggi è schematizzato di seguito. Come per il publisher, il subscriber deve selezionare il Message Oriented Middleware e stabilire una connessione utilizzando l'indirizzo specifico. Una connessione è stabilita attraverso un indirizzo multicast qualora si utilizzi **UDP UADP** come profilo di trasporto, oppure attraverso l'indirizzo del broker, se si utilizza un approccio BrokerBased. Una volta sottoscritto, il subscriber è in ascolto dei messaggi. Il processo inizia con l'arrivo di un NetworkMessage. Dopo l'analisi dell'header e stabilito l'interesse verso il messaggio esso viene decifrato e decodificato, ricavando il DataSetMessage attraverso DataSetReader. Qui è utilizzato il DataSetMetadata per la decodifica in un DataSet. Si ricorda che il DataSetMetaData fornisce la sintassi completa di tutti campi, includendo nome, tipo di dato e altre proprietà rilevanti come unità di misura e range.

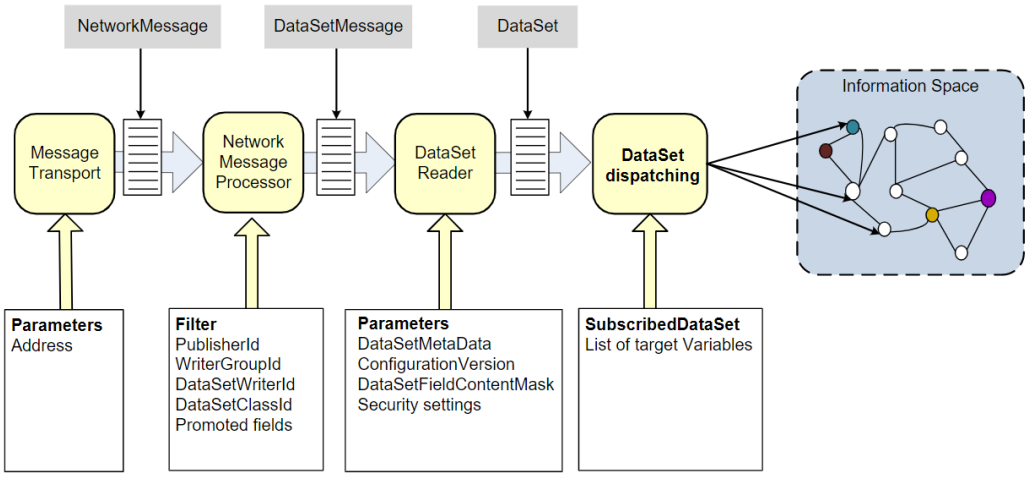


Figura: Subscriber nel dettaglio

Message Oriented Middleware Broker-less

Con l'approccio Broker-less, OPC UA PubSub si affida ad un'infrastruttura di rete per la consegna dei NetworkMessage a uno o piu ricevitori. Un esempio potrebbe essere una switched network che utilizza UDP unicast o multicast. I vantaggi di questo approccio sono:

1) Richiesto soltanto equipaggiamento standard di rete, senza componentI software addizionali come un Broker.
2) La consegna dei messaggi è diretta, senza intermediari software, riducendo latenza e overhead di comunicazione.
3) UDP supporta multipli subscriber utilizzando l'indirizzamento multicast.

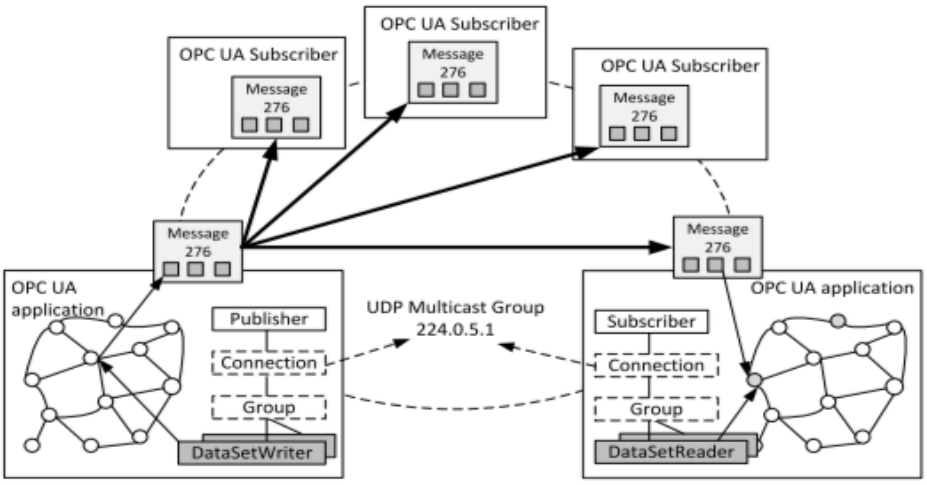


Figure 14 – UDP Multicast Overview

L’oggetto ***PublishSubscribe***, identificato come nodo all’interno dell’AddressSpace, contiene a sua volta un oggetto Connection per ogni indirizzo IP unicast/multicast. La connessione puo’ avere uno o piu WriterGroup che contengono diversi DataSetWriters. Un gruppo puo’ pubblicare DataSet ad un determinato PublishingInterval. In ogni intervallo viene collezionato un DataSet per un determinato PublishedDataset. Per ogni DataSet si crea un DataSetMessage, composto a sua volta in NetworkMessage, impacchettato in un datagramma e inviato all’indirizzo IP designato.

Con UADP UDP si riduce l’overhead di comunicazione e la latenza, a discapito però della mancanza di garanzia di tempistiche limitate, consegna e ricezione, ordinamento o duplicazione dei messaggi.

2. Implementazione

Configurazione e ciclo di vita

Il comportamento dell'applicazione server è condizionato dal modo in cui esso viene eseguito. È stato realizzato in modo tale da poter specificare argomenti da riga di comando al momento di avvio per abilitare/disabilitare i meccanismi di sicurezza e personalizzare l'URL per la trasmissione di specifici dati esposti attraverso la comunicazione PubSub Brokerless.

Il server è implementato nel progetto denominato OPC_UA_Server. La stessa soluzione Visual Studio comprende inoltre anche un Sample Subscriber che ha il semplice compito di visualizzare il set di dati al quale è sottoscritto. Il punto di ingresso al programma si trova nel file *main.c*, dove si definisce il ciclo di vita dell'applicazione e i parametri di configurazione. Per la creazione, l'esecuzione e la terminazione del server, Open62541 fornisce un meccanismo basilare ma efficace:

```
1.  volatile UA_Boolean running = true;
2.
3.  //Shutdown del server al segnale
4.  void stopHandler(int sign) {
5.   UA_LOG_INFO(UA_Log_Stdout, UA_LOGCATEGORY_SERVER, "Received ctrl-c");
6.   running = false;
7.  }
```

Si utilizza una variabile globale volatile booleana *running* per determinare lo stato del server. Fintantoché la variabile è settata *true,* il server continuerà a servire le richieste; alla ricezione del segnale SIGINT/SIGTERM, la variabile cambierà il suo stato in *false* e il server terminerà.

