UNIVERSITA' DEGLI STUDI DI CATANIA

Corso di Laurea Magistrale in Ingegneria Informatica

TUTORIAL PER L'IMPLEMENTAZIONE DI UN CLIENT OPC UA IN JAVA CON FOCUS SUL TESTING DEI SERVIZI DI BROWSING, LETTURA DELLE VARIABILI, SUBSCRIPTION E MONITORED ITEM

Professore:
Salvatore Cavalieri

A cura di:
Santo Finocchiaro
Alessandro Vecchio

Sommario

1.Introduzione

1.1 Overview

Questo tutorial vuole mostrare come realizzare ed usare un client OPC UA in java che impieghi i più importanti servizi dello standard, con lo scopo di implementare determinati obbiettivi, quali:
1)***Browsing*** per l'esplorazione dell'Address Space di un server organizzato secondo un Information Model (in questo caso di uno specifico server .NET ma può essere utilizzato anche per altri tipi di server).
2)***Lettura delle Variabili*** esposte dal server.
3)***Subscription*** e ***Monitored Item*** sulle variabili esposte dal server.

1.2 Strumenti Utilizzati

Gli strumenti utilizzati e necessari sono:

- OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]
- OPC UA SDK 1.01 Quickstarts Setup [320.0 Stable]
- BrowsingSubscription.java
- Eclipse
- JDK 7 o +
- Documentazione ufficiale dello Standard OPC UA

1.3 Installazione

Di seguito sono fornite le informazioni necessarie all'installazione degli strumenti necessari:

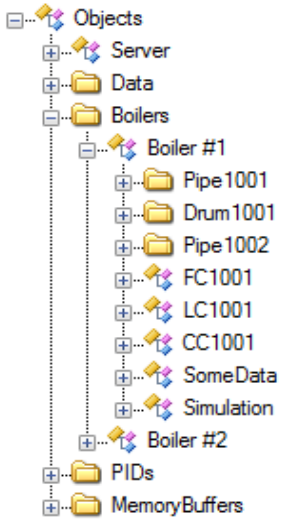
1. installare il Java SDK 7 o superiore;
2. installare l'ambiente di sviluppo Eclipse;
3. copiare il contenuto del pacchetto OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable] nel workspace di eclipse;
4. Copiare la cartella html su workspace\OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]. All'interno della cartella sono presenti i file css e le varie librerie per la visualizzazione dell'albero a menù che rappresenterà dinamicamente browsing e variabili.
5. Copiare all'interno della cartella /example/org/opcfoundation/ua/examples/ dello stack il file `BrowsingSubscription.java` associato a questo tutorial;
6. seguire le istruzioni nel file install.txt contenuto nel pacchetto OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable] per importare lo stack nell'ambiente di sviluppo eclipse;

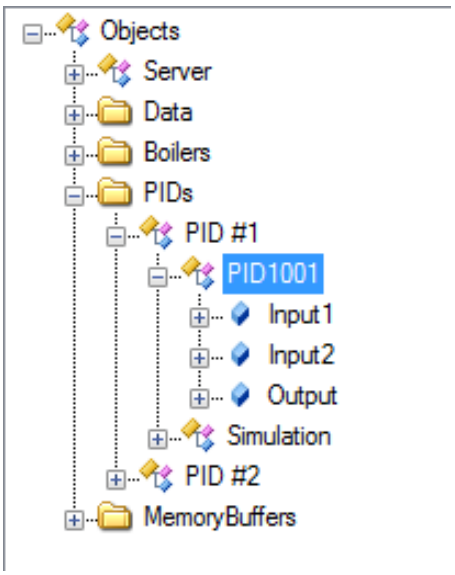
7. In Eclipse, nel Project Explorer sulla sinistra, aprire il package examples e quindi il file `BrowsingSubscription.java` associato a questo tutorial.
8. Installare OPC UA SDK 1.01 Quickstarts Setup [320.0 Stable], andare sulla cartella di installazione, selezionare la cartella "bin" e lanciare Quickstarts.BoilerServer.
9. Eseguire BrowsingSubscription

1.4 Server OPC UA .NET

L'obiettivo principale è la realizzazione di un Client che usando le API OPC UA scritte in JAVA possa connettersi ad un server che invece usa lo stack protocollare scritto in .NET.

Effettuando una breve analisi della parte server si nota che il server installato fa parte del pacchetto Quickstart, in particolare verrà usato Quickstarts.BoilerServer. L'information model del server contiene le informazioni su due Bollitori che prendono il nome di Boiler #1 e Boiler #2. Di seguito li vediamo rappresentati:





Tra le variabili mostrate nell'Information Model verranno scelti i MonitoredItems. Dopo vari test si è scoperto che solo una variabile per boiler varierà e quindi potenzialmente genererà notifiche sulla variazione dei dati ai client sottoscritti. Le variazioni sono in entrambi i casi di una unità al secondo, nel boiler 1 il range è tra 0 e 99, mentre nel boiler 2 tra 0 e 19.

1.5 Client OPC UA Java

Il Client è stato implementato usando le API OPC UA scritte in JAVA, nello specifico istanziando la già presente classe Client.java. Nella parte iniziale del main avviene l'inizializzazione ed avvio del client che effettua la ricerca del server e degli endpoints da selezionare con dati già noti, quali l'URL (relativi al server OPC boiler) e il protocollo di comunicazione (TCP) senza nessun controllo di sicurezza.
È possibile implementare parametrizzazioni come quelli della scelta dei vari URLs di uno o più server, di tutto quello che riguarda la scelta degli endpoints e la sicurezza (fare riferimento alla seconda parte del progetto)

2.Connessione al Server

In questo progetto non sono stati implementati per intero tutti gli aspetti della sicurezza integrati nel processo di connessione al server, dato che il focus è il browsing dell'Address Space di uno specifico server.

Si è deciso quindi di effettuare una connessione diretta con il Server avendo a disposizione già l'URL non selezionando alcuna politica di sicurezza, il minimo indispensabile per poter implementare il focus. In breve verranno spiegati i vari step effettuati per la creazione ed attivazione della sessione, ponendo il luce i vari servizi base forniti dallo standard. Per ulteriori approfondimenti, fare riferimento allo standard [OPC UA part 2] ed alla seconda parte del progetto.

2.1 OPC UA Get Endpoints Service

Il ***GetEndpoints Service*** fa parte del ***Discovery Service Set***. Il client contatterà con una request insieme all'URL del server il DiscoveryEndpoint. Il servizio restituisce un array contenente tutti gli Endpoints supportati dal server stesso. La descrizione di ogni Endpoint contiene tutte le informazioni necessarie per creare un Secure Channel ed una Sessione tra client e server.

GetEndpoints Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
endpointUrl	String	The network address that the *Client* used to access the *Discovery Endpoint*. The *Server* uses this information for diagnostics and to determine what URLs to return in the response. The *Server* should return a suitable default URL if it does not recognize the *HostName* in the URL.
localeIds []	LocaleId	List of locales to use. Specifies the locale to use when returning human readable strings. This parameter is described in 5.4.2.2.
profileUris []	String	List of transport profiles that the returned *Endpoints* shall support. All *Endpoints* are returned if the list is empty.
Response		
responseHeader	ResponseHeader	Common response parameters. The *ResponseHeader* type is defined in 7.27.
Endpoints []	EndpointDescription	List of *Endpoints* that meet criteria specified in the request. This list is empty if no *Endpoints* meet the criteria. The *EndpointDescription* type is defined in 7.9.

All'interno del vettore Endpoints contenuto nella Response saranno presenti l'elenco degli endpoint disponibili con le varie modalità di sicurezza supportate da parte del server.
La request viene effettuata dal client e la response catturata in un vettore di Endpoints.
Il codice OPC UA Java è dato dal seguente comando.

```
EndpointDescription[] endpoints = myClient.discoverEndpoints(url);
```

La request viene effettuata dall'oggetto Client tramite la funzione `discoverEndpoints`(url), inserendo l'url relativo al Discovery Service del server.

Sul vettore ottenuto è possibile effettuare operazioni quali ad esempio, ordinamento per livello di sicurezza degli endpoints e funzioni di filtraggio in base al protocollo da selezionare.

Verrà quindi ordinato il vettore in ordine crescente e saranno selezionati tutti gli endpoints con protocollo TCP con il comando `endpoints = selectByProtocol(endpoints, "opc.tcp").`

Verrà selezionato l'endpoint alla posizione zero del vettore, che conterrà l'endpoint esposto dal server che non implementa nessun livello di sicurezza.

2.2 Open Secure Channel Service

Il service ***Open SecureChannel*** appartiene al ***SecureChannel Service Set***, nel quale è presente anche il servizio di chiusura del canale. Dopo aver selezionato l'endpoint messo a disposizione dal server, viene inviata al server una request relativa all'apertura di un canale sicuro sull'endpoint di nostra scelta. Prima che il canale possa essere aperto il server deve verificare la validità dell'Application Instance Certificate del client, ma che non avviene in questo caso dato che è stato selezionato un endpoint senza implementazione di sicurezza.

OpenSecureChannel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
clientCertificate	ApplicationInstance Certificate	A *Certificate* that identifies the *Client*. The *OpenSecureChannel* request shall be signed with this *Certificate*. The *ApplicationInstanceCertificate* type is defined in 7.2.
requestType	enum SecurityToken RequestType	The type of *SecurityToken* request: An enumeration that shall be one of the following: ISSUE_0 creates a new *SecurityToken* for a new *SecureChannel*. RENEW_1 creates a new *SecurityToken* for an existing *SecureChannel*.
secureChannelId	ByteString	The identifier for the *SecureChannel* that the new token should belong to. This parameter shall be null when creating a new *SecureChannel*.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type *MessageSecurityMode* type is defined in 7.14. A *SecureChannel* may have to be created even if the *securityMode* is NONE. The exact behaviour depends on the mapping used and is described in the Part 6.
securityPolicyUri	String	The URI for *SecurityPolicy* to use when securing messages sent over the *SecureChannel*. The set of known URIs and the *SecurityPolicies* associated with them are defined in Part 7.
clientNonce	ByteString	A random number that shall not be used in any other request. A new *clientNonce* shall be generated for each time a *SecureChannel* is renewed. This parameter shall have a length equal to key size used for the symmetric encryption algorithm that is identified by the *securityPolicyUri*.
requestedLifetime	Duration	The requested lifetime, in milliseconds, for the new *SecurityToken*. It specifies when the *Client* expects to renew the *SecureChannel* by calling the *OpenSecureChannel Service* again. If a *SecureChannel* is not renewed, then all *Messages* sent using the current *SecurityTokens* shall be rejected by the receiver.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type definition).
securityToken	ChannelSecurityToken	Describes the new *SecurityToken* issued by the *Server*.
channelId	ByteString	A unique identifier for the *SecureChannel*. This is the identifier that shall be supplied whenever the *SecureChannel* is renewed.
tokenId	ByteString	A unique identifier for a single *SecurityToken* within the channel. This is the identifier that shall be passed with each *Message* secured with the *SecurityToken*.
createdAt	UtcTime	When the *SecurityToken* was created.
revisedLifetime	Duration	The lifetime of the *SecurityToken* in milliseconds. The UTC expiration time for the token may be calculated by adding the lifetime to the *createdAt* time.
serverNonce	ByteString	A random number that shall not be used in any other request. A new *serverNonce* shall be generated for each time a *SecureChannel* is renewed. This parameter shall have a length equal to key size used for the symmetric encryption algorithm that is identified by the *securityPolicyUri*.

All'interno delle API di OPC UA Java, l'oggetto client ha i metodi relativi alla creazione del canale sicuro. I parametri sono l'URL del server e l'endpoint selezionato.
Ciò che verrà ritornato è un canale sicuro da cui poter estrarre tutti parametri relativi alla response.

```
mySecureChannel = myClient.createSecureChannel(url, selectedEndpoint);
```

Anche se è stato selezionato un endpoint senza nessun livello di sicurezza, è obbligatorio creare un canale sicuro.

2.3 OPC UA CreateSession Service

Servizio appartenente al SessionService set, permette di creare una sessione.

Nelle API OPC UA java il servizio è implementato tramite il seguente comando

```
mySession = myClient.createSession(mySecureChannel);
```

Il client effettua la request tramite il metodo createSession, inserendo il canale sicuro creato precedentemente. La response ritornerà un oggetto Session il quale conterrà tutti parametri descritti dallo standard. Nelle tabelle successive è possibile notare i parametri relativi a request e response.

2.4 API OPC UA Java createSessionChannel

All'interno delle API dell'OPC UA Java esistono diversi comandi e parametrizzazioni che riguardano la creazione di canali sicure e sessioni. Le API permettono la creazione in un canale sicuro e di una sessione in solo comando, tramite:

```
mySessionChannel = myClient.createSessionChannel(url,endpoint);
```

In questo modo verrà creato ed attivato il canale sicuro più la creazione di una sessione non ancora attivata. È possibile richiamare dall'oggetto mySessionChannel, i vari servizi relativi sia alla sessione che al canale. L'operazione di chiusura di canale e sessione è infatti effettuata sempre sull'oggetto mySessionChannel.

```
mySessionChannel.closeAsync();
```

Il comando sopracitato chiuderà la sessione ed il canale in sequenza in maniera asincrona.

CreateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
clientDescription	Application Description	Information that describes the *Client* application. The type *ApplicationDescription* is defined in 7.1.
serverUri	String	This value is only specified if the *EndpointDescription* has a *gatewayServerUri*. This value is the *applicationUri* from the *EndpointDescription* which is the *applicationUri* for the underlying *Server*. The type *EndpointDescription* is defined in 7.9.
endpointUrl	String	The network address that the *Client* used to access the *Session Endpoint*. The *HostName* portion of the URL should be one of the *HostNames* for the application that are specified in the *Server's ApplicationInstanceCertificate* (see Cause 7.2). The *Server* shall raise an *AuditUrlMismatchEventType* event if the URL does not match the Server's *HostNames*. *AuditUrlMismatchEventType* event type is defined in Part 5. The Server uses this information for diagnostics and to determine the set of *EndpointDescriptions* to return in the response.
sessionName	String	Human readable string that identifies the *Session*. The *Server* makes this name and the *sessionId* visible in its *AddressSpace* for diagnostic purposes. The *Client* should provide a name that is unique for the instance of the *Client*. If this parameter is not specified the *Server* shall assign a value.
clientNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. Profiles may increase the required length. The *Server* shall use this value to prove possession of its application instance *Certificate* in the response.
clientCertificate	ApplicationInstance Certificate	The application instance *Certificate* issued to the *Client*. The *ApplicationInstanceCertificate* type is defined in 7.2.
Requested SessionTimeout	Duration	Requested maximum number of milliseconds that a *Session* should remain open without activity. If the *Client* fails to issue a *Service* request within this interval, then the *Server* shall automatically terminate the *Client Session*.
maxResponse MessageSize	UInt32	The maximum size, in bytes, for the body of any response message. The *Server* should return a *Bad_ResponseTooLarge* service fault if a response message exceeds this limit. The value zero indicates that this parameter is not used. 5.3 provides more information on the use of this parameter.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type).
sessionId	NodeId	A unique *NodeId* assigned by the *Server* to the *Session*. This identifier is used to access the diagnostics information for the *Session* in the *Server* address space. It is also used in the audit logs and any events that report information related to the *Session*. The *Session* diagnostic information is described in Part 5. Audit logs and their related events are described in 6.2
authentication Token	Session AuthenticationToken	A unique identifier assigned by the *Server* to the *Session*. This identifier shall be passed in the *RequestHeader* of each request and is used with the *SecureChannelId* to determine whether a *Client* has access to the *Session*. This identifier shall not be reused in a way that the *Client* or the *Server* has a chance of confusing them with a previous or existing *Session*. The *SessionAuthenticationToken* type is described in 7.29.
revisedSession Timeout	Duration	Actual maximum number of milliseconds that a *Session* shall remain open without activity. The *Server* should attempt to honour the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The *Client* shall use this value to prove possession of its application instance *Certificate* in the *ActivateSession* request. This value may also be used to prove possession of the *userIdentityToken* it specified in the *ActivateSession* request.
serverCertificate	ApplicationInstance Certificate	The application instance *Certificate* issued to the *Server*. A *Server* shall prove possession by using the private key to sign the *Nonce* provided by the *Client* in the request. The *Client* shall verify that this *Certificate* is the same as the one it used to create the *SecureChannel*. The *ApplicationInstanceCertificate* type is defined in 7.2.
serverEndpoints []	Endpoint Description	List of *Endpoints* that the server supports. The *Server* shall return a set of *EndpointDescriptions* available for the *serverUri* specified in the request. The *EndpointDescription* type is defined in 7.9. The *Client* shall verify this list with the list from a *Discovery Endpoint* if it used a *Discovery Endpoint* to fetch the *EndpointDescriptions*.

serverSoftware Certificates []	SignedSoftware Certificate	These are the *SoftwareCertificates* which have been issued to the *Server* application. The *productUri* contained in the *SoftwareCertificates* shall match the *productUri* in the *EndpointDescription* used by the *Client* to connect to the *Server*. *Certificates* without matching *productUris* should be ignored. *Clients* should call *CloseSession* if they are not satisfied with the *SoftwareCertificates* provided by the *Server*. The *SignedSoftwareCertificate* type is defined in 7.31.
serverSignature	SignatureData	This is a signature generated with the private key associated with the *serverCertificate*. This parameter is calculated by appending the *clientNonce* to the *clientCertificate* and signing the resulting sequence of bytes. The *SignatureAlgorithm* shall be the *asymmetricSignature* algorithm specified in the *SecurityPolicy* for the *Endpoint*. The *SignatureData* type is defined in 7.30.
maxRequest MessageSize	UInt32	The maximum size, in bytes, for the body of any request message. The *Client* stack should return a *Bad_RequestTooLarge* error to the application if a request message exceeds this limit. The value zero indicates that this parameter is not used. 5.3 provides more information on the use of this parameter.

