	CTA	Ref.:
	MOS Documentation	Version: Date: 28/08/2014 Page: 1/76

MOS (*M*ultipurpose *O*PCUA *Server*)

User Guide

Author	*Institution*
T. Le Flour	LAPP
Jean-Luc Panazol	LAPP
Eric Chabanne	LAPP

To be approved by	*Institution*

History		
Version	Date	Observation

Distribution	CTA internal

List of Abbreviations			
ACS	Alma Common Software		
ACTL	CTA work package 'array control'		
CTA	Cherenkov *Telescope* Array		
FSM	Final State Machine		
GUI	Graphical User interface		
MOS	Multipurpose OPCUA Server		
OPCUA	OPC Unified Architecture		
XML	Extensible Markup Language (XML)		
XSD	XML Schema Definition		

List of figures

List of tables

TABLE DES MATIÈRES

INTRODUCTION

The main objective of this document is to provide explanations on how using the tools “Multipurpose OPCUA Server” (MOS). MOS is an OPCUA Server application allowing the control and monitoring of devices. Each device can be considered as a set of data points and methods. These elements and the connection to the device will be described with a XML file. People responsible of the device, (Designer or Device Integrators) will provide the XML file describing the device(s) by using a dedicated dictionary (XSD file). This document will illustrate all the description items needed to complete a device description.

MOTIVATION

The OPCUA standard has been chosen by CTA/ACTL as the communication standard to connect the framework components (ACS) and the different devices composing the telescopes and auxiliary systems. This is the way to homogenize the monitoring and control of devices.
Briefly, the OPCUA, an evolution if the OPC standard, is an abstract software-hardware layer allowing the interfacing between a device and software components. This software standard is platform independent and based on a dedicated “Namespace” which contains all the items needed to monitor and control a device. Basically, from a software point of view, we can considerer an OPCUA server as a set of data points and methods. However, writing an OPCU code implies to have a good knowledge of all the OPCUA mechanisms and internal machinery. As people in charge of the device integration are not systematically software developers, it is quite important to offer to this category of people, tools to facilitate the OPCUA development. (. I.e. minimize the part of code to write, limit the OPCUA knowledge...).
The idea of describing the device to integrate with an XML description has been retain. This description (with a associated dictionary), provided by the device integrators, will be the input of the MOS. By reading the XML input file, the MOS will populate the OPCUA namespace with a coherent naming convention and organization. Regarding to this, the part of code to write is consequently reduced even if, in some case, the integrators should have to write dedicated code (as plugins described later in the document) to be integrated as a part of the MOS. This document will not describe the internal mechanisms of the OPCUA standard.

HOW TO READ THIS DOCUMENT

This document can be read in two ways.

- A quick reading to have a global view of the functionalities offered by the MOS. This document is divided in many chapters and paragraphs more often associated to specific part. At the beginning of each chapter, a short description of the functionality is given.
- A deeper reading to learn precisely about a functionality. Each chapter is mostly composed of a table describing in detail the different keywords associated to functionality. This is a mean to understand how to use the description of a MOS.

Often, an example is associated to the chapter/paragraph in order to illustrate how the current functionality is used. You can find all example files in your distribution of MOS in the folder example/0_tutorial/

M.O.S.

GENERAL MOS DESCRIPTION

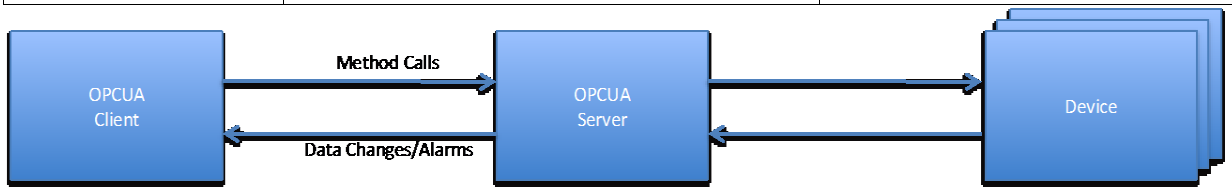


Figure 1 : general view

The schema above illustrates the organization and the connection between a OPCUA client (. i.e. ACS component) and the device. We can distinguish two types of devices: the hardware one and the logical one. The "hardware device" is a physical device, which can be connected by software with a dedicated protocol. (.i.e. Weather station) The "Logical Device " can be considered as a piece of software only, not associated to an hardware part, which can also be connected with a dedicated protocol. These two types of devices can be controlled and monitored by using the same approach.
The OPCUA server connects the device by using the defined protocol, collects and analyses the data frame coming from the device. In the same way, the OPCUA server can emit data to the device to perform the control and the configuration. All theses data exchanges are used to populate all the data points characterizing the devices. The monitoring, control, configuration and communication can be described by using a XML. Based on a XML dictionary (XSD) defining the keywords and the syntax, it allows validating the XML file. During the launching stage of the OPCUA server, the XML file is first analysed accordingly to the dictionary. Then, the XML content is used to establish the communication with the device and to populate the OPCUA namespace with all the described data points, methods, …

The XML description brings up also a coherent naming convention of all the elements (data points, methods...) in the OPCUA name space, respectively to the element hierarchy used to describe a device organization.

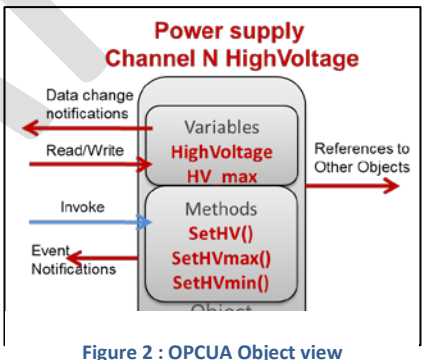


Figure 2 : OPCUA Object view

More generally, the implementation and deployment of a MOS follow the following phases (see diagram below): Production of an ICD (Interface Control Document), XML description of the concerned device based on the ICD, MOS deployment. (XML validation, OPCUA namespace initialization, device communication setup, data exchanges and analysis...)

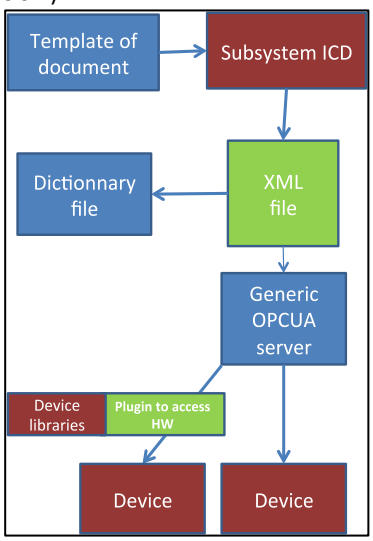


Figure 3 MOS implementation and deployment

MAIN MOS ACTORS

The UML schema above illustrates the different actors using a MOS and their associated actions.

"The dictionary manager: Each description is done accordingly to the dictionary. This is the way to insure the syntax and the XML structure. This actor is in charge of the dictionary evolution (adding or removing keywords, modification of the organization...)

"The designer or integrator": This is the person responsible of the device conception or integration. To realize the device integration with an OPCUA server (MOS) , he has to provide the XML file corresponding to the description of the device to be integrated. (Ex.: the description of all the registers characterizing an electronic board...).

"The integrator": This is the person or logical component responsible of launching /stopping a MOS :

"The final user" (human of software): This is the client part, which is going to use the MOS. The main actions are:

- Connection to the MOS
- Subscribe to one or more data points
- Read value of one or more data points
- Write value of one or more data points
- Disconnection

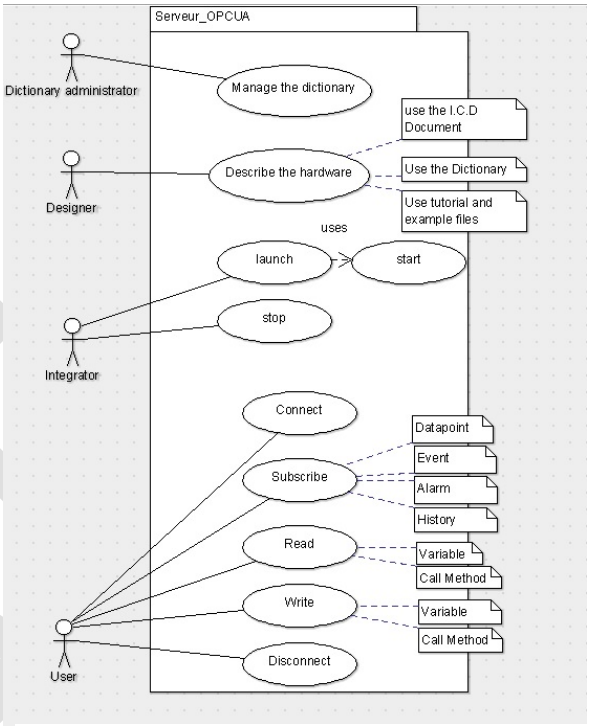


Figure 4 : UML actor view

DEVICE DESCRIPTION

In this paragraph, we will consider the case of the "Designer/Integrator" actor using a MOS to integrate his device. As mentioned above, the first step is to provide a XML file of the device description. This description, as explained above, is build accordingly to a dictionary. The following parts of this document will describe the keywords and the global organization.

1. MODEL ROOT

The entry point of the XML description.

Keyword	Type	nb	Description
OPCUA			Starting point of the device description
Name	Mandatory	1	Name of the OPCUA server. This name will be used to build the OPCUA name space and will be the root part of the name space hierarchy.
FileName	Optional	N	Used if many others XML files are needed.