Preliminarmente è necessario istanziare l'entità *server* e una struct di configurazione dove inserire tutte le informazioni necessarie per il suo funzionamento; agendo su questa si modifica il comportamento del Server OPC UA.

```
1.  //Inizializza il server e ServerConfig
2.  UA_Server *server = UA_Server_new();
3.  UA_ServerConfig *config = UA_Server_getConfig(server);
```

Per modulare il comportamento dell'applicazione (e a sua volta anche del server) in funzione degli argomenti da riga di comando che riceve, è stata realizzata una funzione *Parser* che accetta i valori attuali degli argomenti. Se vengono forniti i *path* del certificato digitale e della chiave privata del server, si procede all'abilitazione di tutti i livelli di sicurezza, offrendo sia firma digitale che cifratura. Se non vengono passati, di default, non si utilizza nessun meccanismo di sicurezza e si espone un unico endpoint con *security-None*. Il *parsing* restituisce una *struct* di configurazione personalizzata *appConf,* la cui definizione è disponibile sul file *utils.h*. Il sorgente contiene inoltre la funzione di caricamento *loadFile* per il certificato e la chiave.

Da notare che le due struct di configurazione hanno ruoli diversi. *AppConf* rappresenta la configurazione dell'intera applicazione; *Config* rappresenta la configurazione dell'astrazione del Server OPC UA fornito dall'SDK .

```
1.  if (appConf.encryption) {
2.                  /* Carica certificato e chiave se esiste */
3.                  UA_ByteString certificate = loadFile(appConf.certPath);
4.                  UA_ByteString privateKey = loadFile(appConf.keyPath);
5.
6.                  retval = UA_ServerConfig_setDefaultWithSecurityPolicies(c
    onfig, 4840, &certificate, &privateKey, NULL, 0, NULL, 0, NULL, 0);
7.          } else {
8.                  retval = UA_ServerConfig_setDefault(config);
9.          }
10.
11.         if (retval != UA_STATUSCODE_GOOD) {
12.                 UA_LOG_INFO(UA_Log_Stdout, UA_LOGCATEGORY_SERVER, "Error
    initializating server");
13.                 UA_Server_delete(server);
14.                 return retval == UA_STATUSCODE_GOOD ? EXIT_SUCCESS : EXIT
    _FAILURE;
15.         }
```

Maggiori dettagli sulle modalità di esecuzione del programma sono forniti in readme.md presente nel repository del progetto.

Information Modelling

La modellazione delle informazioni in OPC UA (e specularmente in Open62541) si basa fortemente sul concetto di *object-orientation* e *semantic modelling*. Tutte le informazioni sono rappresentate in un grafo orientato tramite Nodi (astrazione di qualsiasi tipo di oggetto). Ogni nodo contiene Attributi e Reference (relazioni tipizzate e dirette tra due nodi) ed è identificato da **NodeId** unico all'interno del server. È possibile ottenere oggetti complessi come combinazione di nodi elementari legati tramite reference.
Per modellare le informazioni e i dati ottenuti da un ideale stazione meteo, si è definito un ***ObjectType*** personalizzato, realizzato tramite la funzione ***defineObjectTypeWeather***. ***WeatherType*** rappresenta l'astrazione di una stazione, fornisce misurazioni di temperatura e umidità ed è definito come sottotipo di ***BaseObjectType*** (NodeClass built-in in OPC UA). WeatherType non contiene *value,* ma piuttosto è un contenitore per altri *DataVariable*. In questo caso, ad un nodo (oggetto) stazione sono legate le variabili (attributi) CityName, TemperatureVariabile e HumidityVariable.

```
1.  UA_NodeId defineObjectTypeWeather(UA_Server* server) {
2.
3.      UA_NodeId weatherId;
4.
5.      //Definizione dell'oggetto weatherType
6.      UA_ObjectTypeAttributes otAttr = UA_ObjectTypeAttributes_default;
7.      otAttr.description = UA_LOCALIZEDTEXT("en-US", "Weather Type");
8.      otAttr.displayName = UA_LOCALIZEDTEXT("en-US", "WeatherType");
9.      UA_Server_addObjectTypeNode(server, UA_NODEID_NUMERIC(1, 0), UA_NODE
    ID_NUMERIC(0, UA_NS0ID_BASEOBJECTTYPE),
10.         UA_NODEID_NUMERIC(0, UA_NS0ID_HASSUBTYPE), WEATHER_OBJECTTYPE_QU
    ALIFIEDNAME, otAttr, NULL, &weatherId);
11.
12.     //Primo attributo - Nome della localita meteo
13.     UA_NodeId cityNameId;
14.     UA_VariableAttributes vAttr = UA_VariableAttributes_default;
15.     vAttr.description = UA_LOCALIZEDTEXT("en-
    US", "Name of the city for which record Temp/Hum");
16.     vAttr.displayName = UA_LOCALIZEDTEXT("en-US", "CityName");
17.     vAttr.dataType = UA_TYPES[UA_TYPES_STRING].typeId;
18.     vAttr.valueRank = UA_VALUERANK_SCALAR;
19.     UA_Server_addVariableNode(server, UA_NODEID_NUMERIC(1, 0), weatherId
    , UA_NODEID_NUMERIC(0, UA_NS0ID_HASCOMPONENT),
20.         CITYNAME_VARIABLE_QUALIFIEDNAME, UA_NODEID_NUMERIC(0, UA_NS0ID_B
    ASEDATAVARIABLETYPE), vAttr, NULL, &cityNameId);
21.     //variabile obbligatoria
22.     UA_Server_addReference(server, cityNameId,
23.         UA_NODEID_NUMERIC(0, UA_NS0ID_HASMODELLINGRULE),
24.         UA_EXPANDEDNODEID_NUMERIC(0, UA_NS0ID_MODELLINGRULE_MANDATORY),
    true);
```

Il primo attributo è la variabile scalare *CityName*, di tipo stringa, che identifica la località della stazione meteo. Per le variabili temperatura e umidità, entrambe di tipo Float, si associato anche le *properties* che ne definiscono la semantica. In particolare, la variabile temperatura conterrà l'unità di misura e il range, mentre per umidità si specifica solo il range essendo essa una misura relativa.

```
1.  //Secondo attributo - Temperatura della citta
2.     UA_NodeId tempId;
3.     UA_VariableAttributes tmpAttr = UA_VariableAttributes_default;
4.     tmpAttr.description = UA_LOCALIZEDTEXT("en-
    US", "TemperatureVariable of City");
5.     tmpAttr.displayName = UA_LOCALIZEDTEXT("en-US", "Temperature");
6.     tmpAttr.accessLevel = UA_ACCESSLEVELMASK_READ | UA_ACCESSLEVELMASK_WR
    ITE;
7.     tmpAttr.dataType = UA_TYPES[UA_TYPES_FLOAT].typeId;
8.     tmpAttr.valueRank = UA_VALUERANK_SCALAR;
9.     UA_Float tmp = 0.0;
10.    UA_Variant_setScalar(&tmpAttr.value, &tmp, &UA_TYPES[UA_TYPES_FLOAT])
    ;
11.    UA_Server_addVariableNode(server, UA_NODEID_NUMERIC(1, 0), weatherId,
     UA_NODEID_NUMERIC(0, UA_NS0ID_HASCOMPONENT),
12.        TEMPERATURE_VARIABLE_QUALIFIEDNAME, UA_NODEID_NUMERIC(0, UA_NS0ID
    _BASEANALOGTYPE), tmpAttr, NULL, &tempId);
13.    //variabile obbligatoria
14.    UA_Server_addReference(server, tempId, UA_NODEID_NUMERIC(0, UA_NS0ID_
    HASMODELLINGRULE),
15.        UA_EXPANDEDNODEID_NUMERIC(0, UA_NS0ID_MODELLINGRULE_MANDATORY), t
    rue);
16.
17.    //Properties della Temperatura
18.    UA_NodeId temp_range;
19.    UA_VariableAttributes rangePropAttr = UA_VariableAttributes_default;

20.    rangePropAttr.displayName = UA_LOCALIZEDTEXT("en-
    US", "InstrumentRange");
21.    UA_String rangePropName = UA_STRING("-100:100");
22.    UA_Variant_setScalar(&rangePropAttr.value, &rangePropName, &UA_TYPES[
    UA_TYPES_STRING]);
23.    UA_Server_addVariableNode(server, UA_NODEID_NULL, tempId, UA_NODEID_N
    UMERIC(0, UA_NS0ID_HASPROPERTY),
24.        UA_QUALIFIEDNAME(1, "RangeTemperatureQualifiedName"), UA_NODEID_N
    UMERIC(0, UA_NS0ID_PROPERTYTYPE), rangePropAttr, NULL, &temp_range);
25.    //variabile obbligatoria
26.    UA_Server_addReference(server, temp_range, UA_NODEID_NUMERIC(0, UA_NS
    0ID_HASMODELLINGRULE),
27.        UA_EXPANDEDNODEID_NUMERIC(0, UA_NS0ID_MODELLINGRULE_MANDATORY), t
    rue);
```