2.5 OPC UA ActivateSession

Questo è il secondo dei due servizi che si occupano dell'handshaking per la creazione della sessione tra client e server, appartenente anch'esso al ***Session Service set***. Viene utilizzato anche per cambiare l'utente della sessione o per assegnare un nuovo Secure Channel alla sessione.
Qui la sessione creata al passo precedente viene attivata. Si può accedere alla sessione tramite accesso anonimo, username e password o un Issued Identity Token.

Il comando relativo all'attivazione è relativo al sessionChannel ma valido anche per una normale sessione. Da notare che all'interno del metodo non vengono passati parametri riguardanti l'autenticazione dell'utente, parte non implementata in questo progetto.

```
mySessionChannel.activate();
```

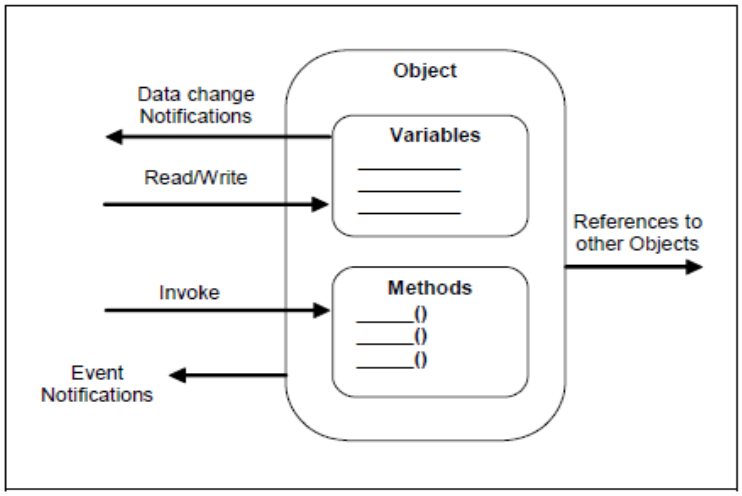
ActivateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The type *RequestHeader* is defined in 7.26.
clientSignature	SignatureData	This is a signature generated with the private key associated with the *clientCertificate*. The *SignatureAlgorithm* shall be the *asymmetricSignature* algorithm specified in the *SecurityPolicy* for the *Endpoint*. The *SignatureData* type is defined in 7.30.
clientSoftwareCertificates []	SignedSoftware Certificate	These are the *SoftwareCertificates* which have been issued to the *Client* application. The *productUri* contained in the *SoftwareCertificates* shall match the *productUri* in the Application*Description* passed by the *Client* in the *CreateSession* requests. *Certificates* without matching *productUris* should be ignored. *Servers* may reject connections from *Clients* if they are not satisfied with the *SoftwareCertificates* provided by the *Client*. This parameter only needs to be specified during the first call to *ActivateSession* for a single application *Session*. The *SignedSoftwareCertificate* type is defined in 7.31.
localeIds []	LocaleId	List of locale ids in priority order for localized strings. The first *localeId* in the list has the highest priority. If the *Server* returns a localized string to the *Client*, the *Server* shall return the translation with the highest priority that it can. If it does not have a translation for any of the locales identified in this list, then it shall return the string value that it has and include the locale id with the string. See Part 3 for more detail on locale ids. If the *Client* fails to specify at least one locale id, the *Server* shall use any that it has. This parameter only needs to be specified during the first call to *ActivateSession* during a single application *Session*. If it is not specified the *Server* shall keep using the current *localeIds* for the *Session*.
userIdentityToken	Extensible Parameter UserIdentityToken	The credentials of the user associated with the *Client* application. The *Server* uses these credentials to determine whether the *Client* should be allowed to activate a *Session* and what resources the *Client* has access to during this *Session*. The *UserIdentityToken* is an extensible parameter type defined in 7.35. The EndpointDescription specifies what *UserIdentityTokens* the Server shall accept.
userTokenSignature	SignatureData	If the *Client* specified a user identity token that supports digital signatures, then it shall create a signature and pass it as this parameter. Otherwise the parameter is omitted. The *SignatureAlgorithm* depends on the identity token type. The *SignatureData* type is defined in 7.30.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The *Client* shall use this value to prove possession of its application instance *Certificate* in the next call to *ActivateSession* request.
results []	StatusCode	List of validation results for the *SoftwareCertificates* (see 7.33 for *StatusCode* definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information associated with *SoftwareCertificate* validation errors (see 7.8 for *DiagnosticInfo* definition). This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

3. Address Space

3.1 Object Model

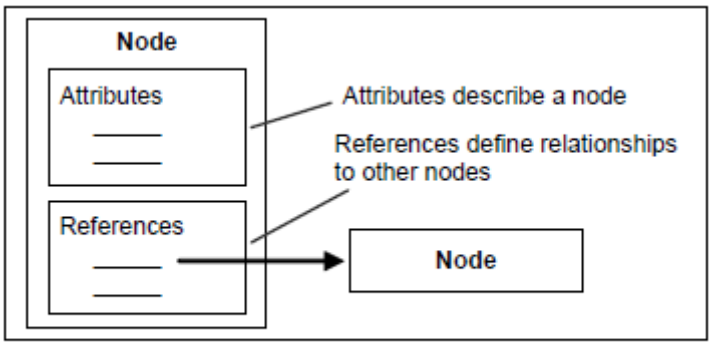
L'obiettivo primario di un OPC UA Address Space è quello di fornire uno standard ai server per rappresentare gli *Objects* ai client. Esso definisce Objects in termini di Variables e Methods e permette la rappresentazioni di relazioni tra oggetti.



OPC UA Object Model

Gli elementi dell'Object Model sono rappresentati nell'Address Space come nodi (*Nodes*). Ogni Node è assegnato ad un *NodeClass* ed ogni NodeClass rappresenta un elemento differente dell'Object Model.

Il set di oggetti e le informazioni relative che l'OPC UA server rende disponibili ai client, fa riferimento al proprio Address Space. Gli oggetti ed i loro componenti sono rappresentati nell'Address Space come un insieme di nodi descritti da attributi ed interconnessi da References.



AddressSpace Node Model

I NodeClass sono definiti in termini di Attributes e References che devono essere istanziati quando un nodo è definito nell'Address Space. L'insieme di tutti i NodeClasses sono considerati come i metadata dell'Address Space.

Ogni Nodo nell'AddressSpace è un'istanza di uno di questi NodeClass. Nessun altro NodeClass deve essere usato per definire i nodi, ed inoltre, client e server, non possono definire nuovi NodeClass o estendere le loro definizioni. Per ulteriori dettagli consultare lo standard (OPC UA part 3).

3.2 Objects, Variables, Methods

I NodeClass principali su cui si fonda l'Address Space sono Objects,Variables, Methods. Il concetto è quello dell'object-oriented programming. Gli oggetti hanno variabili e metodi e possono generare eventi.

I nodi di tipo NodeClass Variable rappresentano un valore. Il tipo di dato dipende dalla variabile. I client possono leggere e scrivere il valore, sottoscriversi ai cambiamenti del valore. Una variabile è usata per esempio per rappresentare una temperatura misurata da un sensore o un setpoint di temperature per la gestione di qualche controllo di applicazione, ma in generale per esporre qualunque dato nell'Address Space.

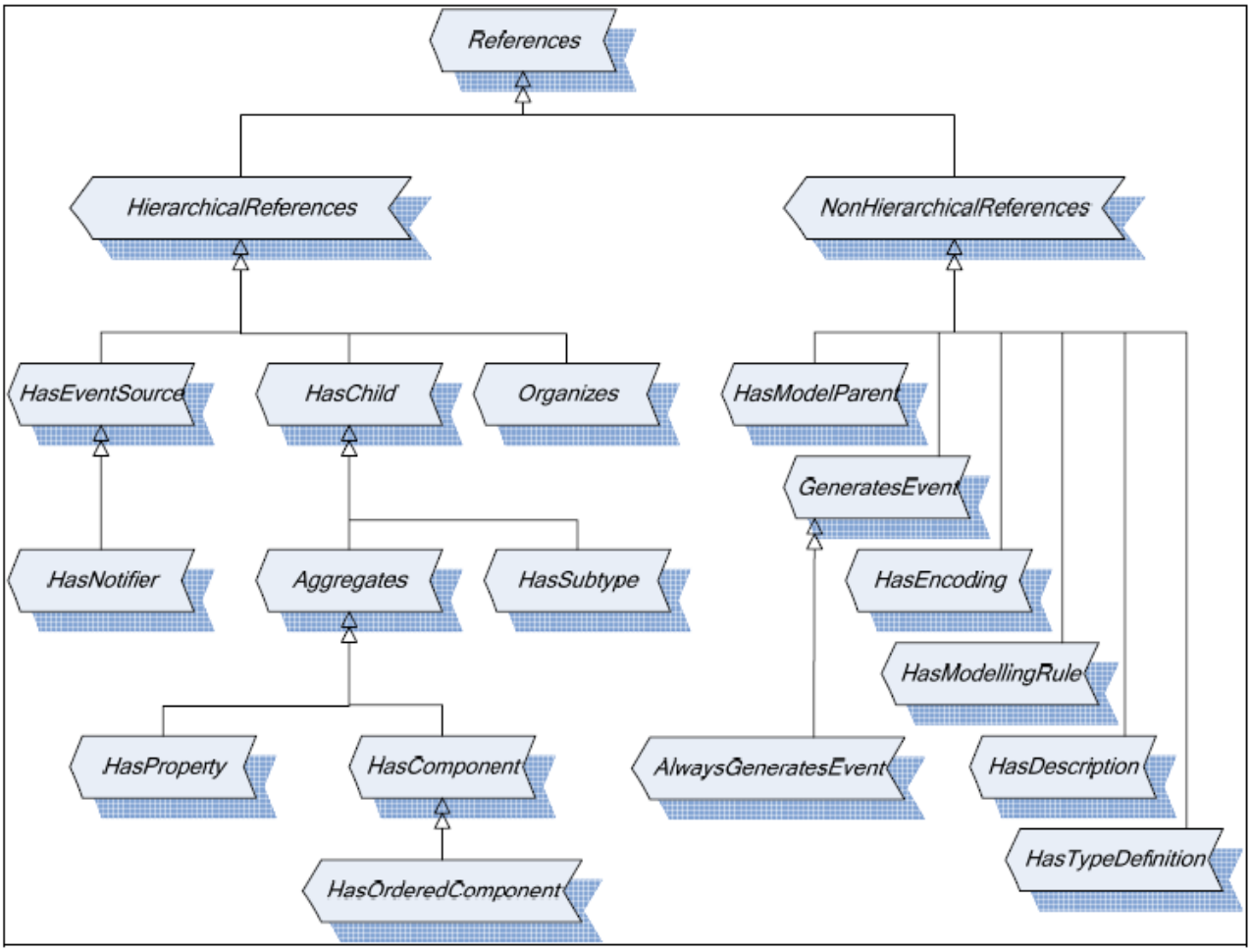
I nodi di tipo NodeClasse Method rappresentano un metodo, ovvero qualcosa chiamato dal client e ritorna un risultato.

Nodi appartenenti a NodeClass Object vengono usati per strutturare l'Address Space. Gli Objects non contengono Attributi che espongono valori, come nel caso dei Variables, ma possono essere usati per raggruppare Variables, Methods ed altri oggetti.

Methods e Variables appartengono sempre ad un Oggetto.

3.3 Reference

Una reference è una connessione tra due Nodes. Si può accedere ad una Reference solo indirettamente tramite il browsing di un nodo perché non è rappresentata come un nodo, e non può contenere altri Attributes o Properties. Le references sono usate per esporre semantiche differenti di come i nodi sono connessi all'interno dell'AddressSpace. Per definire la semantica delle References, OPC UA usa i *ReferenceTypes*, organizzati in maniera gerarchica. In OPC UA part 3 sono definite dettagliatamente le semantiche di ogni ReferenceType e del significato all'interno dell'albero gerarchico.



Standard ReferenceType Hierarchy

4.Information Model

4.1 Intro

L'Information Model (IM) rappresenta il fondamento per l'information modeling in OPC UA e per definire Information Model addizionali. Il Base Information Model è estendibile ed è usato come standard per definire capacità, informazioni di diagnostica di un OPC UA server nel suo Address Space e come base per modellare specifiche informazioni riguardanti per dati attuali, dati storici, stato delle macchine, programmi, allarmi e condizioni, etc

Dal punto di vista dell'OPC UA Address Space, l'Information Model principalmente definisce dei Nodes, parte dei quali hanno NodeIds "well-know". Tipi differenti di nodi possono essere definiti in base al contesto. Tipicamente, un Information Model definisce TypeDefinition, EventTypes, ReferenceTypes e DataTypes. Un Information Model può definire standard Properties e Methods definendo uno specifico BrowseName e una semantica per ognuno. L'insieme di tutte queste definizioni, fornisce una "semantica" alle informazioni presenti sul server.

L'Information Model può definire standard Objects e Views utilizzabili come standard "entry point" nell'Address Space, e Variable standard contenenti dati ben definiti.

Oltre a ciò, l'IM può definire vincoli che non sono visibili nell'Address Space. Per esempio, esso definisce regole che restringono per esempio l'uso di ReferenceTypes, o che specificano che ogni device di un determinato sistema deve essere rappresentato nell'OPC UA server da uno specifico ObjectType.

4.2 Information Model multipli

I server possono supportare più di un Information Model allo stesso momento. OPC UA provvede a dei meccanismi molto semplici per poter permettere ciò. Un IM definisce Nodes univoci nell'Address Space, Properties standard e Metodi. L'unicità dei Nodi è fornita dal NodeId, l'unicità delle proprietà e dei metodi dal BrowseName.

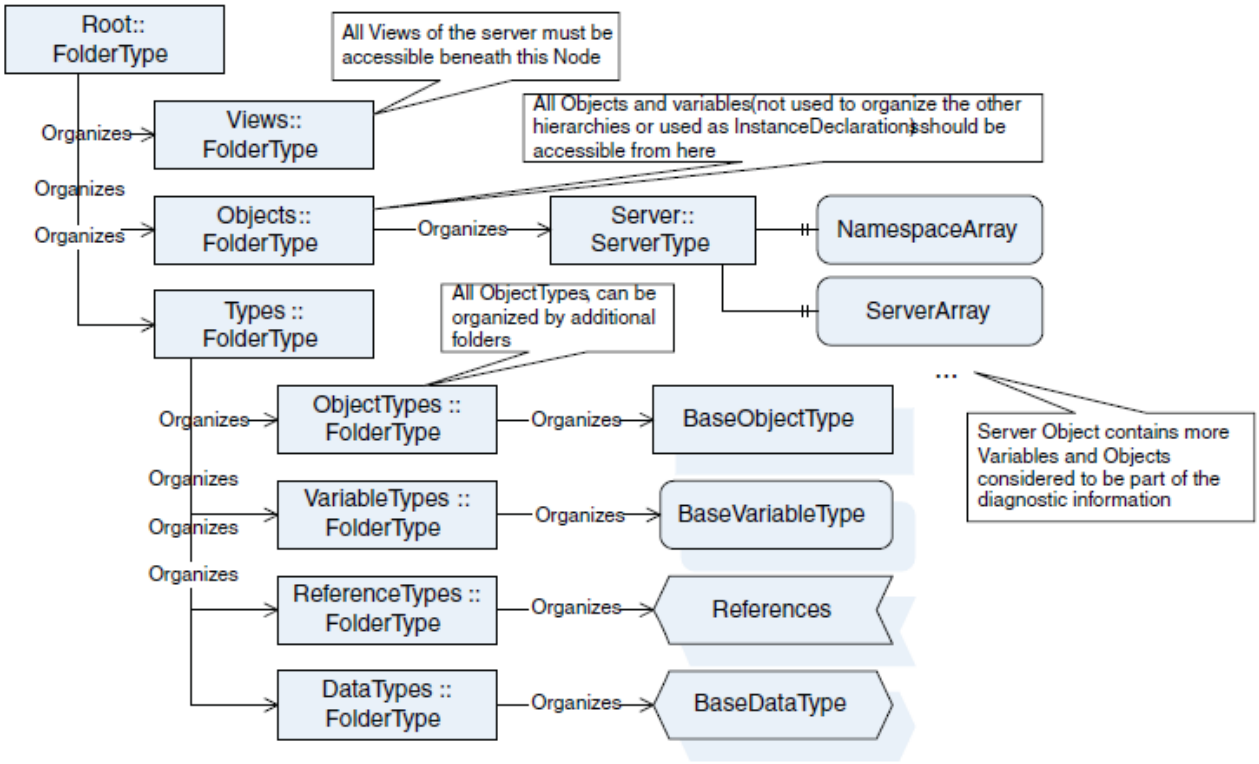
Per evitare il rischio che due Information Model usino lo stesso NodeId o lo stesso BrowseName, entrambe contengono un NameSpaceURI (ottimizzato dal NameSpaceIndex [OPC UA part 3]). Ogni organizzazione usa il proprio NameSpaceURI, il quale per definizione stessa di URI, è univoco. In questo modo, NodeIds e BrowseNAmes diventano unici per l'organizzazione che li definisce. E questo permette ai server di esporre più IMs senza conflitti tra i nomi.