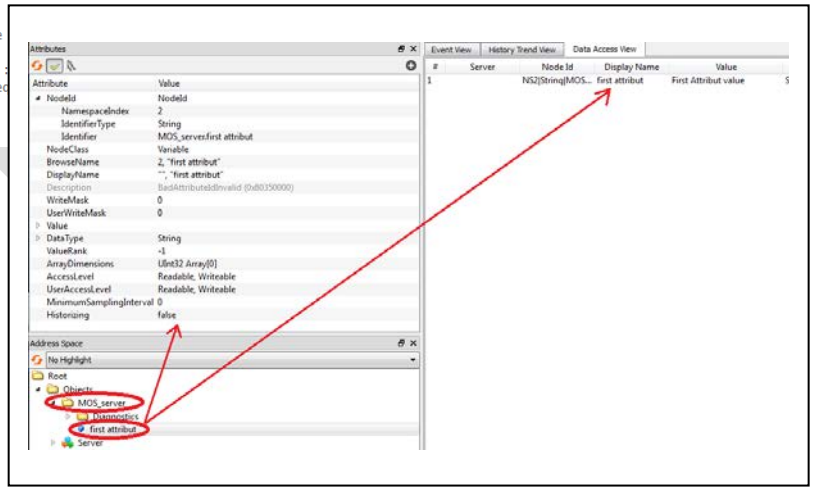
Attribut	Optional	N	Definition of general information about the server itself See chapter on attribut
ServerPort	Optional		Value of the OPCUA server port. **This element is not use in the C++ environment**. In C++ the server port number is already define in the server config XML file. This file defines all internal elements of the server. By default, we use the file ../properties/ServerConfig.xml
SimpleDevice CompoundDevice	Optional	N	Definition of one or many devices composing the OPCUA server. See below for the Simple Device and Compound Device description.

Table 1 : XML root keyword description table

Description example of this keyword:

1. Example n°1 : a very simple example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/0_declareAttribut.xml *** -->
<!-- **** description : the first simple application                              *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                     *** -->
<!-- **** date        : 25/08/14                                                  *** -->
<!-- ********************************************************************************** -->


<!-- ************************************************************** -->
<!-- **** This file start a OPCUA server call MOS_server   -->
<!-- **** and add a attribut element call "first attribut" -->
<!-- **** with the value "First attribut value"  -->
<!-- ************************************************************** -->
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<!-- this tag <OPCUA> is absolutly require -->

  <Name>MOS_server</Name>   <!-- Require : Name
  <Attribut>
    <Name>first attribut</Name>   <!-- Require :
    <Value>First Attribut value</Value> <!-- Rec
  </Attribut>
</OPCUA>

```



2. Example n°2: Integrate several xml files

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/9_file1.xml                          *** -->
<!-- **** description : integrate several xml files                                            *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                  *** -->
<!-- **** date        : 01/09/14                                                               *** -->
<!-- ********************************************************************************************** -->


<!-- ********************************************************** -->
<!-- **** This file is used with the example 9_declare_Multi_FichierXml.xml -->
<!-- ****    -->
<!-- **** This file start a OPCUA server call MOS_server    -->
<!-- **** and add 1 attribut and 1 simpleDevice     -->
<!-- ********************************************************** -->

<OPCUA  xmlns:xsi="http:/
        <Name>MOS_Server<
        <Attribut>
                <Name>fil
                <Value>Fi
        </Attribut>
        <SimpleDevice>
                <Name>fir
        </SimpleDevice>

</OPCUA>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/9_declare_Multi_FichierXml.xml   *** -->
<!-- **** description : integrate several xml files                                            *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                  *** -->
<!-- **** date        : 01/09/14                                                               *** -->
<!-- ********************************************************************************************** -->


<!-- ********************************************************** -->
<!-- **** This file start a OPCUA server call MOS_server    -->
<!-- **** and add severals  xml file     -->
<!-- **** for example each xml file describe one functionality -->
<!-- **** This main xml file integrate all the functionality defined in the sub-files -->
<!-- ********************************************************** -->

<OPCUA  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="dict_cta.xsd">
   <Name>General_MOS_Server</Name>
   <FileName>9_file1.xml</FileName>
   <FileName>9_file2.xml</FileName>
   <FileName>9_file3.xml</FileName>
   <FileName>9_file4.xml</FileName>
   <FileName>9_file5.xml</FileName>
</OPCUA>
```

1.1 THE “ATTRIBUT”

You can define some attribute at different levels of your description.

- OPCUA level
- CompoundDevice level
- SimpleDevice level
- CompoundDatapoint level
- SimpleDatapoint level

The entry point of the XML description.

Keyword	Type	nb	Description
Attribut			Starting point of the device description
Name	Mandatory	1	Name of the attribute. This name will be used to build the OPCUA name space and will be the root part of the name space hierarchy.
Value	Mandatory	1	The value of the attribute (string value).

Table 2 : Attribut description table

2. DEVICE DESCRIPTION

A device is considered as a set of data points, methods and controls. The device description has to be provided in that way. Furthermore, the device control has to be performed with methods instead of using state variable as in PLC. Effectively, in a PLC, modifying the value of a state variable can trigger an action. This implies to have correctly initialized all the "input" variables needed by the corresponding action. In MOS, method calls will be used instead. This means that, during the MOS description phase, the device control has to done by describing methods and all the associated parameters (input and output).
Depending on its complexity, its dependency to others components or its logical composition, a device can be differently described. Schematically, a device can be represented with the UML diagram above. For example, if we consider the MOS as a device assembly, the "CompoundDevice" keyword will be used. If the MOS has just to control a device, the "SimpleDevice" keyword will be used.

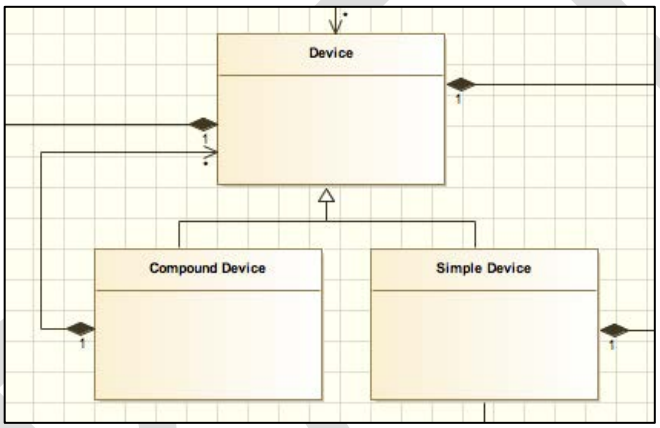


Figure 5 : UML Schema for device

2.1 THE "COMPOUND DEVICE"

Keyword used in the context of the OPCUA keyword.

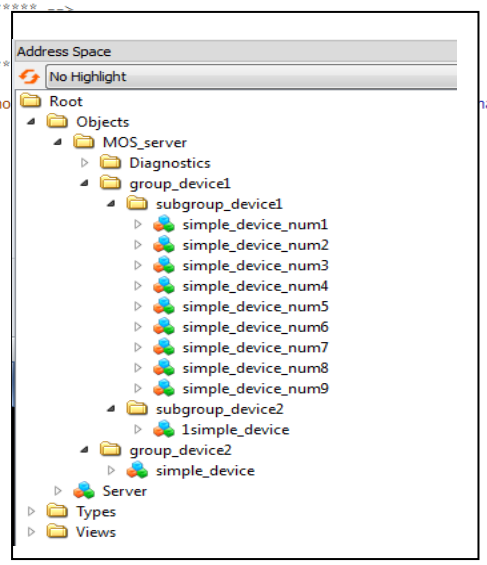
Keyword	Type	nb	Description
CompoundDevice			Used to describe a device assembly
Name	Mandatory	1	Name of the device assembly. This name will be a "directory" name in the OPCUA name space and respectively to the XML hierarchy
Multiplicity	Optional	1	Used if the corresponding device occurs many time in the assembly. As many elements will be created in the OPCUA name space as the value given for the "multiplicity " keyword.
Interface	Optional	1	Used to define the type of interface used by the device. (If it exists) . Example: TCP/ COM/ UDP etc…. See chapter on communication
DataFrameStructureRef	Optional	1	Reference to the XML filename containing the data frame structure (if needed). This file will be then used to analyse the data coming from the device.

HardwareConfig	Optional	1	Used to define the parameters needed by the selected interface. Example: TCP/IP hostname, port... See chapter on communication
Attribut	Optional	N	Used to define general device information
SimpleDevice CompoundDevice	Optional	N	Used to define all the devices composing the assembly. This assembly can be composed of simple or compound devices.
SimpleDatapoint	Optional	N	

Table 3 : CompoundDevice description table

Example of this keyword usage

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- ********************************************************************************** -->
  <!-- **** file : ../example/0_Tutorial/0_firstApplication/7_declare_compound_Device.xml *** -->
  <!-- **** description : the compound device example                                   *** -->
  <!-- **** author      : panazol@lapp.in2p3.fr                                         *** -->
  <!-- **** date        : 25/08/14                                                      *** -->
  <!-- ********************************************************************************** -->


  <!-- ************************************************************ -->
  <!-- **** This file start a OPCUA server call MOS_server    --> 
  <!-- **** and add a lot of compound device  -->
  <!-- ************************************************************

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:no                                    nazol/xsd/v1_0_0/dict_cta.xsd
        <Name>MOS_server</Name>
        <CompoundDevice>
                <Name>group_device1</Name>
                <CompoundDevice>
                        <Name>subgroup_device1</Name>
                        <SimpleDevice>
                                <Name>simple_device_num</Name>
                                <Multiplicity>9</Multiplicity>
                        </SimpleDevice>
                </CompoundDevice>
                <CompoundDevice>
                        <Name>subgroup_device2</Name>
                       <Multiplicity>1</Multiplicity>
                        <SimpleDevice>
                                <Name>1simple_device</Name>
                        </SimpleDevice>
                </CompoundDevice>
        </CompoundDevice>