Il ritorno è della funzione è il *NodeID* assegnato dinamicamente al fine di utilizzarlo nelle successive istanze delle stazioni. Avendo definito un tipo, esso puo' essere istanziato all'interno dell'AddressSpace, tramite la funzione *InstantiateWeatherObject.*

```
1.  UA_NodeId instantiateWeatherObject(UA_Server *server, UA_NodeId wtype, c
    har* locatioName) {
2.              UA_NodeId istantiatedObject;
3.              UA_ObjectAttributes oAttr = UA_ObjectAttributes_default;
4.              oAttr.displayName = UA_LOCALIZEDTEXT("en-
    ES", locatioName);
5.              oAttr.description = UA_LOCALIZEDTEXT("en-
    US", "IstanceCity");
6.              UA_Server_addObjectNode(server, UA_NODEID_NUMERIC(1, 0), UA_
    NODEID_NUMERIC(0, UA_NS0ID_OBJECTSFOLDER),
7.                                      UA_NODEID_NUMERIC(0, UA_NS0ID_ORGANI
    ZES), UA_QUALIFIEDNAME(1, locatioName), wtype,
8.                                      oAttr, NULL, &istantiatedObject);
9.
10.             UA_NodeId citynameVariable = findNodeIdByBrowsename(server,
    istantiatedObject, CITYNAME_VARIABLE_QUALIFIEDNAME);
11.
12.             //Inizializza variabile cityName
13.             UA_String cityName = UA_STRING(locatioName);
14.             UA_Variant value;
15.             UA_Variant_setScalar(&value, &cityName, &UA_TYPES[UA_TYPES_S
    TRING]);
16.             UA_Server_writeValue(server, citynameVariable, value);
17.
18.             return istantiatedObject;
19.  }
```

La funzione riceve in ingresso il NodeID che identifica il tipo dell'oggetto da istanziare e una stringa per inizializzare la variabile CityName, utile anche ad impostare il nome dell'oggetto istanziato. La struct UA_ObjectAttributes contiene la definizione degli attributi di un oggetto. Con la funzione *UA_Server_addObjectNode* si aggiunge un oggetto WeatherType all'interno di Objects Folder presente in AddressSpace; essa rappresenta il nodo che contiene le istanze correnti all'interno del server. Vista l'assegnazione dinamica dei NodeID, è stata realizzata la funzione *findNodeIdByBrowseName* per recuperare l'ID della variabile CityName all'interno dell'oggetto appena istanziato al fine di inizializzarne il valore.

```
1.  UA_NodeId instantiateWeatherObject(UA_Server *server, UA_NodeId wtype, c
    har* locatioName) {
2.
3.              UA_NodeId istantiatedObject;
4.              UA_ObjectAttributes oAttr = UA_ObjectAttributes_default;
5.              oAttr.displayName = UA_LOCALIZEDTEXT("en-
    US", locatioName);
6.              oAttr.description = UA_LOCALIZEDTEXT("en-
    US", "IstanceCity");
7.              UA_Server_addObjectNode(server, UA_NODEID_NUMERIC(1, 0), UA_
    NODEID_NUMERIC(0, UA_NS0ID_OBJECTSFOLDER),
8.                                      UA_NODEID_NUMERIC(0, UA_NS0ID_ORGANI
    ZES), UA_QUALIFIEDNAME(1, locatioName), wtype,
9.                                      oAttr, NULL, &istantiatedObject);
10.
```

```
            UA_NodeId citynameVariable = findNodeIdByBrowsename(server, istantiatedObject, CITYNAME_VARIABLE_QUALIFIEDNAME);

            //Inizializza variabile cityName
            UA_String cityName = UA_STRING(locatioName);
            UA_Variant value;
            UA_Variant_setScalar(&value, &cityName, &UA_TYPES[UA_TYPES_STRING]);
            UA_Server_writeValue(server, citynameVariable, value);

            return istantiatedObject;
 }
```

All'interno del sorgente main.c, le funzioni sono richiamate prima dell'istruzione di esecuzione del server nel seguente modo al fine di definire il tipo e istanziare N oggetti:

```
// Generazione nuovo ObjecType -> WeatheType. Ritorna il NodeId che identifica il tipo
wtype = defineObjectTypeWeather(server);

// Definisco N istanze delle stazioni meteo. Ritorna il NodeId che identifica le istanze
const exposedNode_t weatherStations[WEATHER_STATIONS_COUNT] = {
    {"Catania",      instantiateWeatherObject(server, wtype, "Catania")},
    {"Enna",         instantiateWeatherObject(server, wtype, "Enna")},
    {"Palermo",      instantiateWeatherObject(server, wtype, "Palermo")},
    {"Agrigento",    instantiateWeatherObject(server, wtype, "Agrigento")},
    {"Siracusa",     instantiateWeatherObject(server, wtype, "Siracusa")},
    {"Trapani",      instantiateWeatherObject(server, wtype, "Trapani")},
    {"Ragusa",       instantiateWeatherObject(server, wtype, "Ragusa")},
    {"Caltanissetta", instantiateWeatherObject(server, wtype, "Caltanissetta")},
};
```