4.3 Base OPC UA Information Model

Il Base IM è il modello base che espone lo standard OPC UA. È definito sia in OPC UA part3, che in part5, con definizioni ulteriori di Nodes che definiscono capacità e servizi di diagnostica del server. Tutti i NodeIds definiti dall'OPC Foundation usano il Namespace URI dell'OPC UA e NamespaceIndex zero.

Nel Base IM, viene descritta l'organizzazione gerarchica standard della definizione dei tipi (TypeDefinitions), delle references (ReferenceTypes), dei tipi di dato (DataTypes), dei tipi di Event, etc. Si consiglia di consultare lo standard per un loro approfondimento.

Il base Information Model definisce inoltre alcuni “nodi standard”, come entry point nell’Address Space. L’organizzazione di questi nodi oggetto e delle variabili definisce le capacità e le proprietà del server.



Objects and Variables of the base Information Model

Per quanto riguarda questo lavoro, ci si concentrerà sull’accesso dell’Address Space di un server DOTnet, escludendo la parte dell’IM relativa alla definizione dei tipi, dei dati etc. Lo scopo del progetto è quello di esplorare il server da uno dei nodi well-know forniti dal Base IM, quali uno fra tutti l’ObjectsFolder, nodo figlio di RootFolder, dove sono esposti i nodi contenenti dati e variabili, che nel server DOTnet in questione rappresentano dei Boiler.

5. Funzione di Browsing

5.1 Intro

La funzione di browsing in questo progetto consiste nell'esplorazione dei nodi posti all'interno dell'Address Space [UA part 3] visualizzandone il contenuto ed il tipo di reference con cui sono legati, facendo perno sui servizi definiti nello standard OPC UA [UA part 4].
L'esplorazione non è di tipo "assoluto", cioè partire dalla radice dell'Infomation Model ed esplorare l'intero Address Space, ma "relativo", ossia navigare nell'Address Space a partire da un nodo radice scelto dall'utente ed esplorare fino ad una certa profondità.
L'output è di due tipi:

1. su terminale, con la stampa a video di tutte le caratteristiche dei nodi esplorati
2. di tipo grafico, con la creazione ad ogni esplorazione di un file html che visualizzerà tramite un menù ad albero i vari nodi legati dalle references, esponendo così in maniera più fruibile le informazioni in esame.

Il file html verrà creato in automatico ogni volta che viene effettuato un browsing o una lettura delle variabili, e verrà posto nella cartella
workspace\OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]\html\BrowsingOutput.html.

5.2 OPC UA Browse Service

Il "core" dell'algoritmo è basato su un'iterazione di una funzione (service) implementata nello standard OPC UA utilizzata per trovare informazioni nell'Address Space.
La funzione è ***Browse Service*** [UA part 4], appartenente al ***View Service Set***. La funzione è usata dal client per navigare attraverso l'Address Space del server, inviando un *Browse Request* e passando in input un nodo (Node) o più nodi di partenza. In output il client riceverà il relativo *Browse Response*, contenente un vettore con al suo interno informazioni sui nodi collegati al nodo posto in input. Il Browse Service acquisisce in input una lista di Node iniziali e ritorna un lista di Node connessi per ogni nodo di partenza tramite relative References. Nella tabella seguente sono indicati i parametri di input e output del server.

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>Request</td><td></td><td></td></tr>
<tr><td>requestHeader</td><td>RequestHeader</td><td>Common request parameters (see 7.26 for RequestHeader definition).</td></tr>
<tr><td>view</td><td>ViewDescription</td><td>Description of the View to browse (see 7.37 for ViewDescription definition). An empty ViewDescription value indicates the entire AddressSpace. Use of the empty ViewDescription value causes all References of the nodeToBrowse to be returned. Use of any other View causes only the References of the nodeToBrowse that are defined for that View to be returned.</td></tr>
<tr><td>requestedMax ReferencesPerNode</td><td>Counter</td><td>Indicates the maximum number of references to return for each starting Node specified in the request. The value 0 indicates that the Client is imposing no limitation (see 7.5 for Counter definition).</td></tr>
<tr><td>nodesToBrowse []</td><td>BrowseDescription</td><td>A list of nodes to Browse</td></tr>
<tr><td>nodeId</td><td>NodeId</td><td>NodeId of the Node to be browsed. The passed nodeToBrowse shall be part of the passed view.</td></tr>
<tr><td>browseDirection</td><td>enum BrowseDirection</td><td>An enumeration that specifies the direction of References to follow. It has the following values :
FORWARD_0 select only forward References.
INVERSE_1 select only inverse References.
BOTH_2 select forward and inverse References.
The returned References do indicate the direction the Server followed in the isForward parameter of the ReferenceDescription.
Symmetric References are always considered to be in forward direction therefore the isForward flag is always set to TRUE and symmetric References are not returned if browseDirection is set to INVERSE_1.</td></tr>
<tr><td>referenceTypeId</td><td>NodeId</td><td>Specifies the NodeId of the ReferenceType to follow. Only instances of this ReferenceType or its subtypes are returned.
If not specified then all References are returned.</td></tr>
<tr><td>includeSubtypes</td><td>Boolean</td><td>Indicates whether subtypes of the ReferenceType should be included in the browse. If TRUE, then instances of referenceTypeId and all of its subtypes are returned.</td></tr>
<tr><td>nodeClassMask</td><td>UInt32</td><td>Specifies the NodeClasses of the TargetNodes. Only TargetNodes with the selected NodeClasses are returned. The NodeClasses are assigned the following bits:
<table><tr><th>Bit</th><th>NodeClass</th></tr><tr><td>0</td><td>Object</td></tr><tr><td>1</td><td>Variable</td></tr><tr><td>2</td><td>Method</td></tr><tr><td>3</td><td>ObjectType</td></tr><tr><td>4</td><td>VariableType</td></tr><tr><td>5</td><td>ReferenceType</td></tr><tr><td>6</td><td>DataType</td></tr><tr><td>7</td><td>View</td></tr></table>If set to zero, then all NodeClasses are returned.</td></tr>
<tr><td>resultMask</td><td>UInt32</td><td>Specifies the fields in the ReferenceDescription structure that should be returned. The fields are assigned the following bits:
<table><tr><th>Bit</th><th>Result</th></tr><tr><td>0</td><td>ReferenceType</td></tr><tr><td>1</td><td>IsForward</td></tr><tr><td>2</td><td>NodeClass</td></tr><tr><td>3</td><td>BrowseName</td></tr><tr><td>4</td><td>DisplayName</td></tr><tr><td>5</td><td>TypeDefinition</td></tr></table>The ReferenceDescription type is defined in 7.24.</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Response</td><td></td><td></td></tr>
<tr><td>responseHeader</td><td>Response Header</td><td>Common response parameters (see 7.27 for ResponseHeader definition).</td></tr>
<tr><td>results []</td><td>BrowseResult</td><td>A list of BrowseResults. The size and order of the list matches the size and order of the nodesToBrowse specified in the request.
The BrowseResult type is defined in 7.3.</td></tr>
<tr><td>diagnosticInfos []</td><td>Diagnostic Info</td><td>List of diagnostic information for the results (see 7.8 for DiagnosticInfo definition). The size and order of the list matches the size and order of the results response parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.</td></tr>
</table>

Ponendo ad esempio come ingresso in *NodeToBrowse[]* un solo nodo da cui iniziare l'esplorazione, il server risponderà con una BrowseResponse contenente all'interno il vettore *result[]*, i cui elementi rappresentano le informazioni relative ad ogni nodo collegato al nodo di input tramite una determinata reference (il cui tipo può essere inserito come parametro per il filtraggio dell'output con nodi legati solamente con quel tipo di reference).

È possibile inserire in input anche una lista di nodi. In questo progetto, il servizio viene usato passando un solo Node di partenza con lo scopo ultimo di creare un albero gerarchico di nodi iterando più volte il servizio.
Il relativo codice della BrowseService in OPC UA java è implementato nel modo seguente.

```
private static BrowseResponsebrowseOPC_UA(NodeIdcurrentRoot, NodeIdReferenceTypeId)
throwsServiceFaultException, ServiceResultException{
// BrowseRoot
     BrowseDescriptionbrowse = newBrowseDescription();
     browse.setNodeId(currentRoot);
     browse.setBrowseDirection(BrowseDirection.Forward);
     browse.setIncludeSubtypes( true );
     if (ReferenceTypeId!=null)
        browse.setReferenceTypeId(ReferenceTypeId);
     browse.setNodeClassMask( NodeClass.Object,NodeClass.Variable,NodeClass.View);
     browse.setResultMask( BrowseResultMask.All );
     BrowseResponse res = mySession.Browse( null, null, null, browse );
     return res;
}
```

Il service è implementato dalla API `mySession.Browse( null, null, null, browse );`
Il parametro `browse` è un oggetto che parametrizza il servizio con

- Il nodo radice da cui iniziare l'esplorazione
- La direzione della Reference (forward, inverse o both)
- Se includere nella ricerca sottotipi della reference
- Il tipo di reference cui sono legati i nodi all'interno dell'Address Space (passata come parametro scelto dall'utente). Nel progetto si è optati nell'indicare la reference con null, nel caso in cui non si voglia effettuare nessun filtraggio (tutte le reference verranno visualizzate di default se non viene settato il parametro)
- Filtrare i nodi per NodeClass di appartenenza
- Il tipo di informazioni da riportare in output

`BrowseResponse res` rappresenta la risposta del server. Richiamando su di esso la funzione *getResults()* si ottiene il vettore *result[]* i cui elementi conterranno informazioni sul nodo i-esimo collegato al nodo di input. Ad esempio i nodo figli sono interamente ottenibili richiamando *res.getResults()[i].getNode()*.

5.3 Browsing Input

Il programma procede inizialmente acquisendo i dati di input dell'utente quali il nodo radice da cui iniziare il browsing e la profondità dell'esplorazione (ovvero nella creazione dell'albero gerarchico fino a quale livello si vuole esplorare), ed il tipo di reference con cui si vuole filtrare.

```
public static void browsing() throws NumberFormatException, ServiceFaultException,
ServiceResultException, IOException {
      //BROWSING
String r = getDaTastiera("Vuoi Fare il Browsing  (y/n) ?");
      if (r.equalsIgnoreCase("y"))
      {
            NodeIdreferenceSelected;
      NodeIdrootOfBrowsing;
```

```
            //selezione della root cui partire il browsing

            do{
                  rootOfBrowsing = selectRoot();

                  //selezione delle reference cui filtrare il browsing
                  referenceSelected=selectReference();

                  //selezione della profondità del browsing
                  profonditàTot=selectDepth();
                  browsing(rootOfBrowsing,referenceSelected);

                  r = getDaTastiera("Vuoi Continuare a Fare il Browsing  (y/n)?");
            } while (r.equalsIgnoreCase("y"));
      }
}
```

Il nodo radice può essere di tipo "well-know", ovvero nodi base relativi al ***Base OPC UA Information Model***, come ad esempio ***Root Folder***, ***Object Folder,*** etc, oppure da un nodo inserito da tastiera.

```
//Selezione della radice da cui partire il browsing
public static NodeIdselectRoot(){
      intans;
      do{
            ans=Integer.valueOf(getDaTastiera("SELEZIONA LA RADICE DA CUI INIZIARE
L'ESPLORAZIONE\n"+
                                                "\n 1)RootFolder\n"+
                                                "\n 2)ObjectsFolder\n"+
                                                "\n 3)ViewsFolder\n"+
                                                "\n 4)TypesFolder\n"+
                                                "\n 5)ServerType_Namespaces\n"+
                                                "\n 6)Server_Namespaces\n"+
                                                "\n 7)Server_NamespaceArray\n"+
                                                "\n 8)NodeId from keyboard\n"));

            switch(ans){

            case 1: return(Identifiers.RootFolder);
            case 2: return(Identifiers.ObjectsFolder);
            case 3: return(Identifiers.ViewsFolder);
            case 4: return(Identifiers.TypesFolder);
            case 5: return(Identifiers.ServerType_Namespaces);
            case 6: return(Identifiers.Server_Namespaces);
            case 7: return(Identifiers.Server_NamespaceArray);
            case 8: return (newNodeId());
      //da migliorare
            default: System.out.println("Wrongselection");
            }
      }while (ans<1||ans>8);
      //non dovresti essere qui
      return null;
}
```

Se si inizia un'esplorazione di un server sconosciuto, basta porsi nel Root Folder per capire come è organizzato l'information model e proseguire man mano con le esplorazioni ripartendo dai nodi figli ottenuti.

5.4 Iterazione della Funzione di Browsing

Una volta ottenuti i dati di input da terminale viene innanzitutto creato un file html tramite la funzione "`private static void writeHtmlHead(NodeIdRoot)`" la quale inizializzerà il file con le opportune intestazioni html ed il relativo codice Jquery, e stampando su file il RootNode che è stato selezionato in precedenza.
Una volta inizializzato il file html, vengono richiamata una delle due funzioni

- `public static void writeHtmlNoLoopReference(NodeIdcurrentRoot, int profondità, NodeIdreferenceSelected)`
- `public static void writeHtml(NodeIdcurrentRoot,int profondità, NodeIdreferenceSelected)`

Entrambe implementano l'algoritmo ***Depth First*** per l'esplorazione dell'Address Space e svolgono lo stesso compito, ma la prima viene usata quando in input si hanno Reference Type che garantiscono l'assenza di loop tra i nodi (le reference tra i Node possono creare loop). Queste reference sono tutte quelle gerarchicamente al di sotto della ***HasChild Reference*** [UA Part 3].
Con la prima funzione si evita quindi l'uso di funzioni che controllano la presenza di loop senza le quali l'algoritmo ricadrebbe in cicli infiniti, evitando calcoli onerosi.
La seconda funzione implementa l'uso di funzioni di controllo dei loop, tramite l'ausilio di strutture dati (java maps). La presenza dei loop verrà segnalata sul file html.
Entrambe le funzioni richiamano ricorsivamente la funzione di browsing, creando man mano un albero gerarchico basandosi sull'algoritmo Depth First.
Nella funzione di browsing, il client invia una BrowseRequest parametrizzando e ponendo in ingresso un nodo, ed il server invia un BrowseResponse che contiene al suo interno la lista di nodi cui punta il nodo inserito in input tramite la relativa reference scelta e gli altri parametri.
La lista di nodi ottenuta viene iterata ed ogni nodo viene posto in input alla funzione di browsing creando quindi la ricorsione. Il criterio di stop della ricorsione sarà la profondità o l'assenza di altri nodi figlio.

5.5 Output

L'output, sia quello su terminale che quello su file html è incastonato all'interno dell'algoritmo Depth First. Durante l'algoritmo vengono stampati nelle relative posizioni i tag html per la creazione di un codice corretto per il browser, che rappresenti effettivamente l'Address Space analizzato, rappresentato graficamente tramite un menù ad albero espandibile. Una volta effettuato il browsing il file è pronto per essere visualizzato. Il file html viene sovrascritto ad ogni browsing.

Partendo dal nodo iniziale RootFolder è possibile comprendere l'organizzazione dell'Address Space.