        <CompoundDevice>
                <Name>group_device2</Name>
                <SimpleDevice>
                        <Name>simple_device</Name>
                </SimpleDevice>
      </CompoundDevice>
</OPCUA>
```



2.2 THE “SIMPLE DEVICE” KEYWORD

This keyword is used in the “OPCUA” or “CompoundDevice” keyword context.

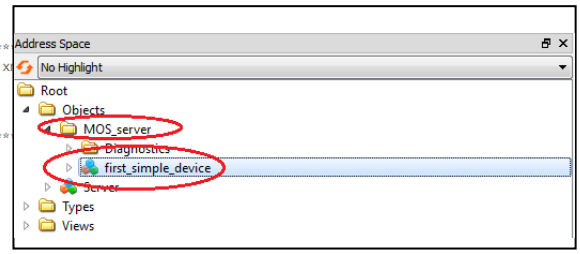
Keyword	Type	nb	Description
SimpleDevice			Used to describe a hardware or logical device.

			Example: A weather station, a power supply, a piece of sofware, DAQ etc.
Name	Mandatory	1	Used to define the name of the device. This name is used to identify the device in the OPCUA name space. (Represented as a directory) accordingly to the XML hierarchy.
Multiplicity	Optional	1	Used if the corresponding device occurs many time in the assembly. As many elements will be created in the OPCUA name space as the value given for the “multiplicity “keyword”
Attribut	Optional	N	Used to define general device information
DataFrameStructureRef	Optional	1	Reference to the XML filename containing the data frame structure (if needed). This file will be then used to analyse the data coming from the device. See chapter on Data format
DataIdDescriptorRef	Optional	1	Reference to an XML file describing data point associated to data frame structure. See chapter on: « Analysis of frame and distribution»
Interface	Optional	1	Many hardware interfaces are predefined. This tag can be used to define on of these interface used by the device. Example: TCP/ SERIAL/ UDP etc.… See chapter on “communication”
Plugins	Optional		Used to describe the plugin of the data processing coming from or to a device. See chapter Plugins.
Instruction_set	Optional	1	Tag used for defining a set of instruction associated to the device. A set of instruction is composed of basic instructions. See chapter on “Device instruction”.
HardwareConfig	Optional	1	Used to define the parameters expected by the selected interfaces. Example: address and port number for TCP/IP See chapter on “communication”
FSM	Optional	1	Each device is associated to a finite state machine. By using this keyword, it is possible to describe to different method calls associated to each state of the FSM. See chapter on « Finite State Machine »
SimpleDatapoint CompoundDatapoint	Optional	N	Used to define all the datapoints composing the assembly. This assembly can be composed of simple or compound datapoints.
Method	Optional	N	It’s possible to add methods to perform the control of the device . Example: open() et read(). See chapter on “Method”.

Table 4 : “SimpleDevice” keyword description table

Example of “SimpleDevice” keyword usage

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************
<!-- **** file : ../example/0_Tutorial/0_firstApplication/1_declareDevice.x
<!-- **** description : the first simple device
<!-- **** author      : panazol@lapp.in2p3.fr
<!-- **** date        : 25/08/14
<!-- ********************************************************************************


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server    -->
<!-- **** and add a simple empty device call "first_simple_device"-->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<Name>MOS_server</Name>
        <SimpleDevice>
                <Name>first_simple_device</Name>
        </SimpleDevice>
</OPCUA>

```



3. DATA POINTS AND METHODS

As mentioned above, a device is logically equivalent to a set of data points (used for the monitoring and control) and a set of methods (to manage the device control and configuration). A data point can be described with the keywords: “SimpleDataPoint” or “CompoundDataPoint”. Schematically, a data point can be represented with following the UML diagram.

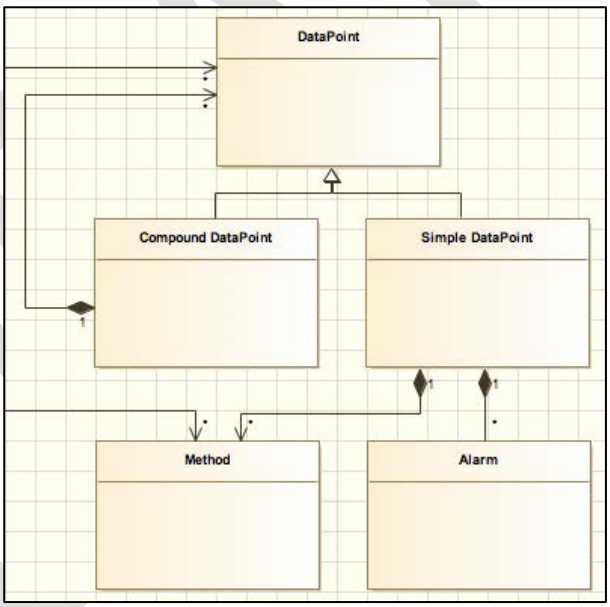


Figure 6 : UML Schema for Data Points

By considering together: Devices, Data Points and Methods, they can be connected as shown in the UML diagram below. The implementation of the MOS server and the MOS dictionary are implemented and organized accordingly to this schema. Some entities have not been yet described (Alarm, Instruction, Plugin, Plugin Interface...) but will be later on in this document.

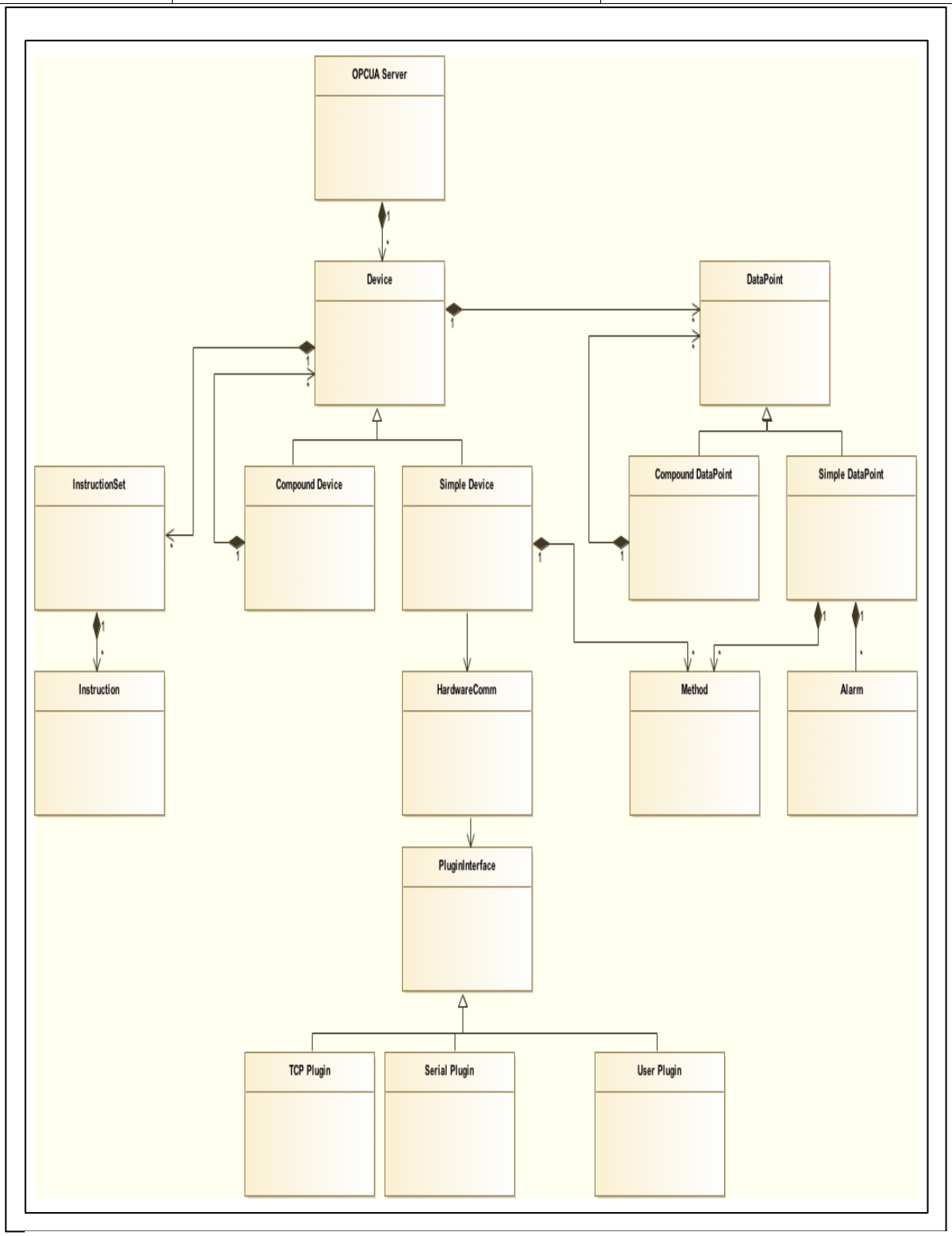


Figure 7 : UML general schema

3.1 THE COMPOUND DATAPOINT

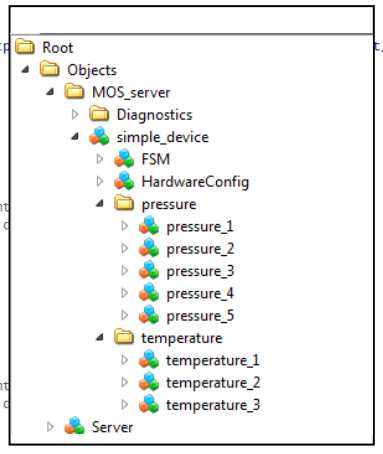
A «Compound Data Point » is used to describe a logical collection of single data points. This way of representing the data characterizing a device can be used to organize the OPCUA name space accordingly to hierarchical criteria.

Keyword	Type	nb	Description
CompoundDatapoint			Used to describe a collection of single data points
Name	Mandatory	1	Name of the collection. This name will be used to identify the collection in the OPCUA name space and will be represented as a directory.
Multiplicity	Optional	1	Used if the data point occurs many time. It will be created in the OPCUA name space as many data points as the value given for the “multiplicity “keyword
Attribut	Optional	N	Used to define general device information
CompoundDatapoint	Optional	N	Used to describe the collection. It can be composed indifferently of simple or compound data points.
SimpleDatapoint			

Table 5 : "CompoundDataPoint" description table.

Example of “CompoundDatapoint” keyword usage

```
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/6_declare_compound_Datapoint.xml   *** -->
<!-- **** description : declare datapoints and group then by compoundatapoint  tag          *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 25/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ********************************************************** -->
<!-- **** This file start a OPCUA server call MOS_server   --> 
<!-- **** and add a simple device call "simple_device"--> 
<!-- **** declare 2 group of datapoint on it  --> 
<!-- **** one group for temperature sensors  call "temperature"--> 
<!-- ****   declare 3 temperature sensors datapoints  (float value)--> 
<!-- **** one group for pressure sensors   call "pressure"--> 
<!-- ****   declare 5 pressure sensors datapoints  (int32 value)--> 
<!-- **** for each datapoint we want historizing the value --> 
<!-- ********************************************************** -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http                                                t_cta.xsd">
<Name>MOS_server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <CompoundDatapoint>
                        <Name>temperature</Name>
                        <SimpleDatapoint>
                                <Name>temperature_</Name>
                                <Multiplicity>3</Multiplicity>
                                <Type>float</Type>              <!-- Require  : value(string,int
                                <Historizing>1</Historizing>    <!-- Optional : value(0 , 1) : d
                        </SimpleDatapoint>
                </CompoundDatapoint>

                <CompoundDatapoint>
                        <Name>pressure</Name>
                        <SimpleDatapoint>
                                <Name>pressure_</Name>
                                <Multiplicity>5</Multiplicity>
                                <Type>int32</Type>              <!-- Require  : value(string,int
                                <Historizing>1</Historizing>    <!-- Optional : value(0 , 1) : d
                        </SimpleDatapoint>
                </CompoundDatapoint>
        </SimpleDevice>
</OPCUA>
```



3.2 THE "SIMPLE DATAPOINT"

A simple data point is used to describe precisely a device characteristic. (Associated command, data type...). The table below gives a description of the most frequently used keywords.