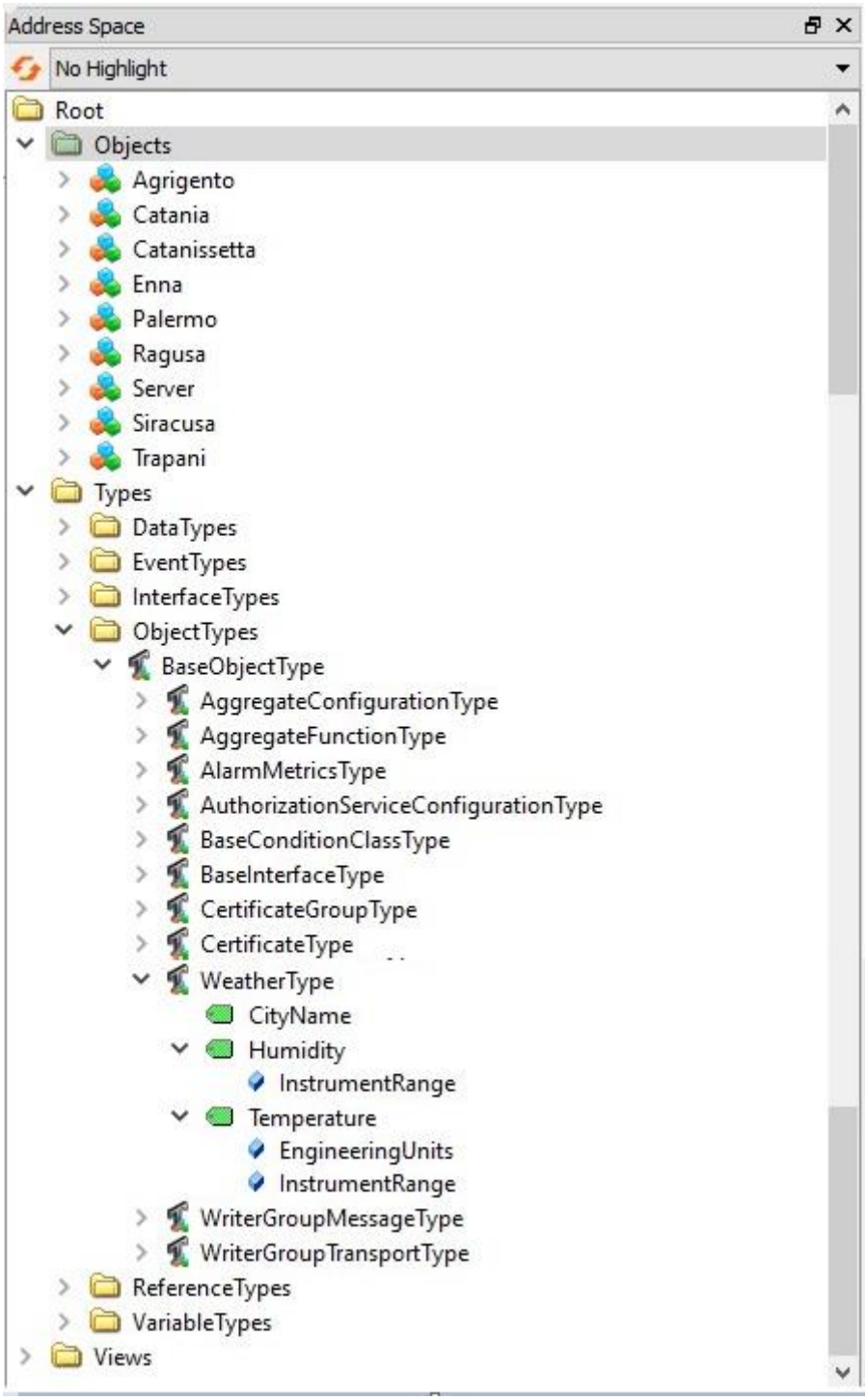
exposedNode_t è una struct personalizzata definita come:

```
typedef struct exposedNode {
    char nodeName[120];
    UA_NodeId nodeId;
} exposedNode_t;
```

utilizzata successivamente per un duplice scopo:

1) Associare delle *callback* ad ogni oggetto
2) Estrarre e preparare i dati da pubblicare tramite meccanismo PubSub

Il risultato della modellazione delle informazioni è il seguente. Esplorando il server attraverso un client OPC UA è possibile notare WeatherType come sottotipo di BaseObjectType, composto dalle 3 variabili specificate, mentre in Object Folder sono presenti le istanze delle stazioni.



Il meccanismo di aggiornamento delle variabili è descritto al paragrafo successivo.

Callback

Nelle architetture basate su OPC UA, principalmente in contesto industriale, un server colleziona le informazioni da una sorgente fisica e vari client consumano dati a runtime. Quando un valore cambia correntemente, l'aggiornamento del valore tramite un'operazione ciclica potrebbe richiedere molte risorse. Attraverso le callback è possibile sincronizzare il valore di una variabile con una rappresentazione esterna, come una richiesta di lettura o scrittura. Associando una callback alla variabile, essa viene eseguita prima di ogni lettura e dopo ogni scrittura [Docs Open62541].

Per il presente progetto l'utilizzo di callback è stato ritenuto adeguato per l'aggiornamento del dato, concentrando lo studio sulla correlazione tra lo standard IEC 625411 e SDK open62541. Non avendo a disposizione una sorgente fisica di dati, si simulano valori di ipotetici sensori, generati ogni volta che viene effettuata una lettura sulla variabile interessata. Qualora si avesse a disposizione uno o più sensori e quindi valori reali, sarebbe necessario esplicitare il comportamento del server per la lettura dell'informazione dal processo fisico e la memorizzazione nella variabile specifica. Ciò sarebbe possibile adottando un approccio multithreading per l'acquisizione, svincolandosi cosi dalle operazioni di lettura/scrittura effettuate dai client. Si pensa all'applicazione reale di questo meccanismo in tutte quelle casistiche in cui il dato non è necessario solo ad un generico Client, ma quando esso sia utile alla generazione di eventi interni al server. In accordo con lo standard, Open62541 permette anche la pubblicazione di eventi oltre che valori.

Ricordando la struct descritta nel paragrafo precedente, che contiene nome e NodeID degli oggetti istanziati, attraverso un ciclo *for* si ricavano i NodeID delle specifiche variabili temperatura e umidità contenute all'interno di ogni istanza

```
1.  // Setto le callback per le letture e scritture sulle variabili temperat
    ure e humididty
2.          for (int i = 0; i < WEATHER_STATIONS_COUNT; i++) {
3.              UA_NodeId temperature = findNodeIdByBrowsename(server, weathe
    rStations[i].nodeId, TEMPERATURE_VARIABLE_QUALIFIEDNAME);
4.              addValueCallbackToVariable(server, temperature, beforeReadTem
    perature, afterWriteTemperature);
5.
6.              UA_NodeId humidity = findNodeIdByBrowsename(server, weatherSt
    ations[i].nodeId, HUMIDITY_VARIABLE_QUALIFIEDNAME);
7.              addValueCallbackToVariable(server, humidity, beforeReadHumidi
    ty, afterWriteHumidity);
8.          }
```

Ad ogni variabile si associa una callback attraverso la funzione *addValueCallbackToVariable,* definita all'interno del sorgente *informationmodel.c* Questa riceve puntatori a funzioni che definiscono le routine da eseguire prima di una read e dopo una write sulla variabile identificata dal nodeID.

```
1.  void addValueCallbackToVariable(UA_Server* server, UA_NodeId variableToU
    date,
2.      void (*beforeReadCallback)(UA_Server *, const UA_NodeId *, void *, c
    onst UA_NodeId *, void *, const UA_NumericRange *, const UA_DataValue *)
    ,
3.      void (*afterWriteCallback)(UA_Server*, const UA_NodeId*, void*, cons
    t UA_NodeId*, void*, const UA_NumericRange*, const UA_DataValue*)) {
4.      UA_NodeId currentNodeId = variableToUdate;
5.      UA_ValueCallback callback;
6.      callback.onRead = (*beforeReadCallback);
7.      callback.onWrite = (*afterWriteCallback);
8.      UA_Server_setVariableNode_valueCallback(server, currentNodeId, callb
    ack);
9.  }
```