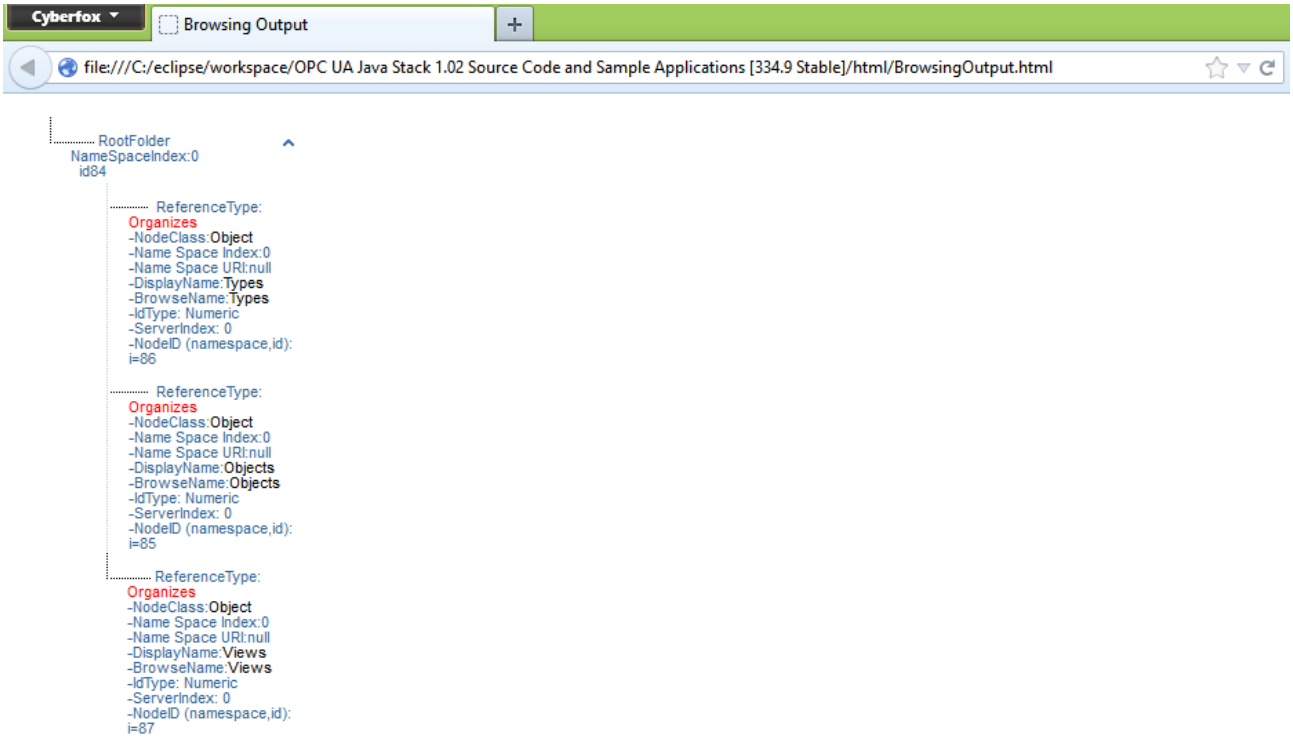
Si hanno

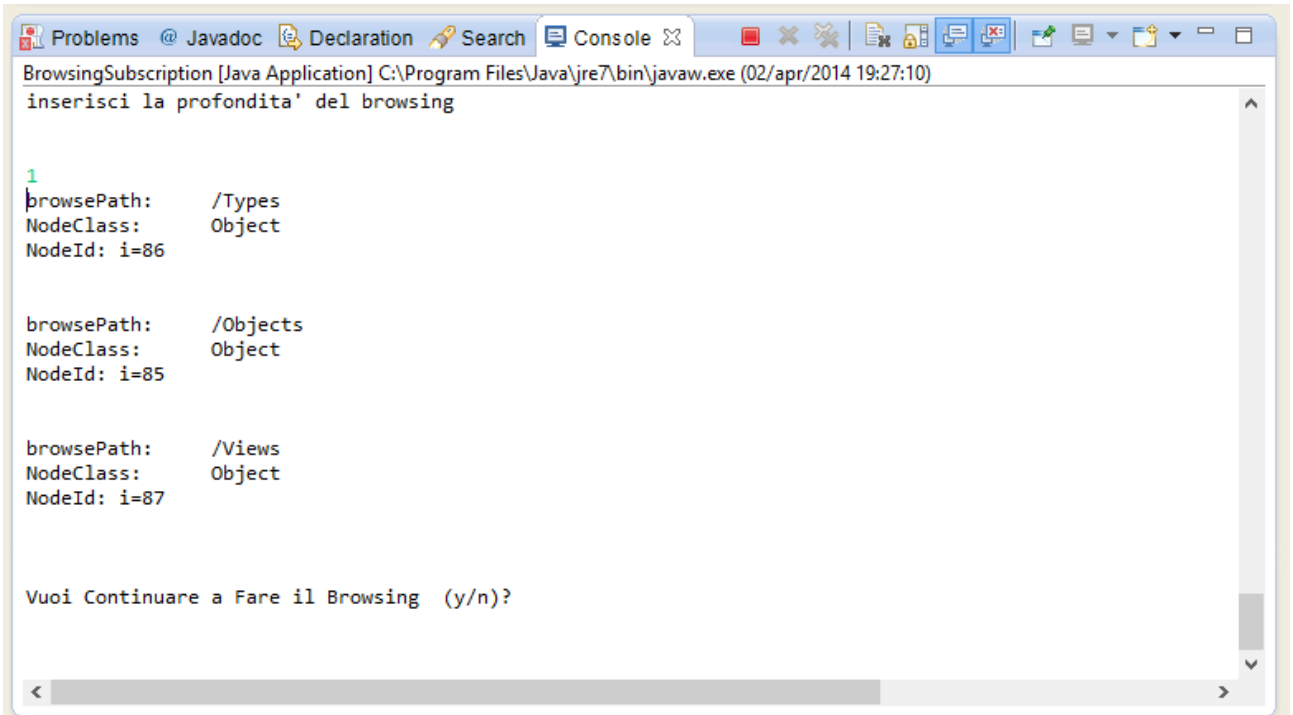
- il nodo TypeFolder, contenente l'organizziazione dei tipi all'interno dell'Infomation Model.

- il nodo ViewFolder, contenente eventuali viste dell'address space [OPC UA part].

- il nodo ObjectsFolder contenente le infomazioni principali esposte dal server.

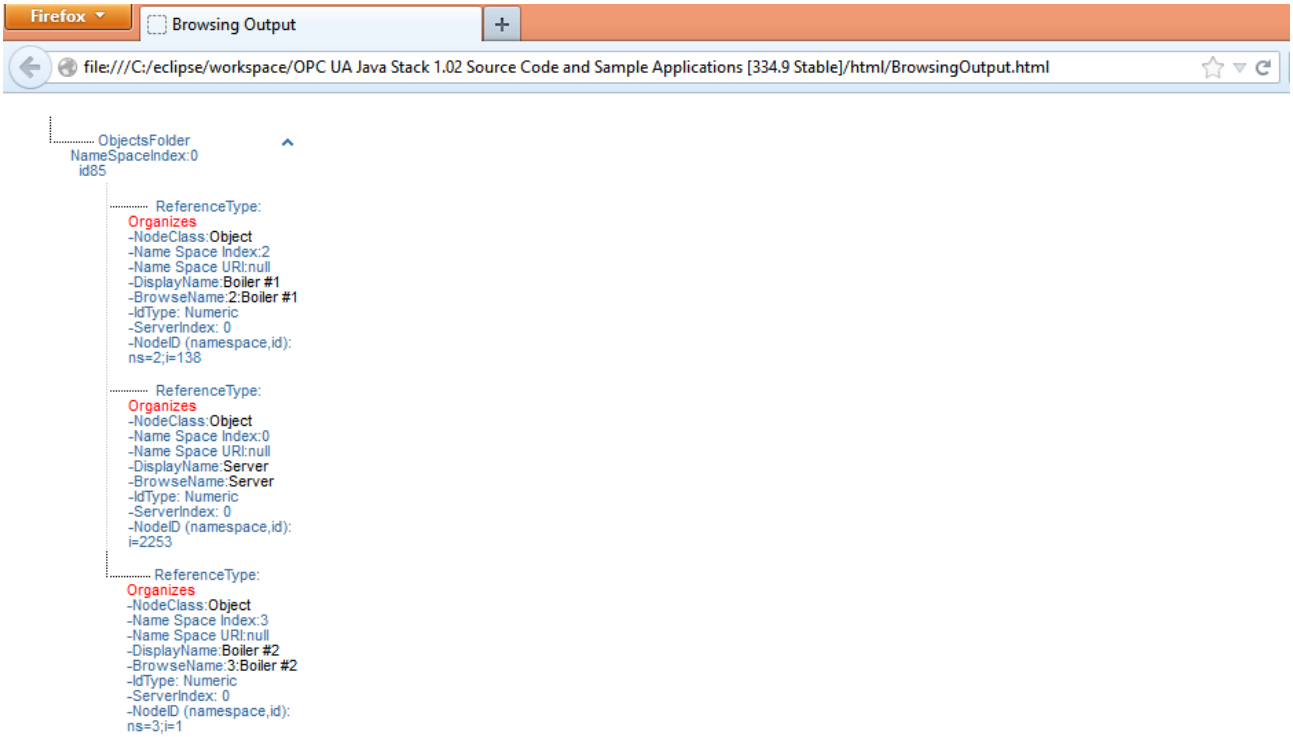
Qui sono stati postati l'output html e l'output su terminale.

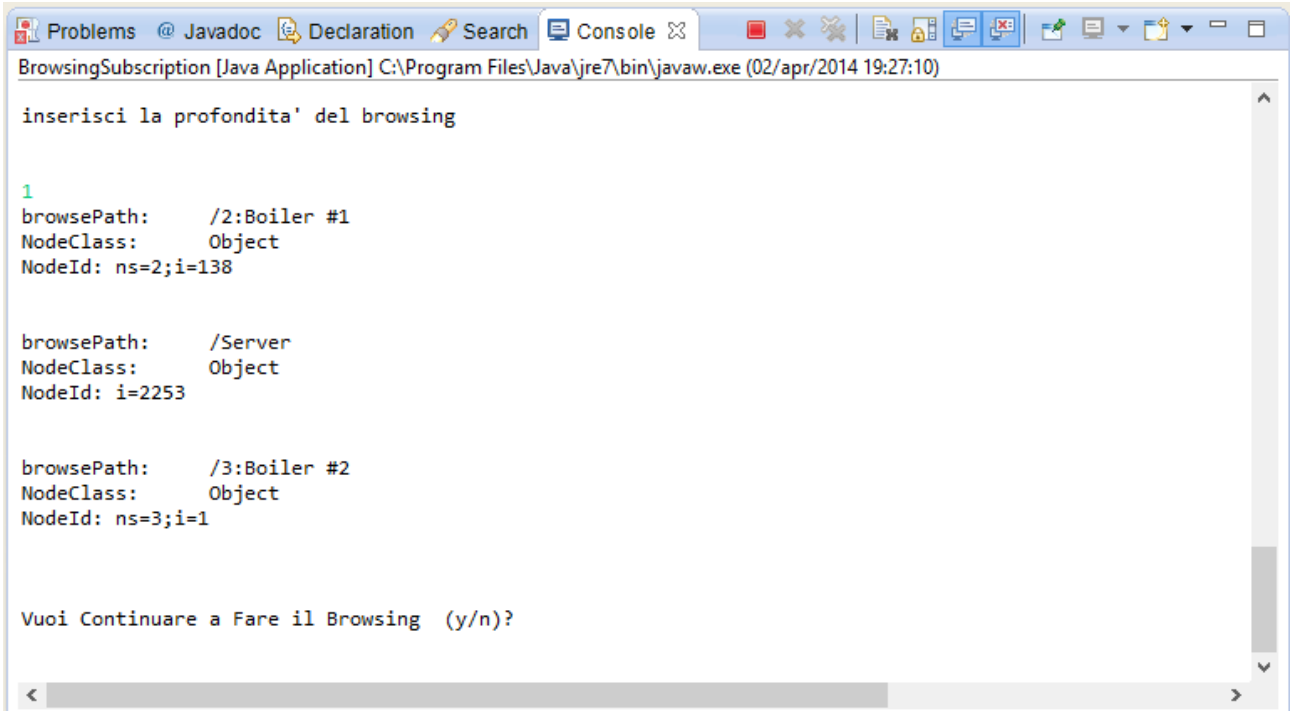




Esplorando l'Address Space dall'ObjectsFolder si notano i due boiler esposti dal server, in cui al proprio interno saranno presenti eventuali variabili.

All'interno ci sono le due variabili che cambiano il loro valore nel tempo, a cui in seguito saranno effettuate le Subscriptions.





5.6 Conclusioni sul Browsing

Il processo globale di browsing può essere riassunto in 3 passi:
Step1: Il client inizia con il richiamare il BrowseService passando in input il nodo radice selezionato.
Step2: La funzione itera i nodi contenuti nel BrowseResult del server response per stamparli a video e salvarli su file html
Step3: Ricorsione, passando come root node il nodo sotto processo al ciclo for, facendo così partire un nuovo ramo dell'albero.

6. Lettura delle Variabili

6.1 Intro

La funzione di lettura di nodi di tipo Variable in questo esempio viene suddivisa in due modalità:

- lettura diretta di una variabile il cui namespaceindex e nodeId sono già noti.
- esplorazione dell'Address Space partendo da un determinato nodo radice per ricercare eventuali variabili esposte dal Server, visualizzandone il contenuto, il tipo di reference con cui sono legati ed il loro valore, facendo perno sui servizi definiti nello standard OPC UA [UA part 4]. In quest'ultima modalità, allo stesso modo delle funzione di Browsing, l'esplorazione non è di tipo "assoluto", cioè partire dalla radice dell'Infomation Model ed esplorare l'intero Address Space, ma "relativo", ossia navigare nell'Address Space a partire da un nodo radice scelto dall'utente ed esplorare fino ad una certa profondità.

In entrambe le modalità, la lettura degli attributi del nodo variabile ha come core la funzione Read Service.

6.2 OPC UA ReadService

La lettura degli attributi di un nodo variabile (come di qualunque altro nodo) è possibile tramite la funzione (Service) ***ReadService.*** Il service fa parte dell'***Attribute Service Set*** [UA part 4]. Il servizio è usato per leggere uno più attributi di uno o più Nodi.
Il client invia un *ReadService Request* al server, ponendo nella request oltre ad alcuni parametri, un nodo o più nodi da cui si vuole ottenere la lista di attributi.
Se viene posto un solo nodo, il response del server sarà un vettore *result[ ]* di dimensione 1 (ovvero *result[0]*) il cui elemento conterrà la lista di attributi richiamabili.
Se viene posta una lista di nodi, il response sarà il vettore *result[ ]* con dimensione pari alla lista di ingresso, ed ogni posizione *result[i]* farà riferimento al nodo i-esimo d'ingresso.
Per attributi compositi i cui valori sono indicizzati, per esempio posti in un array [OPC UA part 3], il service permette al client di leggere l'intero set di valori indicizzati o leggere elementi individuali e leggere un range di elementi. Fare riferimento ad [OPC UA part 4] per la parametrizzazione di ReadService. Qui viene riportata una tabella riassuntiva.

Read Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
maxAge	Duration	Maximum age of the value to be read in milliseconds. The age of the value is based on the difference between the *ServerTimestamp* and the time when the *Server* starts processing the request. For example if the *Client* specifies a *maxAge* of 500 milliseconds and it takes 100 milliseconds until the *Server* starts processing the request, the age of the returned value could be 600 milliseconds prior to the time it was requested. If the *Server* has one or more values of an *Attribute* that are within the maximum age, it can return any one of the values or it can read a new value from the data source. The number of values of an *Attribute* that a *Server* has depends on the number of *MonitoredItems* that are defined for the *Attribute*. In any case, the *Client* can make no assumption about which copy of the data will be returned. If the *Server* does not have a value that is within the maximum age, it shall attempt to read a new value from the data source. If the *Server* cannot meet the requested *maxAge*, it returns its "best effort" value rather than rejecting the request. This may occur when the time it takes the *Server* to process and return the new data value after it has been accessed is greater than the specified maximum age. If *maxAge* is set to 0, the *Server* shall attempt to read a new value from the data source. If *maxAge* is set to the max Int32 value or greater, the *Server* shall attempt to get a cached value. Negative values are invalid for *maxAge*.
timestampsTo Return	enum TimestampsTo Return	An enumeration that specifies the *Timestamps* to be returned for each requested *Variable Value Attribute*. The *TimestampsToReturn* enumeration is defined in 7.34.
nodesToRead []	ReadValueId	List of *Nodes* and their *Attributes* to read. For each entry in this list, a *StatusCode* is returned, and if it indicates success, the *Attribute Value* is also returned. The ReadValueId parameter type is defined in 7.23.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	DataValue	List of *Attribute* values (see 7.7 for *DataValue* definition). The size and order of this list matches the size and order of the *nodesToRead* request parameter. There is one entry in this list for each *Node* contained in the *nodesToRead* parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information (see 7.8 for *DiagnosticInfo* definition). The size and order of this list matches the size and order of the *nodesToRead* request parameter. There is one entry in this list for each *Node* contained in the *nodesToRead* parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

L'implementazione in OPC UA java del service è data dalla seguente funzione.

```
// Read a variable with namespaceindex ns, nodeId i
ReadResponse res = mySession.Read(
null,
500.0,
TimestampsToReturn.Source,
new ReadValueId(newNodeId(), Attributes.Value, null, null )
);
System.out.println(res.getResults()[0].getValue());
```

Il response dato dal server viene catturato e contiene le informazioni relative agli attributi del nodo di input. Essendo stato posto solo un nodo in ingresso effettuato con "`new`ReadValueId(*newNodeId*(), ….)", il risultato è composto da un array con un solo elemento (`res.getResults()[0]`) contenente gli attributi del nodo.
`res.getResults()` rappresenta il vettore result[ ] sopra descritto.
Richiamando `res.getResults()[i].getValue()` si otterrà l'attributo valore (*Value Attribute*) da leggere dall'i-esimo nodo (in questo caso lo 0-esimo). È possibile leggere altri tipi di attributi, basta modificare il parametro Attributes in ReadValueId.
Per esplorare nuove parametrizzazioni, osservare all'interno delle API le dichiarazioni di *mySession*.Read e ReadValueId.

6.2 Implementazione Lettura Variabili

La lettura delle variabili si divide come già citato nell'Intro in:

- lettura di una variabile il cui namespaceindex e nodeId sono già noti
- lettura di un insieme di variabili presenti nell'Address Space partendo da un determinato nodo radice

Entrambe le letture hanno come core la ReadService.

La lettura di una singola variabile con namespaceidex e nodeId già noti è effettuata semplicemente richiamando in maniera diretta la ReadService.
In questa modalità di lettura, il contenuto del nodo variabile sarà solo su terminale.

La funzione di lettura di più variabili all'interno dell'Address Space, ha come core l'algoritmo Depth First, richiamando le stesse funzioni di browsing (*writehtml* e *writeHtmlNoLoop*) già viste prima al cui interno viene posta la funzione di ReadService.