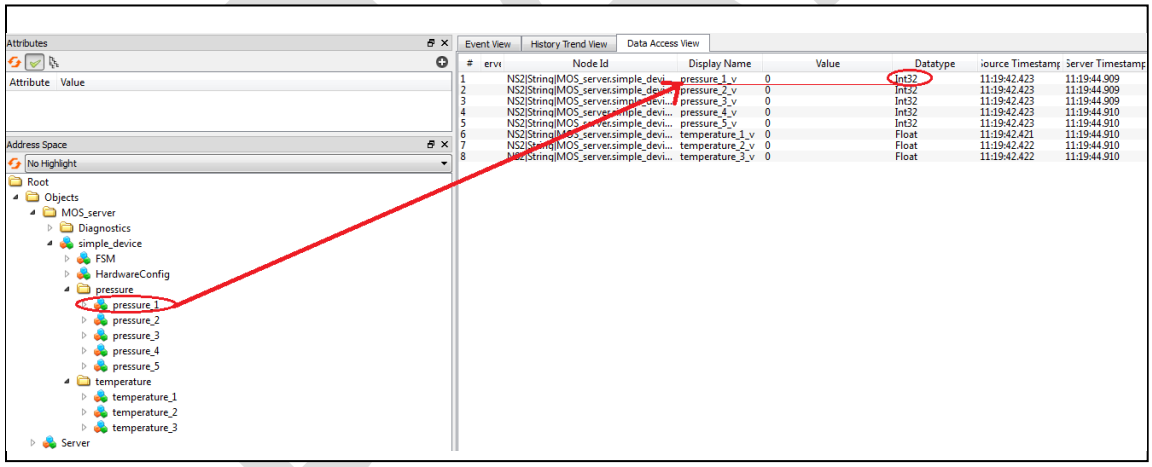
Keyword	Type	nb	Description
SimpleDatapoint			Used for describing a simple data point
Name	Mandatory	1	Name of the data point. This name is used to identify the data point in the OPCUA name space. This object will be created accordingly to the hierarchy of the XML description.
Multiplicity	Optional	1	Used if the data point occurs many time. It will be created in the OPCUA name space as many data points as the value given for the "multiplicity "keyword
Type	Mandatory	1	Used to define the data type of the data point (string, int, float,) for the created object in the OPCUA name space. Possible types are: • string • float • double • int8 • int32 • int16 • int64 • boolean
ArraySize	Optional	1	Used if the data point value needs to be represented with an array. The given value defines the array size. By default, the value is 0 meaning that the data is not an array.
Attribut	Optional	N	Used to define general device information
Id	Optional	1	Associates the id received from a frame with this datapoint. See chapter "analysis of frames and distribution into several datapoints"
AccesLevel	Optional	1	Used to set the data point access rights. 1 = Readable 2 = Writeable 3 = Readable, Writeable By default the value is 3
DefaultValue	Optional	1	Used to initialize the data point value to a default value Possible if the AccesLevel=2 or 3
ElementArray	Optional		When the data point is defined as an array, this tag allows defining a default value in some elements of this array. See: "ElementArray" chapter.
Description	Optional	1	Used to provide to clients a short description of the data point.
Historizing	Optional	1	Used to set the data Historization of a data point. By default, the Historization is not enabled. Historization means that the OPCUA server can store a set of value for the data point for a

			certain period of time. The clients can then ask for this set of time stamped values. (By default, the buffer size if about 1000 values). 0 = no historization 1 = Historization By default Historizing=0
Alarm	Optional	1	Used to set an alarm. Depending on the alarm description, the data point will emit an alarm when a value is over the predefined values. All the clients having subscribed to the alarm will be notified. The threshold and messages are described with the Alarm keyword. See “Alarm” chapter.
AlarmMethod	Optional	1	Used to override the predefined alarm methods such as « clearAlarm » and « troubleShooting ». This is associated to the “Alarm” keyword. See “Alarm” chapter.
MonitoringRate	Optional	1	Used to define the data point value refreshing frequency. See chapter: « AUTOMATIC DATA refreshing (Push/Pull)»
Method	Optional	N	Basically, a data point is associated to a variable in the OPCUA name space. Anyway, it is possible to add methods to perform the control. Example: get() et set() to read and write the data point value. See chapter on “Method”.

Table 6 : "Simple Data Point" description table

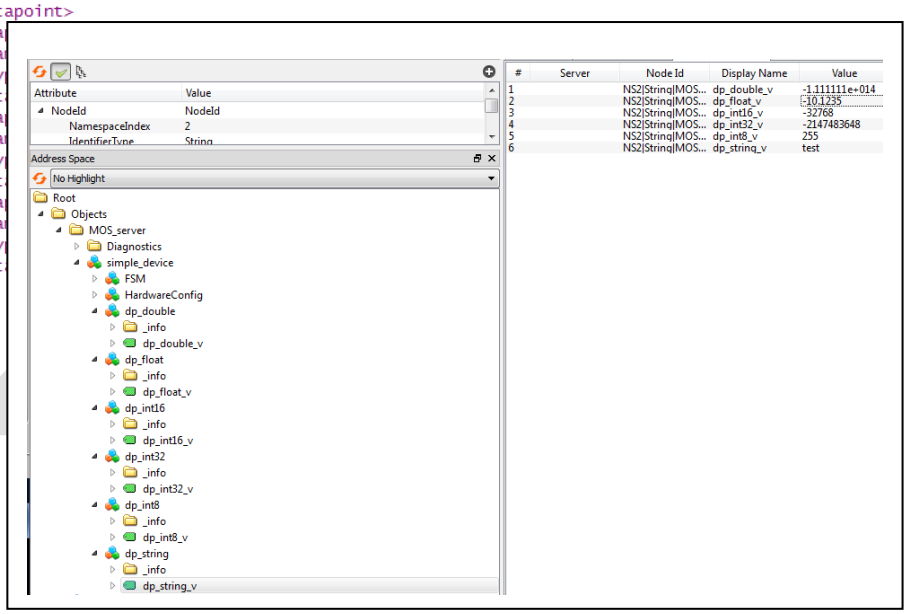
Example of “SimpleDataPoint” keyword usage.

```
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/6_declare_compound_Datapoint.xml   *** -->
<!-- **** description : declare datapoints and group then by compoundatapoint  tag           *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 25/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   -->
<!-- **** and add a simple device call "simple_device"-->
<!-- **** declare 2 group of datapoint on it  -->
<!-- **** one group for temperature sensors  call "temperature"-->
<!-- ****   declare 3 temperature sensors datapoints  (float value)-->
<!-- **** one group for pressure sensors   call "pressure"-->
<!-- ****   declare 5 pressure sensors datapoints  (int32 value)-->
<!-- **** for each datapoint we want historizing the value -->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<Name>MOS_server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <CompoundDatapoint>
                        <Name>temperature</Name>
                        <SimpleDatapoint>
                                <Name>temperature_</Name>
                                <Multiplicity>3</Multiplicity>
                                <Type>float</Type>              <!-- Require  : value(string,int32 ,float,bool) -->
                                <Historizing>1</Historizing>    <!-- Optional : value(0 , 1) : default=0-->
                        </SimpleDatapoint>
                </CompoundDatapoint>

                <CompoundDatapoint>
                        <Name>pressure</Name>
                        <SimpleDatapoint>
                                <Name>pressure_</Name>
                                <Multiplicity>5</Multiplicity>
                                <Type>int32</Type>              <!-- Require  : value(string,int32 ,float,bool) -->
                                <Historizing>1</Historizing>    <!-- Optional : value(0 , 1) : default=0-->
                        </SimpleDatapoint>
                </CompoundDatapoint>
        </SimpleDevice>
</OPCUA>
```



Example N°2 : different type of datapoint. : 3_declare_several_Datapoints.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/3_declare_several_Datapoints.xml   *** -->
<!-- **** description : declare several datapoint with different type                        *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 02/09/14                                                             *** -->
<!-- ********************************************************************************************** -->

<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   --> 
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare datapoints with different type  -->
<!-- ****   string, int8, int16, int32, float, double  -->
<!-- ************************************************************ -->
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<Name>MOS_server</Name>
                <SimpleDevice>
                        <Name>simple_device</Name>