Le callback *beforeReadTemperature*, *beforeReadHumidity* sono definite all'interno del sorgente *sensors.c*

```
1.  void beforeReadTemperature(UA_Server* server, const UA_NodeId* sessionId
    , void* sessionContext,
2.      const UA_NodeId* nodeid, void* nodeContext, const UA_NumericRange* r
    ange, const UA_DataValue* data) {
3.
4.      getFakeTemperature(server, *nodeid);
5.  }
6.
7.  void afterWriteTemperature(UA_Server* server, const UA_NodeId* sessionId
    , void* sessionContext,
8.      const UA_NodeId* nodeId, void* nodeContext, const UA_NumericRange* r
    ange, const UA_DataValue* data) {
9.      UA_LOG_INFO(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND, "Temperature upd
    ated");
10. }
11.
12. void beforeReadHumidity(UA_Server* server, const UA_NodeId* sessionId, v
    oid* sessionContext,
13.     const UA_NodeId* nodeid, void* nodeContext, const UA_NumericRange* r
    ange, const UA_DataValue* data) {
```

```
14.
15.     getFakeHumidity(server, *nodeid);
16. }
17.
18. void afterWriteHumidity(UA_Server* server, const UA_NodeId* sessionId, v
    oid* sessionContext,
19.     const UA_NodeId* nodeId, void* nodeContext, const UA_NumericRange* r
    ange, const UA_DataValue* data) {
20.     UA_LOG_INFO(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND, "Humidity update
    d");
21. }
```

Le callback di lettura chiamano le funzioni di generazione di valori random comprensive della funzione di scrittura di questo valore sulla variabile. In uno scenario realistico, queste funzioni potrebbero essere sostituite dalle funzioni di campionamento del dato dal processo fisico.

```
1.  void getFakeTemperature(UA_Server* server, UA_NodeId node) {
2.      static UA_Float fakeTemp = 20.0;
3.      UA_Variant value;
4.
5.      UA_Float deltaTemperature = (UA_Float)(rand() % 10 / 10.0);
6.      rand() % 2 ? fmodf((fakeTemp -
    = deltaTemperature), 100.0) : fmodf((fakeTemp += deltaTemperature), 100.
    0);
7.
8.      UA_Variant_setScalar(&value, &fakeTemp, &UA_TYPES[UA_TYPES_FLOAT]);

9.      UA_Server_writeValue(server, node, value);
10. }
11.
12. void getFakeHumidity(UA_Server* server, UA_NodeId node) {
13.     static UA_Float fakeHum = 70.0;
14.     UA_Variant value;
15.
16.     UA_Float deltaHumidity = (UA_Float)(rand() % 10 / 10.0);
17.     rand() % 2 ? fabs(fmodf((fakeHum -
    = deltaHumidity), 100.0)) : fabs(fmodf((fakeHum += deltaHumidity), 100.0
    ));
18.
19.     UA_Variant_setScalar(&value, &fakeHum, &UA_TYPES[UA_TYPES_FLOAT]);
20.     UA_Server_writeValue(server, node, value);
21. }
```

PubSub

L'implementazione del meccanismo PubSub in Open62541 consiste nella realizzazione delle componenti astratte descritti nello standard. Per il server, che funge il ruolo di Publisher, si definiscono: ***Connection***, ***PublishedDataSet***, ***DataSetWriter*** e ***WriterGroup***. Il Subscriber sarà analizzato nel capitolo successivo. All'interno del sorgente *main.c* del Server (Publisher) si imposta il profilo di trasporto, l'indirizzo multicast e i DataSetField da pubblicare.

```
1.  //PubSub abilitato con profilo UDP UADP
2.  if (appConf.usingUdpUadp) {
3.      //Set del profilo e del muticast Address
4.      UA_String transportProfile = UA_STRING("http://opcfoundation.org/UA-
    Profile/Transport/pubsub-udp-uadp");
5.      UA_NetworkAddressUrlDataType networkAddressUrl = { UA_STRING_NULL, U
    A_STRING("opc.udp://224.0.0.22:4840/") };
6.
7.      //Custom Url Multicast
8.      if (appConf.customUrl != NULL) {
9.          networkAddressUrl.url = UA_STRING(appConf.customUrl);
10.     }
11.
12.     // array che detiene i campi da publicare nel dataset message => dec
    idiamo di pubblicare le temperature di tutte le stazioni meteo
13.     exposedNode_t fieldsToPublish[WEATHER_STATIONS_COUNT];
14.
15.     //preparazione dei campi da pubblicare
16.     for (int i = 0; i < WEATHER_STATIONS_COUNT; i++) {
17.         char tmp[120];
18.         sprintf(tmp, "%sTemperature", weatherStations[i].nodeName);
19.         fieldsToPublish[i].nodeName = tmp;
20.         fieldsToPublish[i].nodeId = findNodeIdByBrowsename(server, weath
    erStations[i].nodeId, TEMPERATURE_VARIABLE_QUALIFIEDNAME);
21.     }
22.
23.     // configura udp uadp
24.     addPubSubUdpToServerConfig(server, config);
25.     configurePubSubNetworkMessage(server, transportProfile, networkAddre
    ssUrl, WEATHER_STATIONS_COUNT, fieldsToPublish);
26. }
```

Il profilo di trasporto utilizzato è UDP UADP e se l'utente non specifica un *customUrl* si imposta un indirizzo di default "opc.udp://224.0.0.22:4840/".

A scopo didattico si sceglie di pubblicare le sole temperature delle stazioni meteo. Per far ciò è necessario prima preparare i dati che questo caso vengono inseriti nella struct *fieldToPublish*. Essa conterrà un identificativo di tipo stringa e il nodeID della variabile da cui prelevare il dato. Successivamente vengono chiamate le funzioni *AddPubSubUpdToServerConfig* e *configurePubSubNetworkMessage*, la cui definizione è presente nel sorgente *pubsub.c*.

AddPubSubUpdToServerConfig è necessaria per allocare all'interno della struct *config* del server i campi che conterranno i parametri per instaurare la connessione.

```
1.  void addPubSubUdpToServerConfig(UA_Server *server, UA_ServerConfig *conf
    ig) {
2.      config-
    >pubsubTransportLayers = (UA_PubSubTransportLayer*)UA_calloc(2, sizeof(U
    A_PubSubTransportLayer));
3.      if (!config->pubsubTransportLayers) {
4.          UA_Server_delete(server);
5.          exit(EXIT_FAILURE);
6.      }
7.      config->pubsubTransportLayers[0] = UA_PubSubTransportLayerUDPMP();
8.      config->pubsubTransportLayersSize++;
9.  }
```

configurePubSubNetworkMessage racchiude invece le funzioni per istanziare i quattro compenti di un Publisher descritti nei paragrafi precedenti

```
1.  void configurePubSubNetworkMessage(UA_Server *server, UA_String transpor
    tProfile, UA_NetworkAddressUrlDataType networkAddressUrl, const int fiel
    dsCount, exposedNode_t fieldsToPublish[]) {
2.
3.      UA_NodeId connectionIdent, publishedDataSetIdent, writerGroupIdent;

4.
5.      addPubSubConnection(server, &transportProfile, &networkAddressUrl, &
    connectionIdent, "Connection1", 2234);
6.      addPublishedDataSet(server, &publishedDataSetIdent, "PDS1");
7.
8.      for (int i = 0; i < fieldsCount; i++) {
9.          addDataSetField(server, publishedDataSetIdent, fieldsToPublish[i
    ].nodeName, fieldsToPublish[i].nodeId);
10.     }
11.
12.     addWriterGroup(server, connectionIdent, &writerGroupIdent, "WriterGr
    oup1", 100, 1000);
13.     addDataSetWriter(server, publishedDataSetIdent, writerGroupIdent, "D
    ataSetWriter1", 62541);
14. }
```

Essa riceve il profilo di trasporto selezionato, l'url per la trasmissione, il numero dei campi da pubblicare e i nodeID che li identificano (struct fieldToPublish).