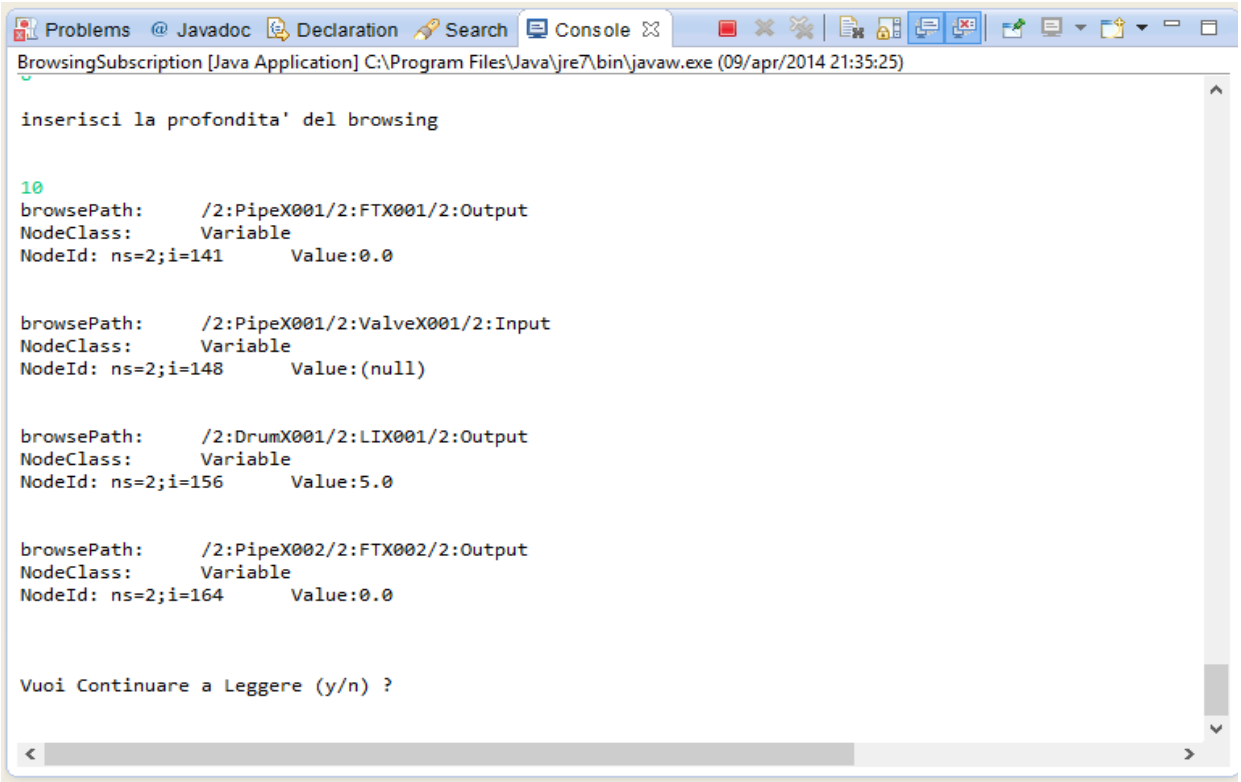
Per far iniziare l'esplorazione, si dovrà inserire un nodo radice da cui far partire il browsing e il tipo di reference.
Durante l'esplorazione, non appena viene trovata una variabile esposta, viene stampata a video.
Ciò viene effettuato tramite un filtro, in modo tale che durante l'esplorazione vengano scartati per la stampa tutti gli altri oggetti e filtrare a video solo le variabili. Nell'output vengono stampati il contenuto del nodo variabile ed il relativo browsepath.

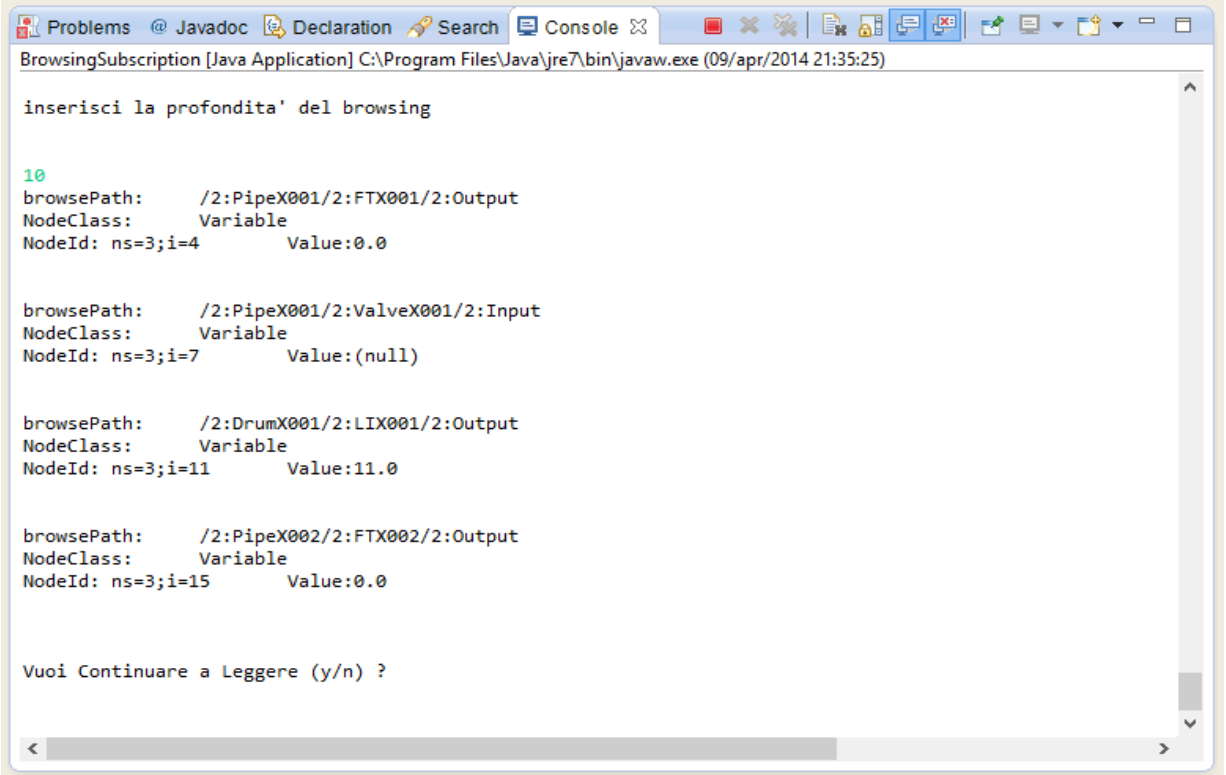
Man mano che l'esplorazione dell'albero relativo avanza, vengono salvate le informazioni dei vari nodi (sia non variabile che variabile) su file html che visualizzerà tramite un menù ad albero i vari nodi legati dalle references, esponendo così in maniera più fruibile le informazioni in esame. Le foglie dell'albero rappresenteranno le eventuali variabili presenti nella parte di Address Space sotto esame.
Il file html verrà creato in automatico ogni volta che viene effettuato una lettura delle variabili, e verrà posto nella cartella
workspace\OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]\html\BrowsingOutput.html.

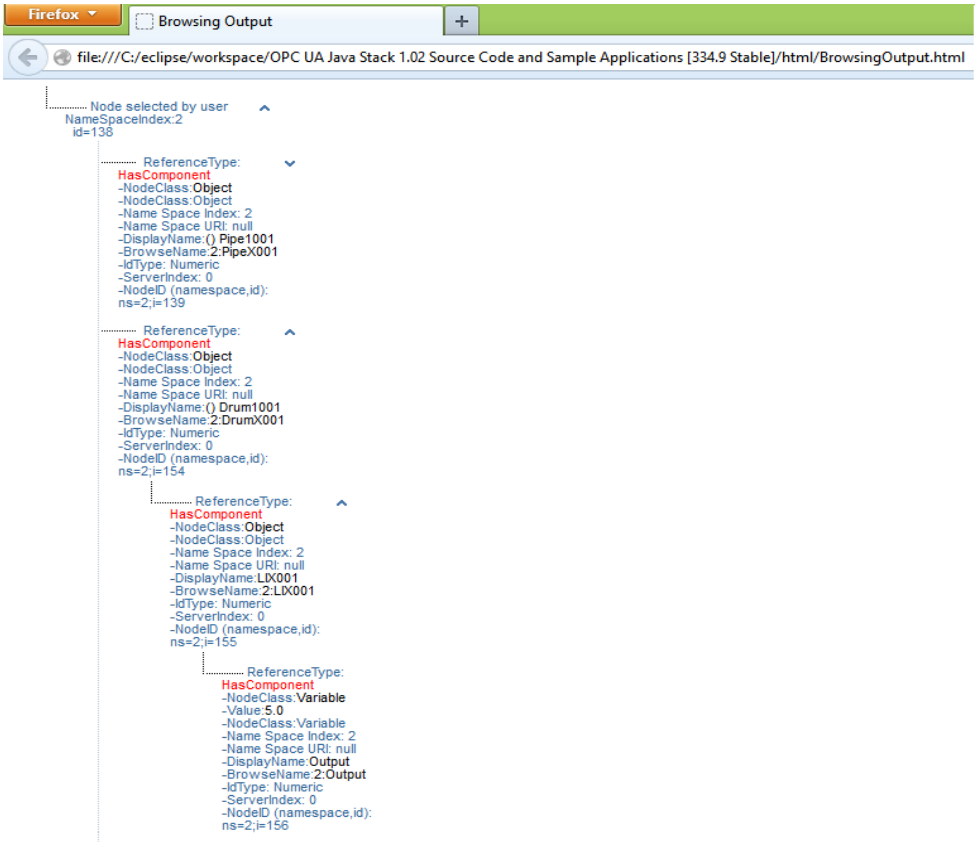
Importante: le uniche due variabili esposte che cambiano valore sono il nodo con namespaceindex 2, id 156 (2,156) pre quanto riguarda il boiler1, e namespaceindex =3, id = 11 (3,11) per quanto riguarda il boiler 2.

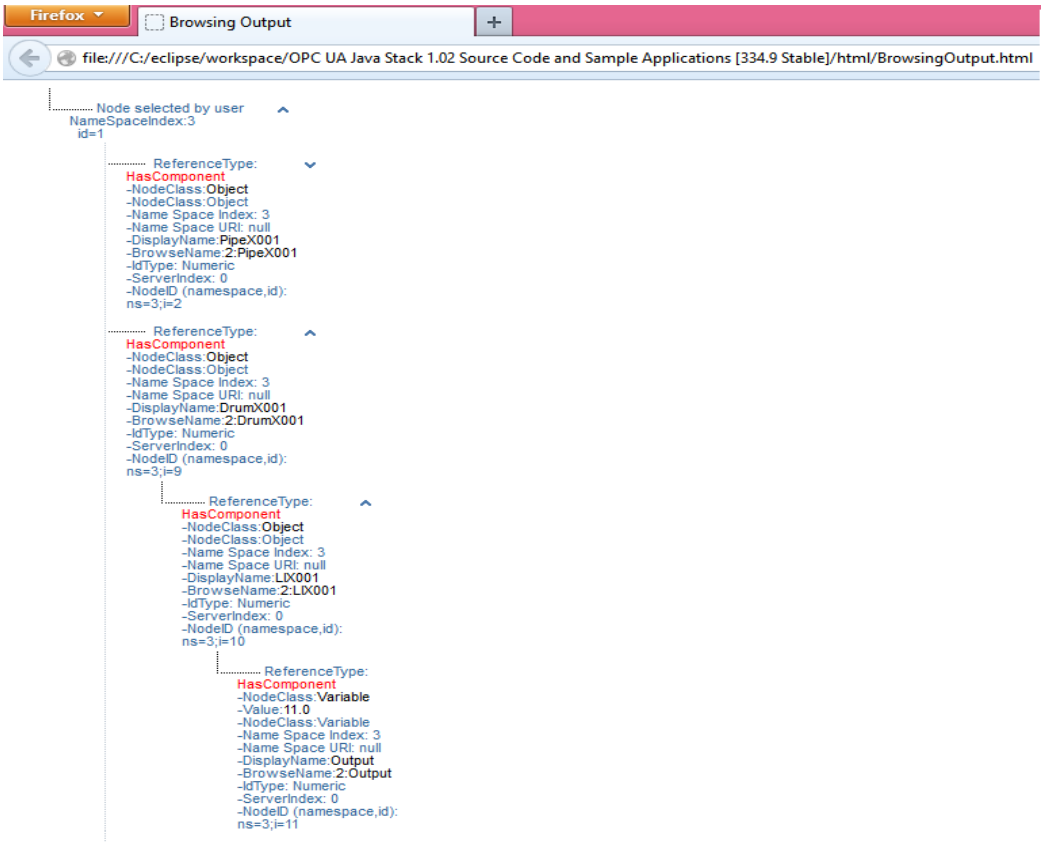
```
public static void readVariables() throws NumberFormatException,
ServiceFaultException, ServiceResultException, IOException {
	//LETTURA VARIABILI
	String r = getDaTastiera("Vuoi Fare la Lettura delle Variabili  (y/n) ?");
	if (r.equalsIgnoreCase("y"))
	do {
		int aux;
		do{
			aux = Integer.parseInt(getDaTastiera("Vuoi Fare la Lettura di una
singola variabile (1)\n"+
		" o il browsing delle variabili a partire da un nodo dell'Information
Model?(2)"));
		}while(aux<1 || aux>2);

		switch (aux){
			//lettura di una variabile cui è già noto il namespaceindex e l'id
			case 1:

			// Read a variable with namespaceindex ns, nodeId i
				ReadResponse res4 = mySession.Read(
							null,
							500.0,
							TimestampsToReturn.Source,
							new ReadValueId(newNodeId(), Attributes.Value,
null, null )
				 );
				System.out.println(res4.getResults()[0].getValue());
				break;
			case 2:
				//lettura delle variabili esposte dal server a partire da
un deterimanto nodo radice
				//da inserire in input

				NodeId referenceSelected;
				NodeId rootOfBrowsing;
				//selezione della root cui partire il browsing


				rootOfBrowsing = selectRoot();

				//selezione delle reference cui filtrare il browsing
				referenceSelected=selectReference();

				//selezione della profondità del browsing
				profonditàTot=selectDepth();
				//set flags per visualizzazione a video sole variabili
				flags[0]=true;
				flags[1]=false;
				flags[2]=false;

				browsing(rootOfBrowsing,referenceSelected);

				//reset flags
				flags[0]=true;
				flags[1]=true;
				flags[2]=true;
				break;
		 }
				r = getDaTastiera("Vuoi Continuare a Leggere (y/n) ?");
	    } while (r.equalsIgnoreCase("y"));}
```

Un esempio di esplorazione di variabili potrebbe essere effettuato usando come reference di input *HasComponent* partendo da un nodo root come ad esempio uno dei figli di Object Folder e scoprire le variabili esposte, usando una profondità pari ad esempio a 10 (vedere [OPC UA part3] per l'organizzazione dell'Address Space).
Qui vengono riportate le variabili esposte dal boiler#1 e boiler#2 relativi al server DOTNET in questione:





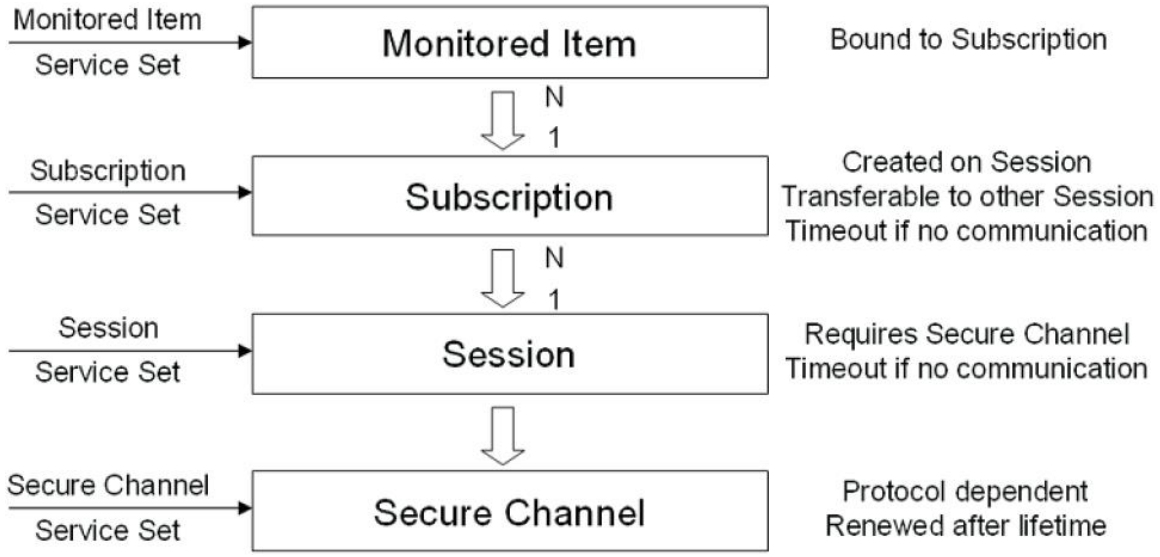

Firefox
Browsing Output
file:///C:/eclipse/workspace/OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]/html/BrowsingOutput.html
Node selected by user
NameSpaceIndex:2
id=138
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 2
-Name Space URI: null
-DisplayName:() Pipe1001
-BrowseName:2:PipeX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=2;i=139
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 2
-Name Space URI: null
-DisplayName:() Drum1001
-BrowseName:2:DrumX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=2;i=154
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 2
-Name Space URI: null
-DisplayName:LIX001
-BrowseName:2:LIX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=2;i=155
ReferenceType:
HasComponent
-NodeClass:Variable
-Value:5.0
-NodeClass:Variable
-Name Space Index: 2
-Name Space URI: null
-DisplayName:Output
-BrowseName:2:Output
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=2;i=156


Firefox
Browsing Output
file:///C:/eclipse/workspace/OPC UA Java Stack 1.02 Source Code and Sample Applications [334.9 Stable]/html/BrowsingOutput.html
Node selected by user
NameSpaceIndex:3
id=1
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 3
-Name Space URI: null
-DisplayName:PipeX001
-BrowseName:2:PipeX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=3;i=2
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 3
-Name Space URI: null
-DisplayName:DrumX001
-BrowseName:2:DrumX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=3;i=9
ReferenceType:
HasComponent
-NodeClass:Object
-NodeClass:Object
-Name Space Index: 3
-Name Space URI: null
-DisplayName:LIX001
-BrowseName:2:LIX001
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=3;i=10
ReferenceType:
HasComponent
-NodeClass:Variable
-Value:11.0
-NodeClass:Variable
-Name Space Index: 3
-Name Space URI: null
-DisplayName:Output
-BrowseName:2:Output
-IdType: Numeric
-ServerIndex: 0
-NodeID (namespace,id):
ns=3;i=11

7.Subscription

7.1 Intro

Gli OPC UA Services non sono di tipo stateless e non possono essere chiamati senza stabilire un contesto di comunicazione in livelli differenti. Per questa ragione molti di questi servizi non sono usati per trasferire dati, ma per creare, mantenere e modificare questi livelli differenti di comunicazione.