                        <SimpleDatapoint>
                                <AccesLevel>3</AccesLevel>
                                <Name>dp_string</Name>
                                <Type>string</Type>
                                <ArraySize>10</ArraySize>
                        </SimpleDatapoint>
                                <ArraySize>10</ArraySize>
                        </SimpleDatapoint>
                        <SimpleDatapoint>
                                <AccesLevel>1</AccesLevel>
                                <Name>dp_int8</Name>
                                <Type>int8</Type>
                                <DefaultValue>test</DefaultValue>
                        </SimpleDatapoint>
                        <SimpleDatapoint>
                                <AccesLevel>3</AccesLevel>
                                <Name>dp_int16</Name>
                                <Type>int16</Type>
                                <DefaultValue>-10</DefaultValue>
                        </SimpleDatapoint>
                        <SimpleData
                                <Na
                                <Ty
                        </SimpleDat
                        <SimpleData
                                <Na
                                <Ty
                        </SimpleDat
                        <SimpleData
                                <Na
                                <Ty
                        </SimpleDat
                </SimpleDevice>
</OPCUA>
```



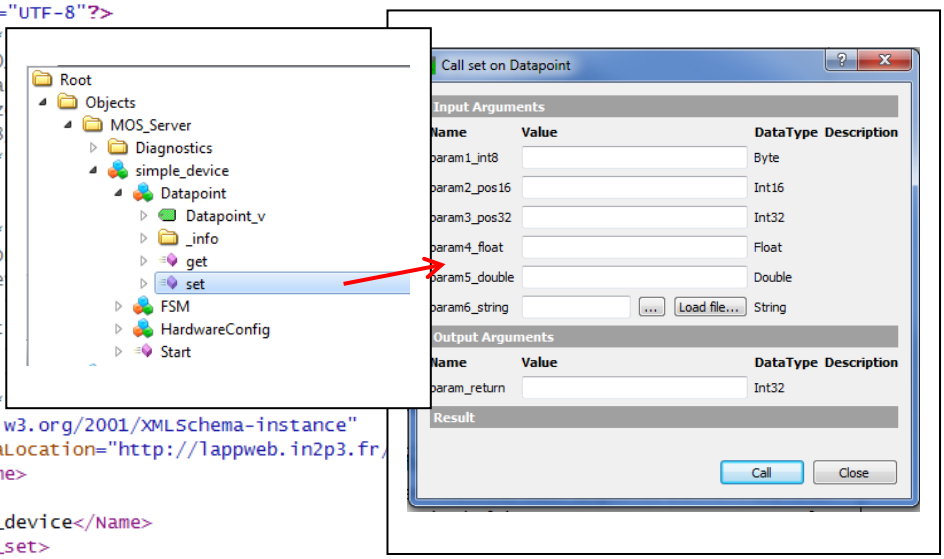
1.1 THE «METHOD»

A method can be defined in different context. It can be used for a device to perform a global action (ex. : start(),stop(), reset(), ...) but it can be also used for managing a data point. In that case, a method can be used to obtain or modify the value of the data point (ex. set(...), get(), ...).
The table below details the keyword used for a method description.

Keyword	Type	nb	Description
Method			Keyword used to start a method description.
Name	Mandatory	1	Name of the method. This name is used to identify the method in the OPCUA name space. This object will be created accordingly to the hierarchy of the XML description.
Sequence	Optional	N	Used to define a sequence of method calls in the context of the current method. Each sequence element is a reference to a method of the current OPCUA server or a reference to a method belonging to another OPCUA server. See chapter on “Sequence”
Argument	Optional	N	Used to define the parameters (input and output) of a method. See chapter on “Arguments”
DeviceInstruction	Optional	N	Used to define the device instruction to be sent to the device during the method call. The name of the “DeviceInstruction” is a mnemonic in the device instruction set defined at the device level. See chapter on « instruction_set ».
EndDelimiter	Optional	1	In some case, a command sent to a device has to be ended by a delimiter to validate the command transmission. This special character cannot be inserted in the command itself because of character interpretation. Ex.: ‘0D’ pour un « Carriage Return» Remark: It’s a hexadecimal format.
Event	Optional	N	Used to specify that a method have to send an event when it is executed. This option can be used when a server has to warn all the connected clients a method execution. Only the clients having subscribed to this event will be notified. See chapter on “Event”

Table 7 : "Method" description table

Example of “Method” keyword usage.




```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ************************
<!-- **** file : ../example/0
<!-- **** description : decla
<!-- **** author      : panaz
<!-- **** date        : 25/08
<!-- ************************


<!-- ************************
<!-- **** This file start a O
<!-- **** and add a simple de
<!-- ****   declare 1 method
<!-- **** declare 1 datapoint
<!-- ****   declare 1 method
<!-- ****   declare 1 method
<!-- ************************
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/
       <Name>MOS_Server</Name>
       <SimpleDevice>
              <Name>simple_device</Name>
              <Instruction_set>
                     <Instruction>
                            <Name>Set_instruction</Name>
                            <Cmd>myinstruction1</Cmd>
                     </Instruction>
                     <Instruction>
                            <Name>Get_instruction</Name>
                            <Cmd>myinstruction2</Cmd>
                     </Instruction>
                     <Instruction>
                            <Name>Start</Name>
                            <Cmd>start 1</Cmd>
                     </Instruction>
              </Instruction_set>

              <Method>
                     <Name>Start</Name>
                     <DeviceInstruction>
                            <Value>Start</Value>
                     </DeviceInstruction>
                     <EndDelimiter>0d</EndDelimiter>
              </Method>

              <SimpleDatapoint>
                     <Name>Datapoint</Name>
                     <Type>int32</Type>

```

```
Browse
                <Method>
                        <Name>get</Name>
                        <DeviceInstruction>
                                <Value>Get_instruction</Value>
                        </DeviceInstruction>
                </Method>
                <Method>
                        <Name>set</Name>
                        <Argument>
                                <Name>param1_int8</Name>
                                <Type>int8</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param2_pos16</Name>
                                <Type>int16</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param3_pos32</Name>
                                <Type>int32</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param4_float</Name>
                                <Type>float</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param5_double</Name>
                                <Type>double</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param6_string</Name>
                                <Type>string</Type>
                                <Access>Input</Access>
                                <DefaultValue>2</DefaultValue>
                        </Argument>
                        <Argument>
                                <Name>param_return</Name>
                                <Type>int32</Type>
                                <Access>Output</Access>
                        </Argument>
                        <DeviceInstruction>
                                <Value>Set_instruction</Value>
                        </DeviceInstruction>
                </Method>
        </SimpleDatapoint>
```

1.1.1 ARGUMENT

Usually, a method is associated to arguments. This table below describes the keyword used to define parameters of a method.

Keyword	Type	nb	Description
Argument			Keyword used to start an argument description.
Name	Mandatory	1	Name of the parameter. This name is used to identify the parameter in the OPCUA name space. This object will be created accordingly to the hierarchy of the XML description. (In that case associated to the method).

Type	Mandatory	1	Type of the argument (int, float string etc.)
Access	Optional	1	Used to define the parameter access. The 2 possible values are: “input “and “output”. By default, the value is « input ».
DefaultValue	Optional	1	Used to define the default value of the argument.
Description	Optional	1	Used to add some information about the argument and accessible to the connected clients.

Table 8 : “Argument”description table

1.1.2 SEQUENCE

A method execution can, in some case, refer to some other methods execution. The method calls have to be ordered and each method of the sequence call belongs to the current OPCUA server or to external OPCUA server data points. To call an external method, the current OPCUA server initiates a connection to the distant OPCUA server as OPCUA client. Currently, the connections are foreseen between MOS-MOS entities. To call a distant method with parameters, no parameters are described and using the default parameters values will be used.

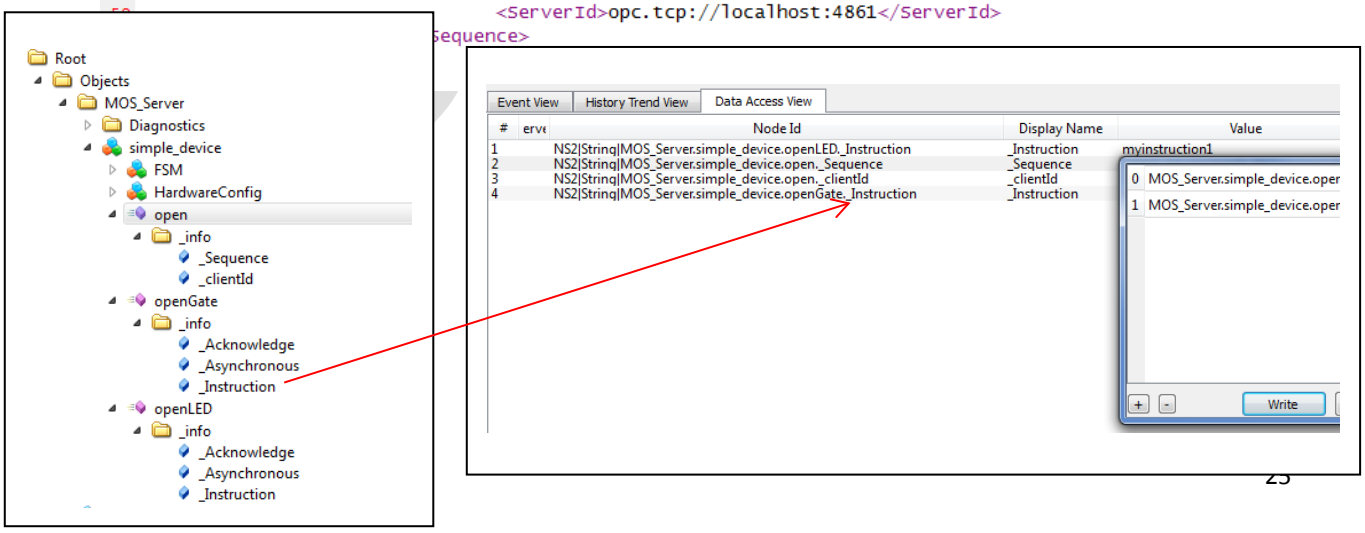
The table below describes the keyword associated to the “Sequence” tag:

Keyword	Type	nb	Description
Sequence			Used to start a sequence description
NodeId	Mandatory	1	Used to identify the data point to be accessed.
MethodId	Mandatory	1	Used to identify the method to be called in the context of the “NodeId”.
ServerId	Optional	1	Used to define the distant OPCUA server identification. Ex. : : ocp.tcp:// lappc-p303:4841

Table 9 : Table des mots clés pour la description de "Sequence"

Example of “Sequence” keyword usage.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/10_declare_Method_Sequence.xml      *** -->
<!-- **** description : declare a methods and use a sequence                                  *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                  *** -->
<!-- **** date        : 25/08/14                                                               *** -->
<!-- ********************************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server    -->
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare 3 methods in this simpledevice call "open", "openLED" "openGate" -->
<!-- **** the method "openLED" use 1 instruction -->
<!-- **** the method "openGate" use 1 instruction-->
<!-- **** the method "open" use 1 sequence call "openLED", "openGate", -->
<!-- **** and also call an external method in external OPCUA server -->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
        <Name>MOS_Server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>

                <Instruction_set>
                        <Instruction>
                                <Name>openLED_instruction</Name>
                                <Cmd>myinstruction1</Cmd>
                        </Instruction>
                        <Instruction>
                                <Name>openGate_instruction</Name>
                                <Cmd>myinstruction2</Cmd>
                        </Instruction>
                </Instruction_set>

                <Method>
                        <Name>open</Name>
                        <Sequence>
                                <NodeId>MOS_Server.simple_device</NodeId>
                                <MethodId>openLED</MethodId>
                        </Sequence>
                        <Sequence>
                                <NodeId>MOS_Server.simple_device</NodeId>
                                <MethodId>openGate</MethodId>
                        </Sequence>
                        <Sequence>
                                <NodeId>MOS_Server.other_device</NodeId>
                                <MethodId>start</MethodId>
                                <ServerId>opc.tcp://localhost:4861</ServerId>
                        </Sequence>
```



```
54                  <Method>
55                          <Name>openLED</Name>
56                          <DeviceInstruction>
57                                  <Value>openLED_instruction</Value>
58                          </DeviceInstruction>
59
60                  </Method>
61
62                  <Method>
63                          <Name>openGate</Name>
64                          <DeviceInstruction>
65                                  <Value>openGate_instruction</Value>
66                          </DeviceInstruction>
67                  </Method>
68          </SimpleDevice>
69  </OPCUA>
```

1.1.3 ALARM AND ALARM METHOD OVERRIDING

Under some condition, it should be useful to associate alarm to a data point. Each client having subscribed to this alarm will be notified when the data point generates the alarm. The notified clients will receive a set of information as the message text, severity, ...
Remark: In the current MOS version, the alarm mechanisms are only managed with integer values.