La funzione *addPubSubConnection* configura la connessione all'interno del server, l'entità che gestisce concretamente la trasmissione. Riceve in ingresso il profilo di trasporto, l'indirizzo multicast, il nome della connessione, il *PublisherID* e il puntatore della variabile in cui scrivere l'identità della connessione.

```
1.  void addPubSubConnection(UA_Server *server, UA_String *transportProfile,
     UA_NetworkAddressUrlDataType *networkAddressUrl, UA_NodeId *connectionI
    dent, char *connectionName, int publisherId) {
2.          UA_PubSubConnectionConfig connectionConfig;
3.
4.          memset(&connectionConfig, 0, sizeof(connectionConfig));
5.          connectionConfig.name = UA_STRING(connectionName);
6.          connectionConfig.transportProfileUri = *transportProfile;
7.          connectionConfig.enabled = UA_TRUE;
8.          UA_Variant_setScalar(&connectionConfig.address, networkAddressUr
    l, &UA_TYPES[UA_TYPES_NETWORKADDRESSURLDATATYPE]);
9.
10.         connectionConfig.publisherId.numeric = publisherId;
11.
12.         UA_Server_addPubSubConnection(server, &connectionConfig, connect
    ionIdent);
13. }
```

Maggiori informazioni sui profili di trasporto sono disponibili al seguente link.

PublishedDataSet e *PubSubConnection* sono le entità a livello più alto e possono esistere in maniera indipendente. *PublishedDataSet* contiene la collezione dei campi pubblicati. In esso si specifica inoltre il tipo di informazione che si vuole pubblicare, che essa sia un evento o un valore. Tutti gli altri elementi PubSub sono direttamente o indirettamente collegati al *PublishedDataSet* o alla *Connection*.

La funzione ritorna per riferimento l'identità del *PublishedDataset* sulla variabile *publishedDataSetIdent.*

```
1.  void addPublishedDataSet(UA_Server *server, UA_NodeId *publishedDataSetI
    dent, char *PDSName) {
2.
3.          UA_PublishedDataSetConfig publishedDataSetConfig;
4.          memset(&publishedDataSetConfig, 0, sizeof(UA_PublishedDataSetCon
    fig));
5.          publishedDataSetConfig.publishedDataSetType = UA_PUBSUB_DATASET_
    PUBLISHEDITEMS;
6.          publishedDataSetConfig.name = UA_STRING(PDSName);
7.          UA_Server_addPublishedDataSet(server, &publishedDataSetConfig, p
    ublishedDataSetIdent);
8.  }
```

La funzione *addDataSetField* riceve in ingresso l'identità del *PublishedDataSet*, una stringa relativa al nome del campo da pubblicare e il NodeID della variabile il cui valore è interessato alla pubblicazione.

```
1.  // Aggiunge come field il campo value della variableId in ingresso
2.  void addDataSetField(UA_Server *server, UA_NodeId publishedDataSetIdent,
     char *fieldName, UA_NodeId variableID) {
3.          /* Add a field to the previous created PublishedDataSet */
4.          UA_NodeId dataSetFieldIdent;
5.          UA_DataSetFieldConfig dataSetFieldConfig;
6.          memset(&dataSetFieldConfig, 0, sizeof(UA_DataSetFieldConfig));
7.          dataSetFieldConfig.dataSetFieldType = UA_PUBSUB_DATASETFIELD_VAR
    IABLE;
8.          dataSetFieldConfig.field.variable.fieldNameAlias = UA_STRING(fie
    ldName);
9.          dataSetFieldConfig.field.variable.promotedField = UA_FALSE;
10.         dataSetFieldConfig.field.variable.publishParameters.publishedVar
    iable = variableID;
11.         dataSetFieldConfig.field.variable.publishParameters.attributeId
    = UA_ATTRIBUTEID_VALUE;
12.         UA_Server_addDataSetField(server, publishedDataSetIdent, &dataSe
    tFieldConfig, &dataSetFieldIdent);
13. }
```

La funzione *addWriterGroup* riceve in ingresso l'identità della connessione, il puntatore alla variabile in cui scrivere l'identità del *WriterGroup*, una stringa che riporta il nome del gruppo, l'ID del *WriterGroup* e il *publishingInterval*.

```
1.  void addWriterGroup(UA_Server *server, UA_NodeId connectionIdent, UA_Nod
    eId *writerGroupIdent, char *writerGroupName, int writerGroupId, int pub
    lishingInterval) {
2.          UA_WriterGroupConfig writerGroupConfig;
3.          memset(&writerGroupConfig, 0, sizeof(UA_WriterGroupConfig));
4.          writerGroupConfig.name = UA_STRING(writerGroupName);
5.          writerGroupConfig.publishingInterval = publishingInterval;
6.          writerGroupConfig.enabled = UA_FALSE;
7.          writerGroupConfig.writerGroupId = writerGroupId;
8.          writerGroupConfig.encodingMimeType = UA_PUBSUB_ENCODING_UADP;
9.          writerGroupConfig.messageSettings.encoding = UA_EXTENSIONOBJECT_
    DECODED;
10.         writerGroupConfig.messageSettings.content.decoded.type = &UA_TYP
    ES[UA_TYPES_UADPWRITERGROUPMESSAGEDATATYPE];
11.         UA_UadpWriterGroupMessageDataType *writerGroupMessage  = UA_Uadp
    WriterGroupMessageDataType_new();
12.
13.         writerGroupMessage-
    >networkMessageContentMask = (UA_UadpNetworkMessageContentMask)(UA_UADPN
    ETWORKMESSAGECONTENTMASK_PUBLISHERID |
14.                                                             (UA_UadpNetworkM
    essageContentMask)UA_UADPNETWORKMESSAGECONTENTMASK_GROUPHEADER |
15.                                                             (UA_UadpNetworkM
    essageContentMask)UA_UADPNETWORKMESSAGECONTENTMASK_WRITERGROUPID |
16.                                                             (UA_UadpNetworkM
    essageContentMask)UA_UADPNETWORKMESSAGECONTENTMASK_PAYLOADHEADER);
17.
18.         writerGroupConfig.messageSettings.content.decoded.data = writerG
    roupMessage;
19.         UA_Server_addWriterGroup(server, connectionIdent, &writerGroupCo
    nfig, writerGroupIdent);
20.         UA_Server_setWriterGroupOperational(server, *writerGroupIdent);

21.         UA_UadpWriterGroupMessageDataType_delete(writerGroupMessage);
22. }
```