Per quanto riguarda le Subscription, è possibile creare nel contesto della sessione, subscription multiple e ogni singola subscription è usata per scambiare *data changes* e *EventNofication*s tra client e server. Una subscription richiede una sessione per trasportare dati al client, ma essa può trasferita in un'altra sessione, in base al contesto e problematiche d'utilizzo.
La lifetime di una subscription è indipendente da quello di una sessione ed ha un timeout che viene resettato ogni volta che un dato o un messaggio viene inviato al client.
MonitoredItems possono essere creati e associati ad una Subscription. Un Monitored Item è usato per definire un attributo di un nodo che deve essere monitorato per i cambiamenti del valore dei dati esposti dal nodo stesso o per definire gli "Event Source" che devono essere monitorate per la notifica degli eventi (EventNotifications).

In questo progetto è stata testata la creazione delle Subscriptions e di Monitored Items solo per monitorare il valore delle variabili e non per notifiche di eventi.
In questa parte vengono richiamati semplicemente i vari service in ordine e linearmente per come sono definiti sullo standard, senza nessuna ricorsione o costruzione di algoritmi più sofisticati, demandando allo standard [UA part 4] ed a eventuali manuali la descrizione minuziosa di tutte le parametrizzazioni di questi services.
Questa parte è suddivisa in tre funzioni quali

- creazione delle subscription
- creazione dei monitored item
- attivazione delle notifiche

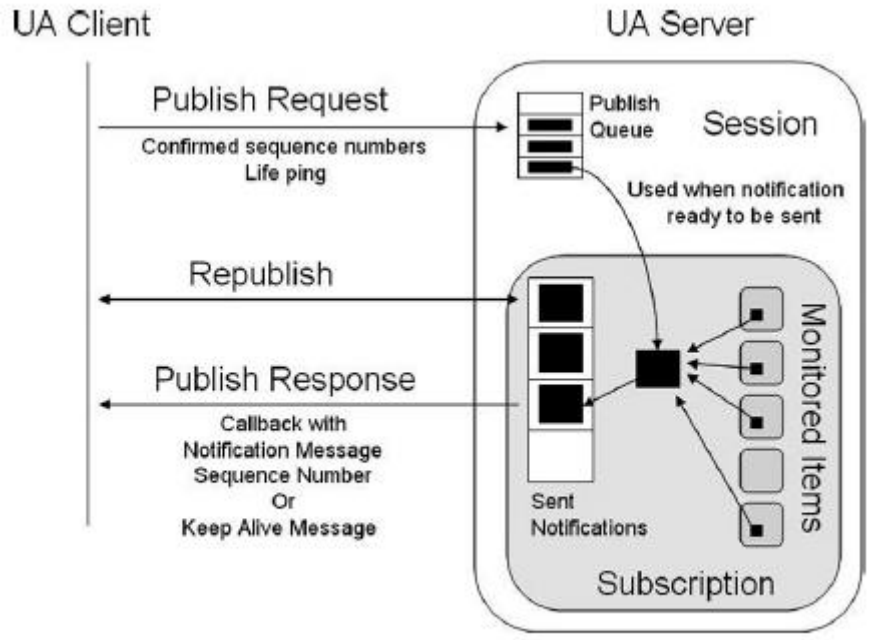
7.2 OPC UA Subscription

Le subscriptions sono usate per riportare notifiche (*Notifications*) al client relative a dati o eventi generati ad esempio da device come sensori, ed in generale qualunque *Node Attribute* che rappresenti valori sensibili da monitorare, etc. Le subscriptions hanno un set di MonitoredItems assegnate loro dal Client. I MonitoredItems generano Notifications che dovranno essere riportate al Client tramite la Subscription.

Il *Publishing Interval* di una subscription definisce il cyclic rate con il quale la Subscription viene eseguita. Ogni volta che viene eseguita, essa attende per inviare un NotificationMessage al client. Un *NotificationMessage* contiene notifications che non sono ancora state consegnate al Client.
Il publishing interval definisce anche il tempo di campionamento per il proprio MonitoredItems.

I NotificationMessages vengono inviati al Client come risposta ad una *Publish Request* inviata dal Client stesso. Le Publish Request sono accodate alla sessione non appena vengono ricevute, e se ci sono Notifications da riportare al Client, ognuna viene tolta dalla coda e processata da una subscription relativa alla sessione ad ogni *Publishing cycle*. Se non ci sono Notifications, la Publish request non viene tolta dalla coda della sessione, ed il Server aspetta fin quando al prossimo ciclo non siano pronte altre Notifications.

All'inizio di un ciclo, se ci sono Notifications da inviare, ma non ci sono Publish Request in coda, il server entra in uno stato d'attesa per la ricezione di Publish request. Quando ne viene ricevuto uno, esso viene processato immediatamente senza aspettare il prossimo publishing cycle.



Delivery of notification messages

I NotificationMessage sono univocamente identificati da un *sequence number* che permette ai Client di ricercare eventuali messaggi non ricevuti, ed è posto all'interno della Response.

Le Subscriptions hanno un *keep-alive counter* che conteggia il numero di cicli di publishing consecutivi nel quale non ci sono state Notification da riportare al client. Quando il conteggio massimo di keep-alive è stato raggiunto, una Publish Request viene eliminata dalla coda ed usata per ritornare un *keep-alive Message* al client. Questo messaggio informa il client che la Subscription è ancora attiva.
Il valore massimo del parametro keep-alive (*MaxKeepAlive*) viene settato dal client al momento della creazione della subscription, e può essere in seguito modificato usando il *ModifySubscription Service*, non implementato in questo progetto.

L'attività di publishing di una subscription può essere abilitata o disabilitata dal Client al momento della creazione della subscription stessa, o in seguito modificare l'abilitazione tramite il *SetPublishingMode Service*. Il disabilitare il publishing causa lo stop da parte della Subscription degli invii di NotificationMessages al client. Comunque, la subscription continua ad eseguire ciclicamente e continua ad inviare keep-alive Message al Client.

Le subscriptions hanno un *lifetime counter* che rappresenta il tempo di vita di una subscription. Quando il numero di publishing cycles consecutivi nei quali non ci sono state Publish request da parte del client raggiunge questo valore, (ovvero la Subscription ha Notification Message pronti ma non può inviarli per assenza di richieste da parte del client), il server deve cancellare la subscription. Il lifetime counter deve essere impostato ad un minimo di tre volte il valore di Max keepAlive.
Il valore viene impostato al momento della creazione della subscription.

7.3 OPC UA CreateSubscription Service

Nel progetto, tra i vari Service relativi alle Subscription è stato utilizzato quello base, il ***CreateSubscription Service***, facente parte al ***Subscription Service Set***, dedicato appunto alla creazione delle Susbcription.
Di seguito è riportata la tabella dei parametri di Request e Response del servizio.

CreateSubscription Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
requestedPublishing Interval	Duration	This interval defines the cyclic rate that the *Subscription* is being requested to return *Notifications* to the *Client*. This interval is expressed in milliseconds. This interval is represented by the publishing timer in the *Subscription* state table (see 5.13.1.2). The negotiated value for this parameter returned in the response is used as the default sample interval for *MonitoredItems* assigned to this *Subscription*. The value 0 is invalid.
requestedLifetimeCount	Counter	Requested lifetime count (see 7.5 for *Counter* definition). The lifetime count shall be a mimimum of three times the keep keep-alive count. When the publishing timer has expired this number of times without a *Publish* request being available to send a *NotificationMessage*, then the *Subscription* shall be deleted by the *Server*.
requestedMaxKeepAlive Count	Counter	Requested maximum keep-alive count (see 7.5 for *Counter* definition). When the publishing timer has expired this number of times without requiring any *NotificationMessage* to be sent, the *Subscription* sends a keep-alive *Message* to the *Client*. The value 0 is invalid.
maxNotificationsPerPublish	Counter	The maximum number of notifications that the *Client* wishes to receive in a single *Publish* response. A value of zero indicates that there is no limit.
publishingEnabled	Boolean	A *Boolean* parameter with the following values : TRUE publishing is enabled for the *Subscription*. FALSE publishing is disabled for the *Subscription*. The value of this parameter does not affect the value of the monitoring mode *Attribute* of *MonitoredItems*.
priority	Byte	Indicates the relative priority of the *Subscription*. When more than one *Subscription* needs to send *Notifications*, the *Server* should dequeue a Publish request to the *Subscription* with the highest *priority* number. For *Subscriptions* with equal *priority* the *Server* should dequeue Publish requests in a round-robin fashion. When the keep-alive period expires for a *Subscription* it shall take precedence regardless of its *priority*, in order to prevent the *Subscription* from expiring. A Client that does not require special priority settings should set this value to zero.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* (see 7.13 for *IntegerId* definition). This identifier shall be unique for the entire *Server*, not just for the *Session*, in order to allow the *Subscription* to be transferred to another *Session* using the TransferSubscriptions service.
revisedPublishingInterval	Duration	The actual publishing interval that the *Server* will use, expressed in milliseconds. The *Server* should attempt to honor the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.
revisedLifetimeCount	Counter	The lifetime of the *Subscription* shall be a minimum of three times the keep-alive interval negotiated by the *Server*.
revisedMaxKeepAliveCount	Counter	The actual maximum keep-alive count (see 7.5 for *Counter* definition). The *Server* should attempt to honor the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.

I parametri relativi alla request sono già stati esposti nel paragrafo precedente ad esclusione della priorità, la quale indica appunto la priorità della Subscription nel contesto in cui più Subscriptions necessitano di inviare Notifications, e dove il Server dovrà selezionare dalla coda la publish request della subscription con la più alta priorità.
Il codice relativo all'OPC UA Java è dato dalle righe seguenti

```
CreateSubscriptionRequest subscription = newCreateSubscriptionRequest(null,
                                requestedPublishingInterval, requestedLifetimeCount,
```

```
                                requestedMaxKeepAliveCount,
maxNotificationsPerPublish,
                                Default_PublishingEnabled, Default_Priority);
            CreateSubscriptionResponse response =
mySession.CreateSubscription(subscription);
```

Come ben visibile, la creazione di un oggetto subscription avviene tramite la Classe `CreateSubscriptionRequest` relativa alle API di OPC UA Java, la quale rappresenta una request da inviare al server. Per poter attivare la subscription bisogna richiamare `mySession.CreateSubscription(subscription)`, il quale associerà la request alla relativa sessione attivata e la invierà al server. Il commando ritornerà una response contenente l'id che identifica la subscription appena creata (che rappresenta a tutti gli effetti un ***Server Handle*** come utilizzato in OPC DCOM e OPC XML utilizzabile dal client per effettuare le Request successive) ed il valore dei parametri che il server riesce effettivamente a supportare (revised parameters) rispetto a quelli richiesti dal client. Nel progetto è stata creata una solo subscription ma è possibile generarne più, con eventuali strutture dati per salvare il proprio id, ovvero gli handle.

7.4 Funzione createSubscription

La funzione relativa al progetto crea semplicemente una subscription, con l'inserimento da tastiera da parte dell'utente dei vari parametri già discussi. Cosa più importante da notare è che al momento della creazione della subscription viene impostato inserito il parametro `requestedPublishingInterval` ovvero la frequenza di esecuzione della subscription, frequenza ereditata anche dai MonitoredItem che verranno successivamente allegati alla subscription (analogia in OPC DCOM dove la frequenza relativa agli OPC Group che viene ereditata dagli OPC Item in esso contenuti).

```
public static void createSubscription() throws ServiceFaultException,
ServiceResultException {
      System.out.println("\nInserisci i Parametri di Sottoscrizione :");
      System.out.println("Requested Publishing Interval [DEFAULT: " +
                        Default_RequestedPublishingInterval + "]:");
      try{
            requestedPublishingInterval = Double.parseDouble(in.readLine());
      }catch(Exception e){
            requestedPublishingInterval = Default_RequestedPublishingInterval;
      }


      System.out.println("RequestedMaxKeepAliveCount [DEFAULT: " +
                  Default_RequestedMaxKeepAliveCount + "]:");
      try{
            requestedMaxKeepAliveCount =
UnsignedInteger.parseUnsignedInteger(in.readLine());
      }catch(Exception e){
            requestedMaxKeepAliveCount = Default_RequestedMaxKeepAliveCount;
      }

      // RequestedLifetimeCount> 3 * RequestedMaxKeepAliveCount
```

```
      do {
      System.out.println("RequestedLifetimeCount [DEFAULT: " +
                  Default_RequestedLifetimeCount + "]:");
      System.out.println("Nota: deve essere superiore a 3 volte il
RequestedMaxKeepAliveCount ");

      try{
            requestedLifetimeCount =
UnsignedInteger.parseUnsignedInteger(in.readLine());
      }catch(Exception e){
            requestedLifetimeCount = Default_RequestedLifetimeCount;
         }
      } while
(requestedLifetimeCount.compareTo(requestedMaxKeepAliveCount.intValue()*3)<=0);

      System.out.println("Maximum Number of Notifications in a Single Publishresponse
[DEFAULT: " +
                  Default_MaxNotificationsPerPublish + "]:");
      try{
            maxNotificationsPerPublish =
UnsignedInteger.parseUnsignedInteger(in.readLine());
      }catch(Exception e){
            maxNotificationsPerPublish = Default_MaxNotificationsPerPublish;
      }

      System.out.println("Il parametro publishingEnabled viene fissato a True ");
      System.out.println("La Priorita' assegnata alla Subscription e' : " +
                  Default_Priority);

      CreateSubscriptionRequest subscription = newCreateSubscriptionRequest(null,
                              requestedPublishingInterval, requestedLifetimeCount,
                              requestedMaxKeepAliveCount,
maxNotificationsPerPublish,
                              Default_PublishingEnabled, Default_Priority);
            CreateSubscriptionResponse response =
mySession.CreateSubscription(subscription);

            subId=response.getSubscriptionId();

      System.out.println("Subscription Creata, il suo Id (SubscriptionId) e' :" +
                              subId);

            System.out.println("    Il RevisedRequested Publishing Interval:" +
                  response.getRevisedPublishingInterval());

            System.out.println("    Il RevisedLifetimeCount :" +
                              response.getRevisedLifetimeCount());

            System.out.println("    Il RevisedMaxKeepAliveCount:" +
                              response.getRevisedMaxKeepAliveCount());

}
```

8.Monitored Item

8.1 OPC UA MonitoredItem

I Client definiscono i *MonitoredItems* per sottoscriversi a dati ed eventi. Ogni MonitoredItem identifica l'item che deve essere monitorato e la Subscription da usare per inviare Notifications. L'item che deve essere monitorato potrebbe essere un qualsiasi *Node Attribute*.

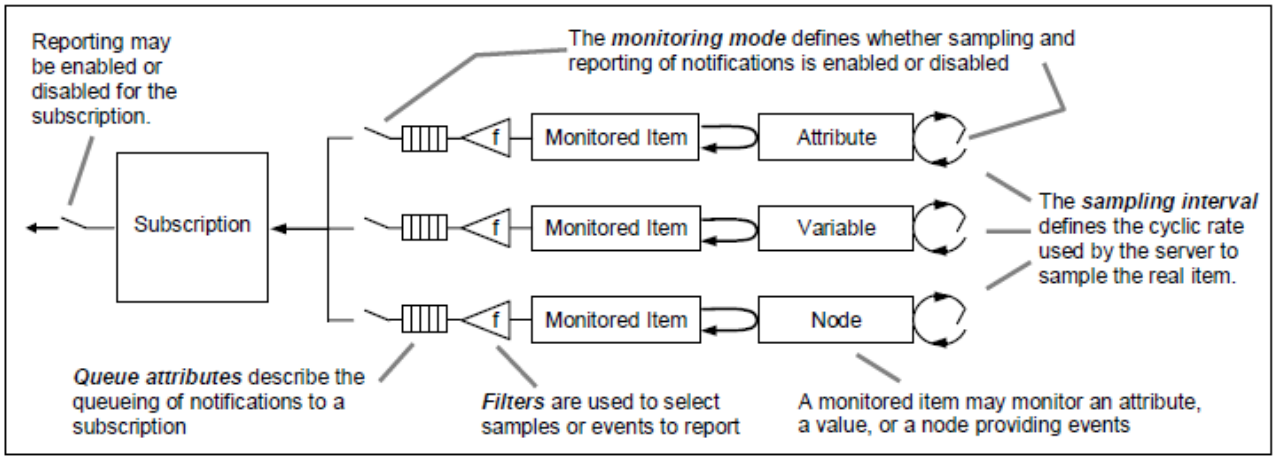
Le Notifications sono strutture dati che descrivono l'occorrenza di "data changes" ed Event. Vengono impacchettati in NotificationMessages per essere trasferiti al client. La Subscription periodicamente invia NotificationMessages in un intervallo definito dall'utente (Publishing Interval).