Two levels of alarms can be considered:

1. Single Alarm
2. Critical Alarm ➔ Device is put in the "Error" state thru the associated FSM.

For each alarm level, it is possible to define the conditions under which the alarm is emitted:

- High
- Low
- Equal
- Not equal

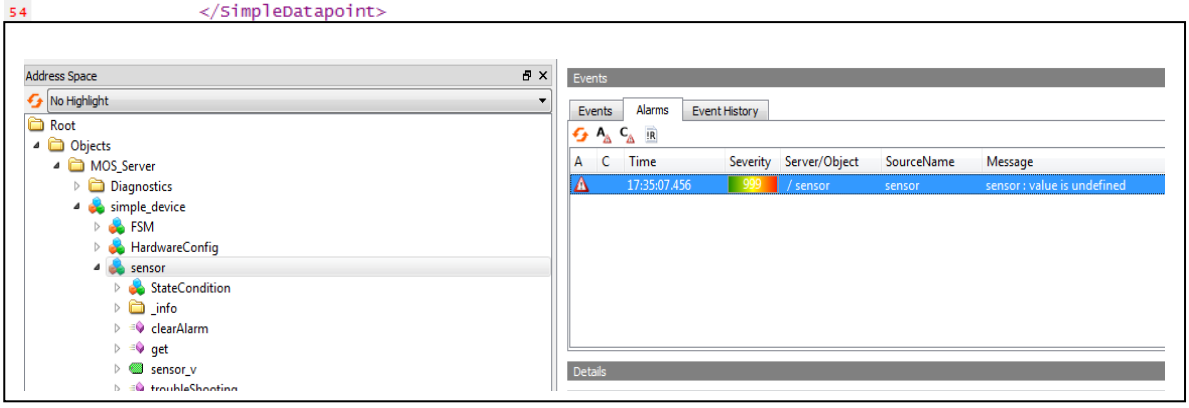
The table below describes the different keywords used to an alarm description:

Keyword	Type	nb	Description
Alarm			Used to start the alarm description
Alarm_LimitHigh	Optional	1	Used to describe the maximum threshold corresponding to an alarm emission
Alarm_LimitLow	Optional	1	Used to describe the minimum threshold corresponding to an alarm emission
Alarm_Equal	Optional	1	Used to describe an alarm emission when the data point value equals the given value.
Alarm_NotEqual	Optional	1	Used to describe an alarm emission when the data point value is different the given value.
Error_LimitLow	Optional	1	Equivalent to "Alarm_LimitLow" and the device is set in an "Error" state
Error_LimitHigh	Optional	1	Equivalent to "Alarm_HighLow" and the device is set in an "Error" state
Error_Equal	Optional	1	Equivalent to "Alarm_Equal" and the device is set in an "Error" state
Error_NotEqual	Optional	1	Equivalent to "Alarm_NotEqual" and the device is set in an "Error" state

Table 10 : "Alarm" description table

Example of "Alarm" keyword usage.

```
<!-- ********************************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/11_declareAlarm.xml                              *** -->
<!-- **** description : declare some alarm about the value of one datapoint                               *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                              *** -->
<!-- **** date        : 25/08/14                                                                           *** -->
<!-- ********************************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   --> 
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare 1 datapoint call "sensors" with the type int32 -->
<!-- **** declare several alarms with this datapoint -->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
       <Name>MOS_Server</Name>
       <SimpleDevice>
              <Name>simple_device</Name>
              <SimpleDatapoint>
                     <Name>sensor</Name>
                     <Type>int32</Type>
                     <Alarm>
                            <Alarm_limitLow>
                                   <Value>10</Value>
                                   <Message>value under the limit</Message>
                                   <Severity>100</Severity>
                                   <Enable>true</Enable>
                                   <Acknowledge>true</Acknowledge>
                            </Alarm_limitLow>
                            <Alarm_limitHigh>
                                   <Value>30</Value>
                                   <Message>value upper the limit</Message>
                                   <Severity>500</Severity>
                                   <Enable>true</Enable>
                                   <Acknowledge>true</Acknowledge>
                            </Alarm_limitHigh>
                             <Error_Equal>
                                   <Value>-1</Value>
                                   <Message>value is undefined</Message>
                                   <Severity>999</Severity>
                                   <Enable>true</Enable>
                                   <Acknowledge>true</Acknowledge>
                            </Error_Equal>
                     </Alarm>
                     <Method>
                            <Name>get</Name>
                            <DeviceInstruction>
                                   <Value>GetCurrent</Value>
                            </DeviceInstruction>
                     </Method>
              </SimpleDatapoint>
```



The table below describes for each of the alarm keywords, some additional information associated to the corresponding alarm.

Keyword	Type	nb	Description
AlarmLimitHigh ou AlarmLimitLow AlarmNotEqual AlarmEqual ErrorLimitHigh ou ErrorLimitLow ErrorNotEqual ErrorEqual			
Value	Mandatory	1	Used to set the limit value
Message	Optional	1	Used to provide the alarm message.
Severity	Optional	1	Used to set the alarm severity level
Enable	Optional	1	Used to validate or invalidate the alarm usage. It should be used to enable or not the corresponding alarm. False = validate this alarm usage. True = invalidate this alarm usage. By default Enable = true
Acknowledge	Optional	1	Allow the server to automatic acknowledgement. See bellow False = no automatic acknowledgement True = automatic acknowledgement By default Acknowledge= false

Table 11 : " LimitHigh LimitLow Equal et NotEqual " description table

Management of the alarm acknowledgment.

Automatic acknowledgement.

After an alarm is emitted and if the data point value returns to a value in the accepted data range, the OPCUA server can decide by itself to acknowledge the alarm. In that case, the tag <Acknowledge> is set to true in the alarm description.

External acknowledgment

By default, when an alarm is associated to a data point, two methods are automatically created: ***clearAlarm*** and ***troubleShooting***.

clearAlarm : This method, when called by a client, acknowledges and clears the alarm and then refreshes the data point value(By calling the get method). If the value got from the device is still out of range, an alarm is re-emitted.

troubleShooting: Before clearing the alarm, it can be important to set the device in coherent state by sending on or many instructions(.i.e. reset device, ...). By default (no overriding) this method does noth-

ing except that the “clearAlarm” does already. However, if the processing of the troubleshooting is well identified, the “clearAlarm” method can be overridden. In that case, the client having to solve the trouble will call this method with the described parameters or without parameters (the default parameters value will be used). The method overriding is done in the same way of a method description (name, parameters...)

The table below describes how to override the 2 functions clearAlarm and troubleShooting.

Keyword		Description
AlarmMethod		Used to start the overriding of the alarm method. The name associated to the tag “Name” will identify the method to override
Name	clearAlarm	Used to override the “clearAlarm” method. The following description of this method is based on the same structure than the one used to describe a method. (Parameters, ...)
	troubleShooting	Used to override the “troubleshooting” method. The following description of this method is based on the same structure than the one used to describe a method. (Parameters, ...)

Table 12 : "AlarmMethod" **description table**

Example of “AlarmMethod” keyword usage.

```
    <AlarmMethod>
      <Name>clearAlarm</Name>
      <Argument>
        <Name>ArgumentExample</Name>
        <Type>string</Type>
        <Access>Input</Access>
        <DefaultValue>default value</DefaultValue>
      </Argument>
    </AlarmMethod>

    <AlarmMethod>
      <Name>troubleShooting</Name>
      <Argument>
        <Name>ArgumentExample</Name>
        <Type>string</Type>
        <Access>Input</Access>
        <DefaultValue>1</DefaultValue>
      </Argument>

    <DeviceInstruction>
      <Value>Reset</Value>
    </DeviceInstruction>
   </AlarmMethod>
</SimpleDatapoint>
```

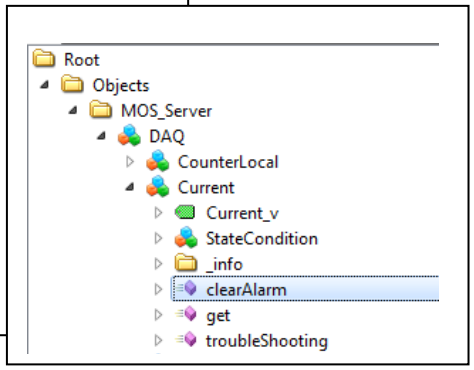


Figure 8 Visualisation de l'espace de nom OPCUA correspondant a la description XML ci-contre

1.1.4 EVENT

During a method call, it is possible to emit an event. This allows to warn all the connected client that the concerned method has been called and executed. This can be used when we want to inform by notification all the connected clients monitoring the device of all the different actions occurring on a device. This description is done in the method description context.