DataSetWriter funge da collante tra *WriterGroup* e *PublishedDataset*. *DataSetWriter* è collegato esattamente ad un *PublishedDataSet* e contiene le informazioni per la generazione dei *NetworkMessage*.

```
1.  void addDataSetWriter(UA_Server *server, UA_NodeId publishedDataSetIdent
    , UA_NodeId writerGroupIdent, char *dataSetWriterName, int dataSetWriter
    Id) {
2.          /*E' necessario inserire un dataSetWriter all'interno di un writ
    erGroup. Questo impone la
3.          generazione della struttura di configurazione DataSetWriterConfi
    g */
4.          UA_NodeId dataSetWriterIdent;
5.          UA_DataSetWriterConfig dataSetWriterConfig;
6.          memset(&dataSetWriterConfig, 0, sizeof(UA_DataSetWriterConfig));

7.          dataSetWriterConfig.name = UA_STRING(dataSetWriterName);
8.          dataSetWriterConfig.dataSetWriterId = dataSetWriterId;
9.  /*
10. * Se decommentato abilita il meccanismo delta/key Frame. è necessaria la
     compilazione della libreria con l'apposito flag Cmake
11. */
12.
13.         //dataSetWriterConfig.keyFrameCount = 10;
14.         UA_Server_addDataSetWriter(server, writerGroupIdent, publishedDa
    taSetIdent, &dataSetWriterConfig, &dataSetWriterIdent);
15. }
```

Dopo aver configurato tutto il necessario, si può abilitare la gestione di eventuali segnali e mandare in esecuzione il server.

```
1.  signal(SIGINT, stopHandler);
2.  signal(SIGTERM, stopHandler);
3.
4.  retval = UA_Server_run(server, &running);
5.
6.  UA_Server_delete(server);
7.  return retval == UA_STATUSCODE_GOOD ? EXIT_SUCCESS : EXIT_FAILURE;
```

3. Subscriber

Il Subscriber presentato è anch'esso un OPC UA Server che espone nel suo AddressSpace le variabili di temperatura pubblicate dal OPC UA Server Publisher. Ciò evidenzia che la relazione tipica Publisher-Server e Subscriber-Client non è obbligatoria, ma è possibile una combinazione di questi elementi.

Per il Subscriber, in accordo con lo standard e specularmente al Publisher, si definiscono le componenti: ***Connection***, ***SubscribedDataSet***, ***DataSetReader*** e ***ReaderGroup***. Le considerazioni riguardo configurazione e il ciclo di vita del server sono state già descritte nei capitoli precedenti e sono applicabili anche nel progetto Subscriber. Per la compilazione della libreria utile al Subscriber, e la sua esecuzione, si rimanda al readme presente in repository.

Implementazione

La funzione *addPubSubConnection* configura la connessione del server. Riceve in ingresso il profilo di trasporto, l'indirizzo multicast. Viene inizializzata l'identità della connessione all'interno della variabile globale *connectionIdentifier*, informazione necessaria alle componenti successive.

```
1.  UA_StatusCode addPubSubConnection(UA_Server* server, UA_String* transpor
    tProfile, UA_NetworkAddressUrlDataType* networkAddressUrl) {
2.      if ((server == NULL) || (transportProfile == NULL) || (networkAddres
    sUrl == NULL)) {
3.          return UA_STATUSCODE_BADINTERNALERROR;
4.      }
5.      UA_StatusCode retval = UA_STATUSCODE_GOOD;
6.
7.      UA_PubSubConnectionConfig connectionConfig;
8.      memset(&connectionConfig, 0, sizeof(UA_PubSubConnectionConfig));
9.      connectionConfig.name = UA_STRING("UDPMC Connection 1");
10.     connectionConfig.transportProfileUri = *transportProfile;
11.     connectionConfig.enabled = UA_TRUE;
12.     UA_Variant_setScalar(&connectionConfig.address, networkAddressUrl, &
    UA_TYPES[UA_TYPES_NETWORKADDRESSURLDATATYPE]);
13.     connectionConfig.publisherId.numeric = UA_UInt32_random();
14.     retval |= UA_Server_addPubSubConnection(server, &connectionConfig, &
    connectionIdentifier);
15.     if (retval != UA_STATUSCODE_GOOD) {
16.         return retval;
17.     }
18.     retval |= UA_PubSubConnection_regist(server, &connectionIdentifier);

19.     return retval;
20. }
```

La funzione *addReaderGroup* aggiunge il *ReaderGroup* alla connessione creata precedentemente. Viene inizializzata l'identità del *ReaderGroup* all'interno della variabile globale *readerGroupIdentifier*, anche questa necessaria alle componenti successive. Analogamente al *WriterGroup* nel Publisher, che produce il *NetworkMessage* partendo da un *DataSetMessage*, il *ReaderGroup* è utile per il procedimento inverso, ovvero ricavare il *DataSetMessage* dal *NetworkMessage*.

```
1.  UA_StatusCode addReaderGroup(UA_Server* server) {
2.      if (server == NULL) {
3.          return UA_STATUSCODE_BADINTERNALERROR;
4.      }
5.
6.      UA_StatusCode retval = UA_STATUSCODE_GOOD;
7.      UA_ReaderGroupConfig readerGroupConfig;
8.      memset(&readerGroupConfig, 0, sizeof(UA_ReaderGroupConfig));
9.      readerGroupConfig.name = UA_STRING("ReaderGroup1");
10.     retval |= UA_Server_addReaderGroup(server, connectionIdentifier, &re
    aderGroupConfig, &readerGroupIdentifier);
11.     UA_Server_setReaderGroupOperational(server, readerGroupIdentifier);

12.     return retval;
13. }
```

Viene quindi definita la funzione *addDataSetReader*; questa imposta i parametri sulla struct *readerConfig,* dichiarata globale, necessari al Subscriber al fine di comprendere quali *DataSetMessage* dovrà processare. Questa discriminazione avviene tramite la definizione di uno specifico *publisherID*, *writerGroupID*, *dataSetWriterID*. Questi identificativi corrispondono con quelli dichiarati nel Publisher. Viene inoltre eseguita la funzione delegata alla configurazione del *DataSetMetaData*. Si inizializza l'identità del *Reader* nella variabile globale *readerIdentifier.*

```
1.  UA_StatusCode addDataSetReader(UA_Server* server) {
2.      if (server == NULL) {
3.          return UA_STATUSCODE_BADINTERNALERROR;
4.      }
5.
6.      UA_StatusCode retval = UA_STATUSCODE_GOOD;
7.      memset(&readerConfig, 0, sizeof(UA_DataSetReaderConfig));
8.      readerConfig.name = UA_STRING("DataSet Reader 1");
9.
10.     UA_UInt16 publisherIdentifier = 2234;
11.     readerConfig.publisherId.type = &UA_TYPES[UA_TYPES_UINT16];
12.     readerConfig.publisherId.data = &publisherIdentifier;
13.     readerConfig.writerGroupId = 100;
14.     readerConfig.dataSetWriterId = 62541;
15.
16.     /* Setting up Meta data configuration in DataSetReader */
17.     fillTestDataSetMetaData(&readerConfig.dataSetMetaData);
```

```
    retval |= UA_Server_addDataSetReader(server, readerGroupIdentifier,
&readerConfig, &readerIdentifier);

    return retval;
}
```