8.2 Parametrizzazioni

Quattro parametri primari sono definiti per i MonitoredItems che infomano il Server sul come l'item deve essere campionato, valutato e riportato. Questi parametri sono:

- Sampling interval
- Monitoring mode
- Il filtro
- Coda dei parametri

La figura seguente espone il funzionamento di subscription e MonitoredItem e dove agisce la parametrizzazione.



MonitoredItem Model

8.2.1 Sampling interval

Ad ogni monitoredItem creata dal client è assegnato un sampling interval che è ereditato dal publishing interval della Subscription o che è definito esplicitamente, sovrascrivendolo.
Il sampling interval indica il rate più veloce al quale un Server dovrebbe campionare la sorgente per i data changes.

Un client deve definire un sampling interval pari a 0 se crea subscription per eventi.

Il sampling interval assegnato definisce un cycle rate "best effort" che il Server usa per campionare l'item dalla sua sorgente. Il best effort comporta che il tempo tra i valori ritornati dal client devono essere maggiori o uguali al sampling inteval, ovvero il server fa del suo meglio per campionare al rate definito.

Il client potrebbe anche specificare 0 per il sampling interval, il quale indica che il Server dovrebbe usare il rate di campionamento più.
Ci si aspetta che i server supporteranno solo un limitato set di intervalli di campionamento per ottimizzare le proprie operazioni. Se l'intervallo esatto richiesto dal client non è supportato dal server, il server assegnerà al monitoredItem l'intervallo più appropriato, ritornandolo al client.
Sebbene il server scelga la frequenza di campionamento, il sistema hardware che il server rappresenta può essere costituito da tecnologie con frequenze fisiche di funzionamento ancora più basse, le quali determineranno l'effettive frequenze con cui rilevare data changes.

8.2.2 Monitoring mode

Il parametro monitoring mode è usato per abilitare e disabilitare il campionamento dei MonitoredItems, e anche a provvedere per abilitare e disabilitare indipendentemente il reporting di Notifications. Questa capacità permette ad un MonitoredItem di essere configurato per campionare, campionare e fare report o nessuno dei due. Disabilitare il campionamento non fa cambiare i valori di qualunque altro parametro del MonitoredItem come per esempio l'intervallo di campionamento.

MonitoringMode Values

Value	Description
DISABLED_0	The item being monitored is not sampled or evaluated, and *Notifications* are not generated or queued. *Notification* reporting is disabled.
SAMPLING_1	The item being monitored is sampled and evaluated, and *Notifications* are generated and queued. *Notification* reporting is disabled.
REPORTING_2	The item being monitored is sampled and evaluated, and *Notifications* are generated and queued. *Notification* reporting is enabled.

8.2.3 Filter

Ogni volta che un MonitoredItem è campionato, il server valuta il campione usando il filtro definito per il MonitoredItem. Il parametro filtro definisce un criterio che il Server usa per determinare se una Notification dovrebbe essere generata per quel campione. Il tipo di filtro è dipendente dal tipo di item monitorato.

8.2.3.1 MonitoringFilter parameters

Altre parametrizzazioni importanti riguardano i filtri. Il *CreateMonitoredItem service* (7.3) permette di specificare un filtro per ogni MonitoredItem, Il MonitoringFilter è un parametro estendibile la cui struttura dipende dal tipo di item che deve essere monitorato. Esistono dei parametri base, esplicitati dalla seguente tabella e altri parametri Extensible definiti all'interno dello standard.

MonitoringFilter parameterTypeIds

Symbolic Id	Description
DataChangeFilter	The change in a data value that shall cause a *Notification* to be generated.
EventFilter	If a *Notification* conforms to the *EventFilter*, the *Notification* is sent to the *Client*.
AggregateFilter	The aggregate and its intervals when it will be calculated and a Notification is generated.

Nel progetto sono stati impostati i parametri relativi al DataChangeFilter, che definiscono le condizioni sotto le quali una Notification relativa ai data changes deve essere riportata al cliente ed opzionalmente, il range di oscillazione dei valori per i quali non deve essere inviata la Notification (DeadBand). Di seguito la tabella mostrerà i parametri.

DataChangeFilter

Name	Type	Description
DataChangeFilter	structure	
trigger	enum DataChangeTrigger	Specifies the conditions under which a data change notification should be reported. It has the following values : STATUS_0 Report a notification ONLY if the *StatusCode* associated with the value changes. See Table 166 for *StatusCodes* defined in this Part. Part 8 specifies additional *StatusCodes* that are valid in particular for device data STATUS_VALUE_1 Report a notification if either the *StatusCode* or the value change. The *Deadband* filter can be used in addition for filtering value changes. **This is the default setting if no filter is set.** STATUS_VALUE_TIMESTAMP_2 Report a notification if either StatusCode, value or the *SourceTimestamp* change. The *Deadband* filter can be used in addition for filtering value changes. If the DataChangeFilter is not applied to the monitored item, STATUS_VALUE_1 is the default reporting behaviour.
deadbandType	UInt32	A value that defines the *Deadband* type and behaviour. Value deadbandType None_0 No *Deadband* calculation should be applied. Absolute_1 AbsoluteDeadband (see below) Percent_2 PercentDeadband (This type is specified in Part 8).
deadbandValue	Double	The *Deadband* is applied only if * the *trigger* includes value changes and * the *deadbandType* is set appropriately. Deadband is ignored if the status of the data item changes. ***DeadbandType = AbsoluteDeadband***: For this type the *deadbandValue* contains the absolute change in a data value that shall cause a *Notification* to be generated. This parameter applies only to *Variable*s with any number data type. An exception that causes a *DataChange Notification* based on an AbsoluteDeadband is determined as follows: Exception if **(absolute value of (last cached value - current value) > *AbsoluteDeadband*)** The last cached value is defined as the most recent value previously sent to the *Notification* channel. If the item is an array of values, the entire array is returned if any array element exceeds the AbsoluteDeadband. ***DeadbandType = PercentDeadband:*** This type is specified in Part 8

8.2.4 Coda dei parametri

Se il sample passa il criterio di filtraggio del filtro, una Notification è generata e posta in coda per essere trasferita dalla subscription. La dimensione della coda è definita quando il MonitoredItem è creato. Quando la coda è piena ed una nuova Notification è ricevuta, il server può scartare la notification più datata ed inserire la nuova, o semplicemente scartare quella nuova. La politica di scarto è decisa al momento della creazione del MonitoredItem. Di seguito viene posta una tabella riassuntiva dei Monitoring Parameter.

MonitoringParameters

Name	Type	Description
MonitoringParameters	structure	Parameters that define the monitoring characteristics of a *MonitoredItem*.
clientHandle	IntegerId	*Client*-supplied id of the *MonitoredItem*. This id is used in *Notifications* generated for the list *Node*. The *IntegerId* type is defined in 7.13.
samplingInterval	Duration	The interval that defines the fastest rate at which the *MonitoredItem*(s) should be accessed and evaluated. This interval is defined in milliseconds. The value 0 indicates that the *Server* should use the fastest practical rate. The value -1 indicates that the default sampling interval defined by the publishing rate of the *Subscription* is used. The *Server* uses this parameter to assign the *MonitoredItems* to a sampling interval that it supports.
filter	Extensible Parameter MonitoringFilter	A filter used by the *Server* to determine if the *MonitoredItem* should generate a *Notification*. If not used, this parameter is null. The *MonitoringFilter* parameter type is an extensible parameter type specified in 7.16. It specifies the types of filters that can be used.
queueSize	Counter	The requested size of the *MonitoredItem* queue. The following values have special meaning: Value — Meaning 1 — the queue has a single entry, effectively disabling queuing. >1 — a first-in-first-out queue is to be used. Max Value — the max size that the *Server* can support. This is used for *Event Notifications*. In this case the *Server* is responsible for the *Event* buffer. If 0 is passed by the client, the server returns the default queue size which shall be 1 as *revisedQueueSize* for data monitored items. For event monitored items the value is ignored and the server shall set the supported maximum queue size as *revisedQueueSize*. If *Events* are lost an *Event* of the type *EventQueueOverflowEventType* is generated.
discardOldest	Boolean	A boolean parameter that specifies the discard policy when the queue is full and a new *Notification* is to be enqueued. It has the following values: TRUE — the oldest (first) *Notification* in the queue is discarded. The new *Notification* is added to the end of the queue. FALSE — the new *Notification* is discarded. The queue is unchanged.

8.3 OPC UA CreateMonitoredItems

Il service utilizzato in questa parte di programma è il ***CreateMonitoredItem Service***, appartenente al ***MonitoredItem Service Set***.
Il servizio è usato per creare e aggiungere uno o più MonitoredItems ad una Subscription.
Un MonitoredItem è cancellato automaticamente dal Server quando una Subscription è cancellata.
Chiamando questo service ripetutamente per aggiungere un piccolo numero di MonitoredItems ogni volta potrebbe rallentare le prestazioni del server.
È consigliabile aggiungere un completo set di MonitoredItems alla subscription quando possibile.
Di seguito la tabella dei parametri relativo al servizio.

CreateMonitoredItems Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* that will report *Notifications* for this *MonitoredItem* (see 7.13 for *IntegerId* definition).
timestampsToReturn	enum TimestampsToReturn	An enumeration that specifies the timestamp *Attributes* to be transmitted for each *MonitoredItem*. The *TimestampsToReturn* enumeration is defined in 7.34. When monitoring *Events*, this applies only to *Event* fields that are of type *DataValue*.
itemsToCreate []	MonitoredItem CreateRequest	A list of *MonitoredItems* to be created and assigned to the specified *Subscription*.
itemToMonitor	ReadValueId	Identifies an item in the *AddressSpace* to monitor. To monitor for *Events*, the *attributeId* element of the *ReadValueId* structure is the id of the *EventNotifier Attribute*. The *ReadValueId* type is defined in 7.23.
monitoringMode	enum MonitoringMode	The monitoring mode to be set for the *MonitoredItem*. The *MonitoringMode* enumeration is defined in 7.17.
requestedParameters	Monitoring Parameters	The requested monitoring parameters. *Servers* negotiate the values of these parameters based on the *Subscription* and the capabilities of the *Server*. The *MonitoringParameters* type is defined in 7.15.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	MonitoredItem CreateResult	List of results for the *MonitoredItems* to create. The size and order of the list matches the size and order of the *itemsToCreate* request parameter.
statusCode	StatusCode	*StatusCode* for the *MonitoredItem* to create (see 7.33 for *StatusCode* definition).
monitoredItemId	IntegerId	*Server*-assigned id for the *MonitoredItem* (see 7.13 for *IntegerId* definition). This id is unique within the *Subscription*, but might not be unique within the *Server* or *Session*. This parameter is present only if the *statusCode* indicates that the *MonitoredItem* was successfully created.
revisedSampling Interval	Duration	The actual sampling interval that the *Server* will use. This value is based on a number of factors, including capabilities of the underlying system. The Server shall always return a *revisedSamplingInterval* that is equal or higher than the *requestedSamplingInterval*. If the *requestedSamplingInterval* is higher than the maximum sampling interval supported by the *Server*, the maximum sampling interval is returned.
revisedQueueSize	Counter	The actual queue size that the *Server* will use.
filterResult	Extensible Parameter MonitoringFilter Result	Contains any revised parameter values or error results associated with the *MonitoringFilter* specified in the request. This parameter may be omitted if no errors occurred. The *MonitoringFilterResult* parameter type is an extensible parameter type specified in 7.16.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *MonitoredItems* to create (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *itemsToCreate* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

La porzione di codice relativo a questo servizio in OPC UA è il seguente

```
//collego alla sessione i monitoreditems agganciati alla subsciption che ho creato
prima
	CreateMonitoredItemsRequest mi = new CreateMonitoredItemsRequest();
	mi.setItemsToCreate(monitoredItems);
	mi.setSubscriptionId(subId);
	// viene richiesto il timestamp del server
	// in alternativa si poteva chiedere il timestamp della sorgente (source)
	mi.setTimestampsToReturn(TimestampsToReturn.Server);


	CreateMonitoredItemsResponse w = mySession.CreateMonitoredItems(mi);
```

Viene creata una request cui vengono associati dei monitoredItem creati precedentemente, l'id della Subscription a cui devono essere associati. Infine viene inviata la response tramite la sessione e catturato il response.

8.4 Funzione createMonitoredItems

La funzione implementa la creazione dei monitoredItems, la parametrizzazione (già discussa in 5.1), la creazione e parametrizzazione dei filtri, la selezione degli item da monitorare con il relativo binding con i nodi variable da monitorare (namespacexindex = 2, id = 156//namespaceindex = 3, id = 11), il binding del MonitoredItem creato con la relativa sessione attiva.

```
public static void createMonitoredItems() throws ServiceFaultException,
ServiceResultException {
	int nc;
	UnsignedInteger NumberMonitoredItems;
	UnsignedInteger Default_NumberMonitoredItems = new UnsignedInteger(1);

	System.out.println("Inserisci il Numero di MonitoredItems da Creare [DEFAULT =
1 ]:");
	try{
		NumberMonitoredItems =
UnsignedInteger.parseUnsignedInteger(in.readLine());
	}catch(Exception e){
		NumberMonitoredItems = Default_NumberMonitoredItems;
	}

	System.out.println("Numero di MonitoredItems da creare
"+NumberMonitoredItems.intValue());

	MonitoredItemCreateRequest[] monitoredItems = new
MonitoredItemCreateRequest[NumberMonitoredItems.intValue()];

	for(nc = 0; nc<NumberMonitoredItems.intValue(); nc++){

		monitoredItems[nc] = new MonitoredItemCreateRequest();
		MonitoringParameters reqParams = new MonitoringParameters();

		//Procedura di Inserimento dei valori del parametro itemToMonitor
```

```
      //prima creo l'oggetto nodeId di tipo NodeId da inserire nella lista dei
Monitored Item
            //Inserisco i valori del parametro itemToMonitor
            System.out.println("Procedura di Inserimento dei valori del parametro
itemToMonitor \n"+
                        "prima creo l'oggetto nodeId di tipo NodeId da inserire
nella lista dei Monitored Item ");
            monitoredItems[nc].setItemToMonitor(newReadValueId(newNodeId(),
Attributes.Value, null, null ) );

            //Procedura di Inserimento dei valori del parametro RequestedParameters

            System.out.println("SamplingInterval [DEFAULT: 1000]:");
            try{
                  samplingInterval = Double.parseDouble(in.readLine());
            }catch(Exception e){
                  samplingInterval = Default_SamplingInterval;
            }

            System.out.println("Queue Size: [DEFAULT: 4]");
            try{
                  queueSize = newUnsignedInteger(in.readLine());
            }catch(Exception e){
                  queueSize = Default_QueueSize;
            }

            System.out.println("DiscardOldest (true or false) [DEFAULT: false]:");
            try{
                  discardOldest = Boolean.parseBoolean(in.readLine());
            }catch(Exception e){
            discardOldest = Default_DiscardOldest;
            }
      chandle=newUnsignedInteger(nc+1);
            reqParams.setClientHandle(chandle);    //identificativo per il
riconoscimento del monitoriteditems
      System.out.println("Il Client Handle assegnato al Monitored Item e' : " +
chandle);
      reqParams.setSamplingInterval(samplingInterval); //intervallo di campionamento
      reqParams.setQueueSize(queueSize);     //grandezza della coda per le notifiche
      reqParams.setDiscardOldest(discardOldest);   //politica di gestione di buffer
overflow

            System.out.println("Deadband: (1.00 -> Absolute / 0.5% ->Percent )
[DEFAULT: 0.01]");
            try{
                  String a = in.readLine();
                  if(a.contains("%")){
                        percentDeadBand = Double.parseDouble(a.split("%")[0]);
                        absoluteDeadBand = null;
                  }else{
                        absoluteDeadBand = Double.parseDouble(a);
                        percentDeadBand = null;
                  }
            }catch(Exception e){
                              absoluteDeadBand = Default_AbsoluteDeadBand;
                              percentDeadBand = null;
            }
      DataChangeFilter filter = new DataChangeFilter();
            if(absoluteDeadBand != null){
                  filter.setDeadbandValue(absoluteDeadBand);
```

```
                    filter.setTrigger(DataChangeTrigger.StatusValue);

      filter.setDeadbandType(UnsignedInteger.valueOf(DeadbandType.Absolute.getValue()
));
            }else{
                  filter.setDeadbandValue(percentDeadBand);
                  filter.setTrigger(DataChangeTrigger.StatusValue);

      filter.setDeadbandType(UnsignedInteger.valueOf(DeadbandType.Percent.getValue())
);
            }
            ExtensionObject fil = ExtensionObject.binaryEncode(filter);
            reqParams.setFilter(fil);

            //Inserisco i valori del parametro RequestedParameters
            monitoredItems[nc].setRequestedParameters(reqParams);

            //Inserisco i valori del parametro monitoringMode
            System.out.println("Il Monitoring Mode viene settato allo stato
Reporting ");

            monitoredItems[nc].setMonitoringMode(MonitoringMode.Reporting);
      //con “sampling” le sorgenti dati sono campionate ma le notifiche non vengono
inviate al client (vedi setTriggering Service),
            //mentre con “reporting” si ha sempre sia il campionamento che l’inoltro

      //    System.out.println("nc = "+nc);
      //    System.out.println("Item = "+
monitoredItems[nc].getItemToMonitor().getNodeId());
      //    System.out.println("Client Handle = "+
monitoredItems[nc].getRequestedParameters().getClientHandle());
      }



      //collego alla sessione i monitoreditems agganciati alla subsciption che ho
creato prima
      CreateMonitoredItemsRequest mi = new CreateMonitoredItemsRequest();
      mi.setItemsToCreate(monitoredItems);
      mi.setSubscriptionId(subId);
      // viene richiesto il timestamp del server
      // in alternativa si poteva chiedere il timestamp della sorgente (source)
      mi.setTimestampsToReturn(TimestampsToReturn.Server);
      System.out.println("Sto per creare i seguenti Items da monitorare Creati
nell'ambito della SubscritionId " + mi.getSubscriptionId());
      for(nc = 0; nc<NumberMonitoredItems.intValue(); nc++){
            System.out.println("     Item          = "+
monitoredItems[nc].getItemToMonitor().getNodeId());
            System.out.println("     Client Handle  = "+
monitoredItems[nc].getRequestedParameters().getClientHandle());
      }

      CreateMonitoredItemsResponsew = mySession.CreateMonitoredItems(mi);

      //    System.out.println("MonitoredItems Creati nell'ambito della
SubscritionId " + subId);

}
```

9. Notifications

9.1 Intro

Lo scopo delle Notification è già stato ampiamente trattate nei capitoli 6 e 7. In questo capitolo verranno affrontate l'implementazione della Notification e la relativa parametrizzazione.