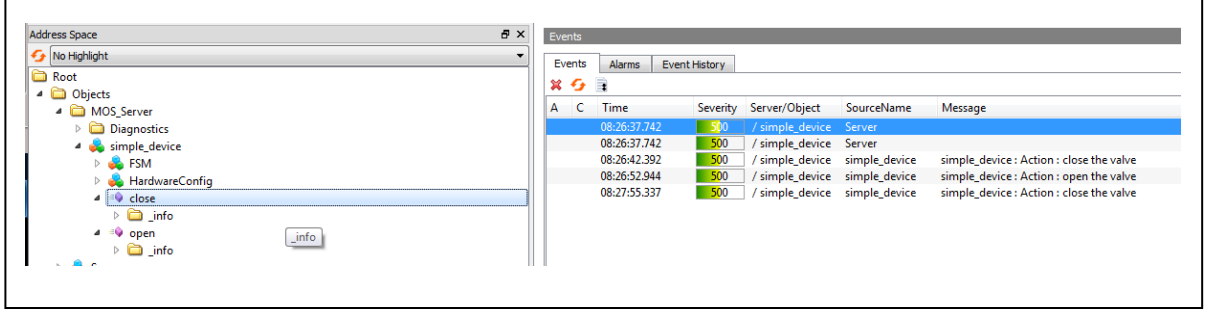
The table below describes the “Event” description.

Keyword	Type	nb	Description
Event			Used to start a event description
Message	Mandatory	1	Used to define the message sent to the client when the event is sent.

Table 13 : "Event" description table.

Example of “Event” keyword usage.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/12_declareEventMethod.xml          *** -->
<!-- **** description : declare 1 event when a method is called                              *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server    --> 
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare 1 method call "open" -->
<!-- ****       declare 1 event for this method with the message "open the valve" -->
<!-- ****   declare 1 method call "close" -->
<!-- ****       declare 1 event for this method with the message "close the valve" -->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
       <Name>MOS_Server</Name>
       <SimpleDevice>
              <Name>simple_device</Name>
              <Method>
                     <Name>open</Name>
                     <Event>
                            <Message>open the valve</Message>
                     </Event>
              </Method>
              <Method>
                     <Name>close</Name>
                     <Event>
                            <Message>close the valve</Message>
                     </Event>
              </Method>
       </SimpleDevice>
</OPCUA>
```



1.1.5 ELEMENT ARRAY

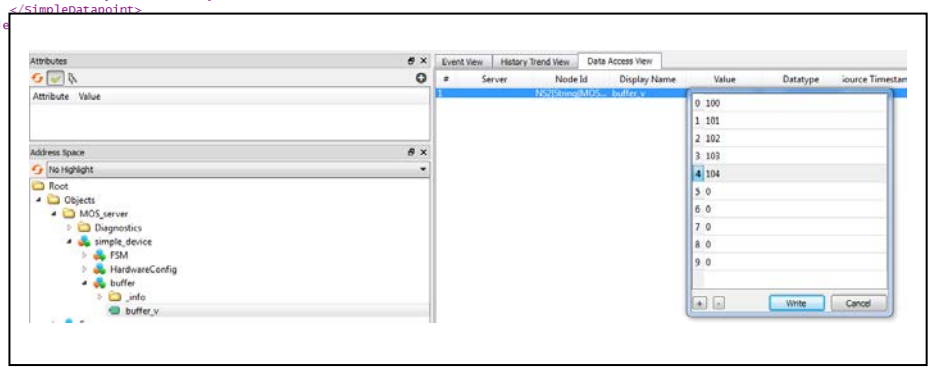
A data point is defined with a type. It can be define as a simple type (int, string,float, ...) or as an array of simple type.
If you declare the data point as an array, you can define default values for each Elements of this array by using the tag "ElementArray".

Keyword	Type	nb	Description
ElementArray			
Pos	Mandatory	1	
Value	Mandatory	1	

Table 14 : "ElementArray" description table.

Example of “ElementArray” keyword usage.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/13_declare_ArrayTypeDatapoint.xml  *** -->
<!-- **** description : declare datapoints and group then by compoundatapoint  tag           *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   --> 
<!-- **** and add a simple device call "simple_device"-->
<!-- **** declare 1 datapoint call "buffer"  -->
<!-- **** declare this datapoint as an array of int32 -->
<!-- **** declare the size of this array 10 elements -->
<!-- **** declare the default value for some element of this array -->
<!-- ****    [0] = 100, [1] = 101, [2] = 102, [3] = 103, [4] = 104 -->
<!-- ************************************************************** -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<Name>MOS_server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <SimpleDatapoint>
                        <Name>buffer</Name>
                        <Type>int32</Type>
                        <ElementArray>
                                <Pos>0</Pos>
                                <Value>100</Value>
                        </ElementArray>
                        <ElementArray>
                                <Pos>1</Pos>
                                <Value>101</Value>
                        </ElementArray>
                        <ElementArray>
                                <Pos>2</Pos>
                                <Value>102</Value>
                        </ElementArray>
                        <ElementArray>
                                <Pos>3</Pos>
                                <Value>103</Value>
                        </ElementArray>
                        <ElementArray>
                                <Pos>4</Pos>
                                <Value>104</Value>
                        </ElementArray>
                        <ArraySize>10</ArraySize>
                </SimpleDatapoint>
        </Simple
</OPCUA>
```



2. INSTRUCTIONS:

During a method call, the OCPUA server can send instructions to the device via the hardware interface or the associated plugin. All the device instructions are grouped in an instruction set ("Instruction_set"). This set is used to make the association between an instruction's mnemonic and the real command to be sent to the device. In the "Method" description, the "device_intsruction" refers to the mnemonic of the instruction set.

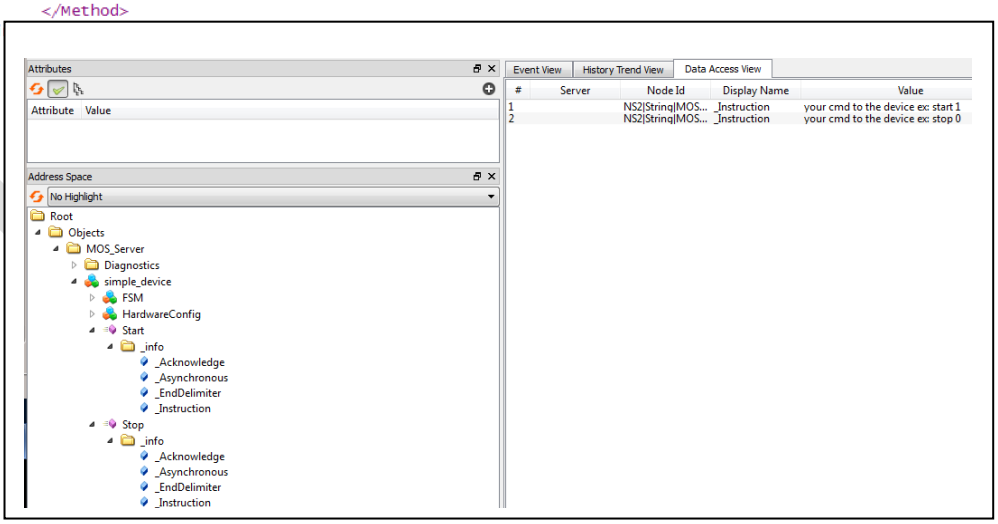
The table below describes the instruction set and its content description.

Instruction_set			Used to start the instruction set description.
Instruction			Used to define a instruction in the current set
Name	Mandatory	1	Used to create the instruction mnemonic.
Cmd	Mandatory	1	Used to set the real command (string) to be sent to the device or the Plugin.
Acknowledge	Optional	1	Used to precise if after the command sending, a command acknowledgment is sent by the device. In that case, the OPCUA server has to wait for this information. A timeout is then initialized in order to wait the device answer for a well-defined amount of time. By default, no command acknowledgement is expected.
Asynchronous	Optional	1	Used to describe how the acknowledgement is managed. When asynchronous, the server does not block to wait the device acknowledgment but organize the expected device answer with a pile of expected command. An error is sent to the client in case of timeout exceeded. By default, the asynchronous mode is not managed. To use it correctly the "tcp_OPCUA_connection" library is recommended.

Table 15 : "Instruction_set" and "Instruction" description table

Example of "Instruction_set" and "Device_Instruction" keywords usage.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/14_declare_Method_Instruction.xm    *** -->
<!-- **** description : declare instruction for methods                                        *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                   *** -->
<!-- **** date        : 28/08/14                                                                *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server    --> 
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare a list of instructions to do (in the hardware or plugin : see later) -->
<!-- ****   declare 1 method in this simpledevice call "start" -->
<!-- ****     associate the good instruction to this method -->
<!-- ****   declare 1 method in this simpledevice call "stop" -->
<!-- ****     associate the good instruction to this method -->
<!-- ************************************************************ -->
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
        <Name>MOS_Server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <Instruction_set>
                        <Instruction>
                                <Name>Start</Name>
                                <Cmd>your cmd to the device ex : start 1</Cmd>
                        </Instruction>
                        <Instruction>
                                <Name>Stop</Name>
                                <Cmd>your cmd to the device ex: stop 0</Cmd>
                        </Instruction>
                </Instruction_set>