A questo punto, una volta definita la connessione sulla quale ricevere e filtrare *DataSetMessage*, è necessario che il Subscriber sia in grado di comprendere la semantica delle informazioni ricevute. In accordo con lo standard, si è scelto di definire staticamente *DataSetMetaData*. Un'alternativa altrettanto valida sarebbe quella di utilizzare un *Configuration Server* per la distribuzione dei metadati in maniera dinamica.

```
void fillTestDataSetMetaData(UA_DataSetMetaDataType* pMetaData) {
    if (pMetaData == NULL) {
        return;
    }

    UA_DataSetMetaDataType_init(pMetaData);
    pMetaData->name = UA_STRING("DataSet 1");

    pMetaData->fieldsSize = WEATHER_STATIONS_COUNT;
    pMetaData->fields = (UA_FieldMetaData*)UA_Array_new(pMetaData-
>fieldsSize, &UA_TYPES[UA_TYPES_FIELDMETADATA]);

    for (int i = 0; i < WEATHER_STATIONS_COUNT; i++) {
        char tmp[120];
        sprintf(tmp, "%sTemperatureReceived", cities[i]);

        UA_FieldMetaData_init(&pMetaData->fields[i]);
        UA_NodeId_copy(&UA_TYPES[UA_TYPES_FLOAT].typeId, &pMetaData-
>fields[i].dataType);
        pMetaData->fields[i].builtInType = UA_NS0ID_FLOAT;
        pMetaData->fields[i].name = UA_STRING_ALLOC(tmp);
        pMetaData->fields[i].valueRank = -1; /* scalar */
    }
}
```

Per visualizzare le informazioni nell'*AddressSpace*, è necessario creare i nodi contenitori dei valori ricevuti, attraverso la funzione *addSubscribedVariables*. Si crea una folder all'interno di *ObjectFolder* (si ricorda che questa folder contiene le istanze degli oggetti sul server), impostando come *displayName* il parametro definito nella struct *readerConfig* (definita nella funzione *addDataSetReader*).

```
UA_StatusCode addSubscribedVariables(UA_Server* server, UA_NodeId dataSe
tReaderId) {
    if (server == NULL) {
        return UA_STATUSCODE_BADINTERNALERROR;
    }

    UA_StatusCode retval = UA_STATUSCODE_GOOD;
    UA_NodeId folderId;
```

```
    UA_String folderName = readerConfig.dataSetMetaData.name;
    UA_ObjectAttributes oAttr = UA_ObjectAttributes_default;
    UA_QualifiedName folderBrowseName;
    if (folderName.length > 0) {
        oAttr.displayName.locale = UA_STRING("en-US");
        oAttr.displayName.text = folderName;
        folderBrowseName.namespaceIndex = 1;
        folderBrowseName.name = folderName;
    }
    else {
        oAttr.displayName = UA_LOCALIZEDTEXT("en-US", "Subscribed Variables");
        folderBrowseName = UA_QUALIFIEDNAME(1, "Subscribed Variables");

    }

    UA_Server_addObjectNode(server, UA_NODEID_NULL, UA_NODEID_NUMERIC(0, UA_NS0ID_OBJECTSFOLDER),
        UA_NODEID_NUMERIC(0, UA_NS0ID_ORGANIZES), folderBrowseName, UA_NODEID_NUMERIC(0,
            UA_NS0ID_BASEOBJECTTYPE), oAttr, NULL, &folderId);

    retval |= UA_Server_DataSetReader_addTargetVariables(server, &folderId,
        dataSetReaderId,
        UA_PUBSUB_SDS_TARGET);
    UA_free(readerConfig.dataSetMetaData.fields);
    return retval;
}
```

Le funzioni finora descritte vengono chiamate nella funzione *run* che rappresenta la routine principale per la gestione dei segnali SIGING/SIGTERM, la configurazione di tutte le entità necessarie per il Subscriber e l'istanza di Server OPC UA.

```
int run(UA_String* transportProfile, UA_NetworkAddressUrlDataType* networkAddressUrl) {
    signal(SIGINT, stopHandler);
    signal(SIGTERM, stopHandler);
    UA_StatusCode retval = UA_STATUSCODE_GOOD;
    UA_Server* server = UA_Server_new();
    UA_ServerConfig* config = UA_Server_getConfig(server);
    UA_ServerConfig_setMinimal(config, 4801, NULL);

    config->pubsubTransportLayers = (UA_PubSubTransportLayer*)
        UA_calloc(2, sizeof(UA_PubSubTransportLayer));
    if (!config->pubsubTransportLayers) {
        UA_Server_delete(server);
        return EXIT_FAILURE;
    }

    config->pubsubTransportLayers[0] = UA_PubSubTransportLayerUDPMP();
    config->pubsubTransportLayersSize++;

    /* Add PubSubConnection */
    retval |= addPubSubConnection(server, transportProfile, networkAddressUrl);
    if (retval != UA_STATUSCODE_GOOD)
        return EXIT_FAILURE;

    /* Add ReaderGroup to the created PubSubConnection */
    retval |= addReaderGroup(server);
```

```
26.     if (retval != UA_STATUSCODE_GOOD)
27.         return EXIT_FAILURE;
28.
29.     /* Add DataSetReader to the created ReaderGroup */
30.     retval |= addDataSetReader(server);
31.     if (retval != UA_STATUSCODE_GOOD)
32.         return EXIT_FAILURE;
33.
34.     /* Add SubscribedVariables to the created DataSetReader */
35.     retval |= addSubscribedVariables(server, readerIdentifier);
36.     if (retval != UA_STATUSCODE_GOOD)
37.         return EXIT_FAILURE;
38.
39.     retval = UA_Server_run(server, &running);
40.     UA_Server_delete(server);
41.     return retval == UA_STATUSCODE_GOOD ? EXIT_SUCCESS : EXIT_FAILURE;
42. }
```

4. Conclusioni

Il presente progetto è stato realizzato con lo scopo di approfondire principalmente il meccanismo PubSub descritto dallo standard e sviluppare una sua implementazione utilizzando lo stack in linguaggio C fornito dalla libreria Open62541. Attualmente lo stato dell'arte della libreria, riguardo le funzionalità analizzate, permette il suo utilizzo su tutte le piattaforme, a meno del Subscriber UDP UADP. Questo infatti è, al momento della scrittura di questa relazione, ancora incompleto, non essendo state rilasciate tutte le interfacce (sorgenti pubsub.c e pubsub.h) necessarie al suo corretto funzionamento. Per questa ragione, la compilazione del subscriber è possibile esclusivamente utilizzando le interfacce private della libreria, pratica non consigliabile in production. Nonostante ciò è evidente la potenzialità della libreria, avendo constatato diverse possibili configurazioni alternative al protocollo UDP UADP utilizzato. Sarebbe possibile infatti utilizzare profili di trasporto alternativi come *Eth UADP*, principalmente indicato in contesti Real-Time (attualmente disponibile solo in ambiente linux based).

Nonostante l'applicazione didattica dell'indagine effettuata, ritenuta parecchio interessante, viene considerata un buon punto di partenza per applicazioni su scenari più complessi come quelli reali.