9.2 OPC UA NotificationData Parameter

La struttura di un NotificationMessage usata nei Subscription Service set permette di specificare differenti tipi di NotificationData. Il parametro NotificationData è un parametro estensibile la cui struttura dipende dal tipo di Notification che deve essere spedita. La tabella in seguito definisce le tre tipologie.

NotificationData parameterTypeIds

Symbolic Id	Description
DataChange	*Notification* data parameter used for data change *Notifications*.
Event	*Notification* data parameter used for *Event Notifications*.
StatusChange	*Notification* data parameter used for Subscription status change *Notifications*

Possono esserci notifications multiple per un singolo MonitoredItem in una singola struttura NotificationData. In questo caso il server deve assicurare che le notification appaiano nello stesso ordine con il quale sono state accodate nel MonitoredItem.

9.2.1 DataChangeNotification parameter

La tabella seguente definisce il parametro NotificationData usato nelle notifiche relative ai data changes. Questa struttura contiene i *data items* monitorati che dovranno essere riportati al client.

DataChangeNotification

Name	Type	Description
DataChangeNotification	structure	Data change *Notification* data
monitoredItems []	MonitoredItem Notification	The list of *MonitoredItems* for which a change has been detected.
clientHandle	IntegerId	*Client*-supplied handle for the *MonitoredItem*. The *IntegerId* type is defined in 7.13
value	DataValue	The *StatusCode*, value and timestamp(s) of the monitored *Attribute* depending on the sampling and queuing configuration. If the *StatusCode* indicates an error then the value and timestamp(s) are to be ignored. If not every detected change has been returned since the *Server*'s queue buffer for the *MonitoredItem* reached its limit and had to purge out data, the *Overflow* bit in the *DataValue InfoBits* of the *statusCode* is set. *DataValue* is a common type defined in 7.7.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information. The size and order of this list matches the size and order of the *monitoredItem* parameter. There is one entry in this list for each *Node* contained in the *monitoredItem* parameter. This list is empty if diagnostics information was not requested or is not available for any of the *MonitoredItems*. *DiagnosticInfo* is a common type defined in 7.8.

9.2.2 EventNotificationList parameter

Questo parametro è usato nel caso di Eventi e definisce una struttura tabellare che è usata per ritornare i campi contenenti informazioni di un evento da inviare alla Client Subscription.

La struttura è nella forma di una tabella e consiste in una o più eventi, ognuna contenente un array di uno o più campi. La selezione e l'ordine dei campi ritornati per ogni evento è identica al parametro selezionato dell'EventFilter. Questo parametro non è stato utilizzato nel progetto perché non sono stati monitorati eventi.

EventNotificationList

Name	Type	Description
EventNotificationList	structure	Event *Notification* data
events []	EventFieldList	The list of *Events* being delivered
clientHandle	IntegerId	*Client*-supplied handle for the *MonitoredItem*. The *IntegerId* type is defined in 7.13
eventFields []	BaseDataType	List of selected *Event* fields. This shall be a one to one match with the fields selected in the *EventFilter*. 7.16.3 specifies how the *Server* shall deal with error conditions.

9.2.3 StatusChangeNotification parameter

Parametro il quale informa il client su eventuali cambiamenti dello stato della subscription.

StatusChangeNotification

Name	Type	Description
StatusChangeNotification	structure	Event *Notification* data
status	StatusCode	The *StatusCode* that indicates the status change.
diagnosticInfo	DiagnosticInfo	*DiagnosticInformation* for the status change

9.3 OPC UA NotificationMessage

La tabella seguente mostra come è strutturato un messaggio di notifica inoltrato dal server.

NotificationMessage

Name	Type	Description
NotificationMessage	structure	The *Message* that contains one or more *Notifications*.
sequenceNumber	Counter	The sequence number of the *NotificationMessage*.
publishTime	UtcTime	The time that this *Message* was sent to the *Client*. If this *Message* is retransmitted to the *Client*, this parameter contains the time it was first transmitted to the *Client*.
notificationData []	Extensible Parameter NotificationData	The list of *NotificationData structures*. The *NotificationData* parameter type is an extensible parameter type specified in 7.19. It specifies the types of *Notifications* that can be sent. The *ExtensibleParameter* type is specified in 7.11. Notifications of the same type should be grouped into one NotificationData element. If a Subscription contains *MonitoredItems* for events and data, this array should have not more than 2 elements. If the *Subscription* contains *MonitoredItems* only for data or only for events, the array size should always be one for this *Subscription*.

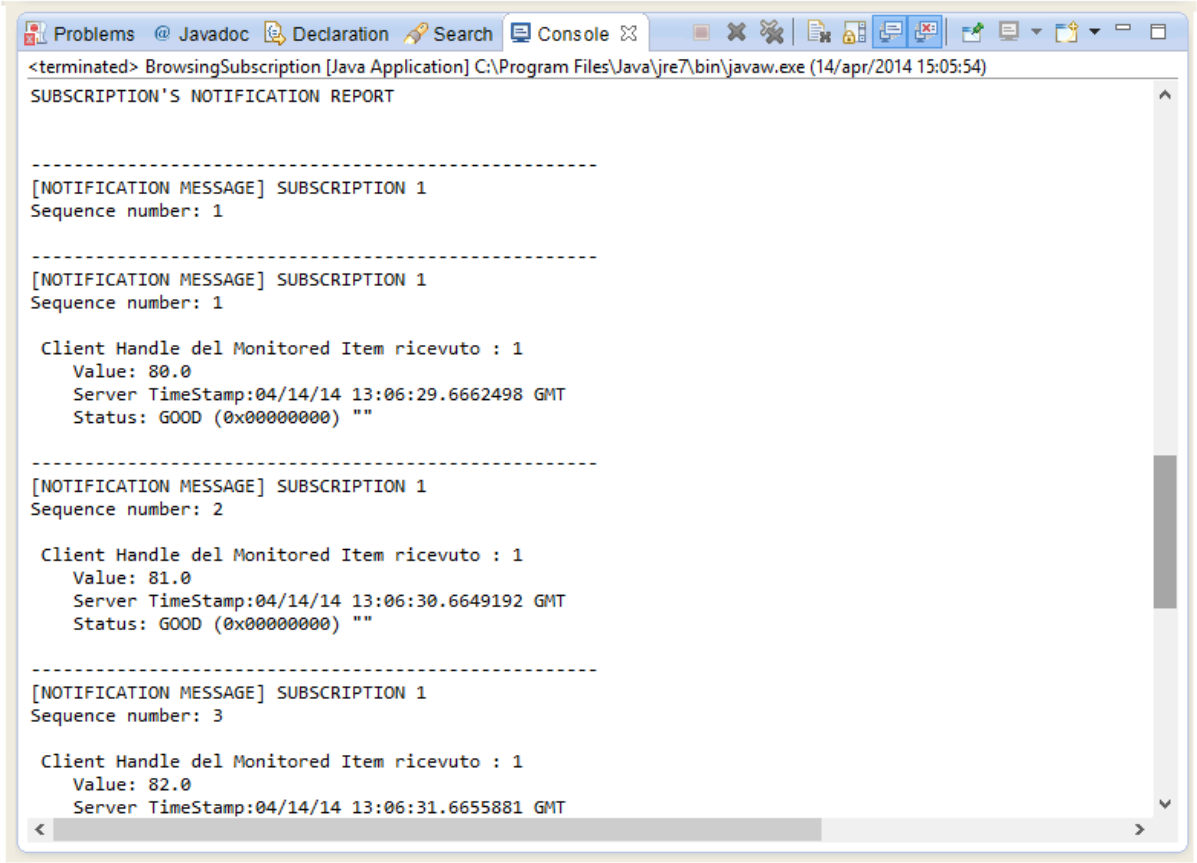
Sarà possibile poi da parte del client ottenere i vari parametri della notifica e visualizzarli a video.

9.4 Funzione activeNotification

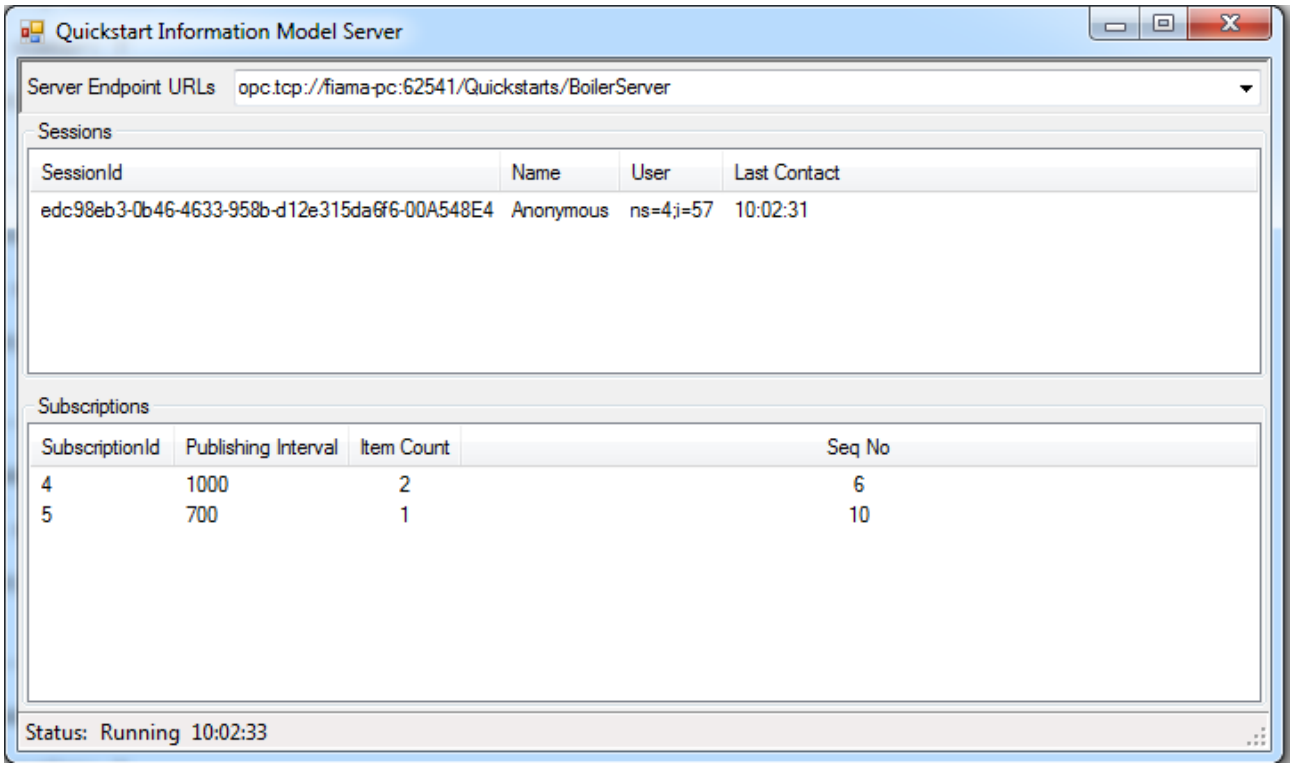
In questa funzione non viene usato nessun servizio, ma semplicemente viene deputato un thread nella gestione delle notifiche in base alle parametrizzazioni effettuate nelle prime due parti.
Si richiama la funzione `mySession.Publish(null,subAck)` che rappresenta a tutti gli effetti l'invio di Publish Request da parte del client, e che ritornerà i PublishResponse (vedi capitolo 6), da cui estrarre le informazioni relative alla subscription ed il notificationMessage.
L'oggetto subAck contiene l'id della subscription (che rappresenta a tutti gli effetti un ***Server Handle*** come utilizzato in OPC DCOM e OPC XML) cui inviare i PublishResponse e il sequence number dell'ultimo Notification Message da essa ricevuto.
Una volta ricevuta la richiesta, la subscription sul server potrà cancellare il Notification Message precedentemente inviato (e conservato al fine di eventuali ritrasmissioni richieste dal client), ed inviarne uno o più nuovi.
In seguito vengono estratti dalla response il NotificationMessage e a sua volta verrà estratto il notificationData[] che al suo interno conterrà i monitoredItems e che a loro volta conterranno i valori riportati dai nodi del server, ovvero dagli item selezionati nelle funzioni precedenti di creazione dei MonitoredItem, stampando a video i suddetti valori e il loro relativo StatusCode (Good, Uncertain, Bad) (vedi OPC UA part 4).

```
public static void activeNotification() {
      try{
                System.out.println("\n\nSUBSCRIPTION'S NOTIFICATION REPORT\n");

                new Thread(new Runnable() {
                     @Override
                     public void run() {

       SubscriptionAcknowledgement subAck = new SubscriptionAcknowledgement();

      while(true) {
            //if (LastSubId != null)
      subAck.setSubscriptionId(LastSubId);
            //if (LastSeqNumber != null)
      subAck.setSequenceNumber(LastSeqNumber);

      try {
            PublishResponse publishResponse = mySession.Publish(null, subAck);

            LastSubId = publishResponse.getSubscriptionId();
      LastSeqNumber = publishResponse.getNotificationMessage().getSequenceNumber();

                                    //
      if(publishResponse.getAvailableSequenceNumbers().length != 0){
                                                System.out.println("\n-----
----------------------------------------------");

      System.out.println("[NOTIFICATION MESSAGE] SUBSCRIPTION "+ LastSubId);

      System.out.println("Sequence number: " + LastSeqNumber);
```

```
            //                                                  for (UnsignedInteger x :
publishResponse.getAvailableSequenceNumbers()){
            //
      System.out.println("Available sequence number: " + x);
            //                                                  }
            //                                          }

      NotificationMessage nm = publishResponse.getNotificationMessage();
      ExtensionObject[] ex = nm.getNotificationData();
      for(ExtensionObject ob : ex) {
            Object change = ob.decode();
            if(change instanceof DataChangeNotification) {

      DataChangeNotification dataChange = (DataChangeNotification)change;

      MonitoredItemNotification[] mnchange = dataChange.getMonitoredItems();

      for(MonitoredItemNotification monitoredItemNotification : mnchange){

System.out.println("\n Client Handle del Monitored Item ricevuto : "+
monitoredItemNotification.getClientHandle());
                  System.out.println("    Value: " +
monitoredItemNotification.getValue().getValue());
                  System.out.println("    Server TimeStamp:" +
monitoredItemNotification.getValue().getServerTimestamp());
                  System.out.println("    Status: " +
monitoredItemNotification.getValue().getStatusCode());
                                                            }
                                                      }
                                                }  Thread.sleep(Default_PRRate);
                  } catch (ServiceFaultException e) {e.printStackTrace();
               } catch (ServiceResultException e) {e.printStackTrace();
            } catch (InterruptedException e) {e.printStackTrace();
      } catch (Exception e) {};
     }
    }
      }).start();////////////////////////////////////    Press enter to shutdown
               System.in.read();
                        ////////////////////////////////////
                    }catch(Exception e){}


}
```



Mentre si ricevono le notifiche, è possibile osservare che sul server verrà visualizzata la presenza di una sessione e di due sottoscrizioni.



10.Sviluppi Futuri

Possibili sviluppi futuri possono incentrarsi nella realizzazione di interfacce grafiche per il Client Java e riuscire a monitorare oltre alle variabili anche Oggetti, nel formato di Event Notifier e variabili aggregate. Effettuare eventuali miglioramenti per quanto riguarda l'algoritmo di browsing. Una possibilità remota sarebbe quella di integrare i vari progetti per la creazione di un progetto più ambizioso quale quello di un IDE per OPC UA Java, che raccolga tutte le funzionalità implementate negli anni dai vari corsi.