                <Method>
                        <Name>Start</Name>
                        <DeviceInstruction>
                                <Value>Start</Value>
                        </DeviceInstruction>
                        <EndDelimiter>0d</EndDelimiter>
                </Method>
                <Method>
                        <Name>Stop</Name>
                        <DeviceInstruction>
                                <Value>Stop</Value>
                        </DeviceInstruction>
                        <EndDelimiter>0d</EndDelimiter>
                </Method>
        </Si
</OPCUA>
```



3. DEVICE COMMUNICATION

As a MOS server has to be connected to the device to be controlled, it is useful to describe the way of connecting and communicating with the device (ex: Ethernet TCP/UDP, USB, RS232...). If we consider, for example, a device connected over Ethernet with the TCP protocol, it is essential to describe the different parameters characterizing this connexion: IP address IP, port number, ... (this is done the "Hardware_Config section). However, in some cases, it is also possible to delegate the connection management to an external library (plugin). This can be useful when the communication is complex or based on dedicated libraries. (ex. snmp protocol, xbee...). By defining a plugin or an hardware interface, the methods "init" and " close" are defined automatically. These methods can be overridden in the "HardwareConfig" section. The overriding is done with the same rules used to describe a method (parameters...)
The table below describes the "Interface" description.

Keyword		Description
Interface		used to define the type of interface associated to the device: Serial,gpib,udp,tcp,com,pci
HardwareConfig		Used to override the Init() and close() methods associated to the interface. Because the overriding of based on the same concept of the method description, it is possible to add parameters to these methods.
HardwareConfigName	Init	Used to identify the name of the method to be overridden.
	Close	
Remark: This tag is also used in the plugin description.		

Table 16 : "Interface" description table

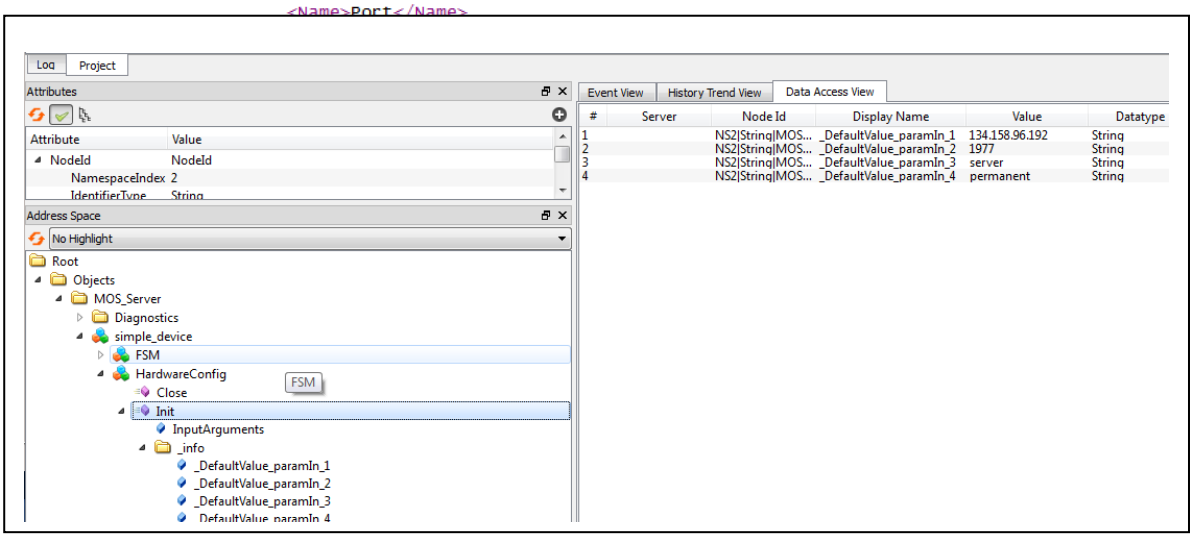
Depending on the selected connexion, some extra parameters are mandatory and need to be described.

Keyword	Type	nb	Description
HardwareConfig			Used to start a hardware connection description
TCP/UDP Interface			(see example below)
Address	Mandatory	1	IP address of the device
Port	Mandatory	1	Port number of the device
Connection	Mandatory	1	Used to set how is done the connection (as a client or server): "Client", "Server"
Validity	Mandatory	1	Used to initialize the connection permanently or temporally (connection is open when need and closed just after being used): Permanent, Temporary.
"COM" interface			
Port	Mandatory	1	Used to set the port name Ex: /dev/ttyUSB0
Speed	Mandatory	1	Communication speed Ex: 9600 (serial link on an USB port)
Parity	Optional	1	
NbBit	Optional	1	
StopBit	Optional	1	

Table 17 : "HardwareConfig" description table

Example of “HardwareConfig” keywords usage.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/15_declare_HardwareConfig.xml      *** -->
<!-- **** description : declare the configuration for the hardware device                    *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   -->
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare a hardware connection with the device in "tcp" -->
<!-- ****   declare the configuration for this connection in the "Init" method -->
<!-- ****     declare the addressIP  ex: 134.158.96.192-->
<!-- ****     declare the port number ex: 1977 -->
<!-- ****     declare that MOS_Server is the server in the open communication (client/server)-->
<!-- ****     declare that the communication is permament: -->
<!-- ****      so the socket is open at the beginning and close and the end of this application -->
<!-- ************************************************************ -->

<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
       <Name>MOS_Server</Name>
       <SimpleDevice>
              <!-- =========Here define the device ========================== -->
              <Name>simple_device</Name>
               <Interface>udp</Interface>
              <!-- =========Here define the config for this hardware ========================== -->
              <HardwareConfig>
                     <NameHardwareConfig>Init</NameHardwareConfig>
                     <Argument>
                            <Name>Address</Name>
                            <Type>string</Type>
                            <Access>Input</Access>
                            <DefaultValue>134.158.96.192</DefaultValue>
                     </Argument>
                     <Argument>
                            <Name>Port</Name>
```



4. THE DATA FORMAT

In some case, a device can sent data accordingly a well-defined data format. The data format can be used to analyse the data frame in order to extract the expected information. It is possible to describe the data format in a independent XML file and refer it in the device description. In the “SimpleDevice” section, the tag «Data-FrameStructureRef » can be used to set the XML file containing the data formation description.
Example: <DataFrameStructureRef>tramestruct.xml</DataFrameStructureRef>

The table below describes the different tags used to provide the device data format description.

Keyword	Type	nb	Description
DataFrameStructure			Used to start the data format description.
Header	Optional	1	Used to define a header. (if exists) Ex : AAAAAAAA
Id1	Optional	1	Ex : 3FE (for frame identification)
Id2	Optional	1	Ex : FF (for sub frame identification)
Data	Optional	1	The Data of the frame
Footer	Optional	1	Used to define a footer (if exists) Ex : AAAAAAAA

Table 18 : “DataFrameStructure" **description table**

The table below describes the data format “Header”

Keyword	Type	nb	Description
Header			Used to start the header description
Name	Optional	1	
Type	Optional	1	
Size	Optional	1	
EndDelimiter	Optional	1	

Table 19 : "Header " keyword description table

The table below describes the data format “Footer”

Keyword	Type	nb	Description
Footer			
Name	Optional	1	
Type	Optional	1	
Size	Optional	1	
EndDelimiter	Optional	1	

Table 20 : “ Footer " keyword description table

The table below describes the data format “id1”

Keyword	Type	nb	Description
Id1			This Id allow to define the identifier for the frame
Name	Optional	1	
Type	Optional	1	
Size	Optional:wq	1	

EndDelimiter	Optional	1	

Table 21 : "Id1" keyword description table

The table below describes the data format "id2"

Keyword	Type	nb	Description
Id2			This Id allow to define a second identifier for the frame
Name	Optional	1	
Type	Optional	1	
Size	Optional	1	
EndDelimiter	Optional	1	

Table 22 : "Id2" keyword description table

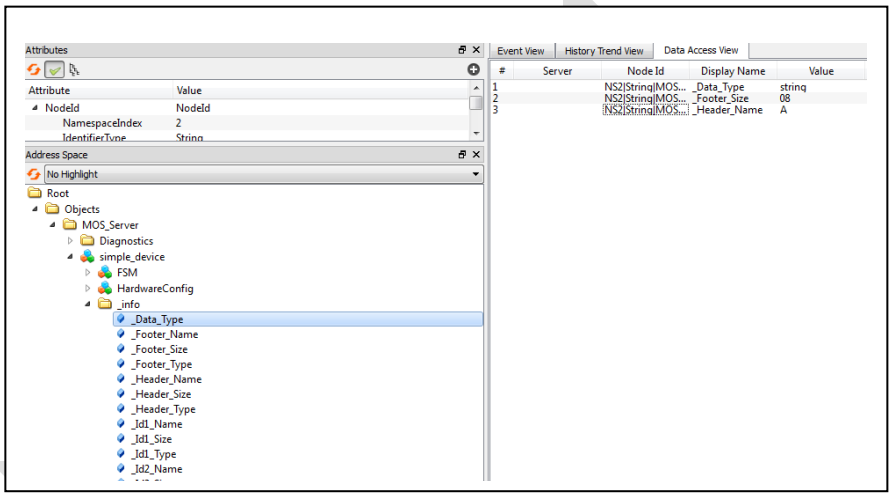
The table below describes the data format "Data"

Keyword	Type	nb	Description
Data			
Name	Optional	1	
Type	Optional	1	
Size	Optional	1	
EndDelimiter	Optional	1	

Table 23 : "Data" keyword description table

Example of "DataFrameStructure" keywords usage.

```
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/16_exempleFrameStructure.xml       *** -->
<!-- **** description : declare the configuration for the hardware device                    *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file is using with the file : 16_declare_FrameStructure.xml -->
<!-- ****   declare the structure of frames receiv by the device -->
<!-- ****   define a header  AAAAAAAA -->
<!-- ****   define the identifier    : id1 : 1 integer -->
<!-- ****   define the sub identifier: id2 : 1 integer -->
<!-- ****   define the type of the data : string -->
<!-- ****   define a header  AAAAAAAA -->
<!-- ****    -->
<!-- ****   This frame structure example can be used for example to define -->
<!-- ****   a data format beetween a plugin and the MOS_server -->
<!-- ****   Here the id1 define the identifier of a datapoint -->
<!-- ****   The id2 define the lengh of the data string -->
<!-- ****   The data content the message or value -->
<!-- ************************************************************ -->

<DataFrameStructure xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
        <Header>
                <Name>A</Name>
                <Type>int8</Type>
                <Size>08</Size>
        </Header>
        <Id1>
                <Name>idNum</Name>
                <Type>int8</Type>
                <Size>01</Size>
        </Id1>
        <Id2>
                <Name>idNum1</Name>
                <Type>int8</Type>
                <Size>01</Size>
        </Id2>
        <Data>
                <Type>string</Type>
        </Data>
        <Footer>
                <Name>A</Name>
                <Type>int8</Type>
                <Size>08</Size>
        </Footer>
</DataFrameStructure>
```




```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ********************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/16_declare_FrameStructure.xml      *** -->
<!-- **** description : declare the configuration for the hardware device                    *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   -->
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare that this device receive some frames -->
<!-- ****   the definition of the structure of frames are define in the file : "tramstruct.xml" -->
<!-- ************************************************************ -->
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
        <Name>MOS_Server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <DataFrameStructureRef>16_exempleFrameStructure.xml</DataFrameStructureRef>
        </SimpleDevice>
</OPCUA>
```

5. THE FINITE STATE MACHINE (FSM)

Defining a “simple” device (SimpleDevice tag) implies the creation of a FSM associated to the device in the OPCUA server. The finite state machine is build accordingly to the schema below:

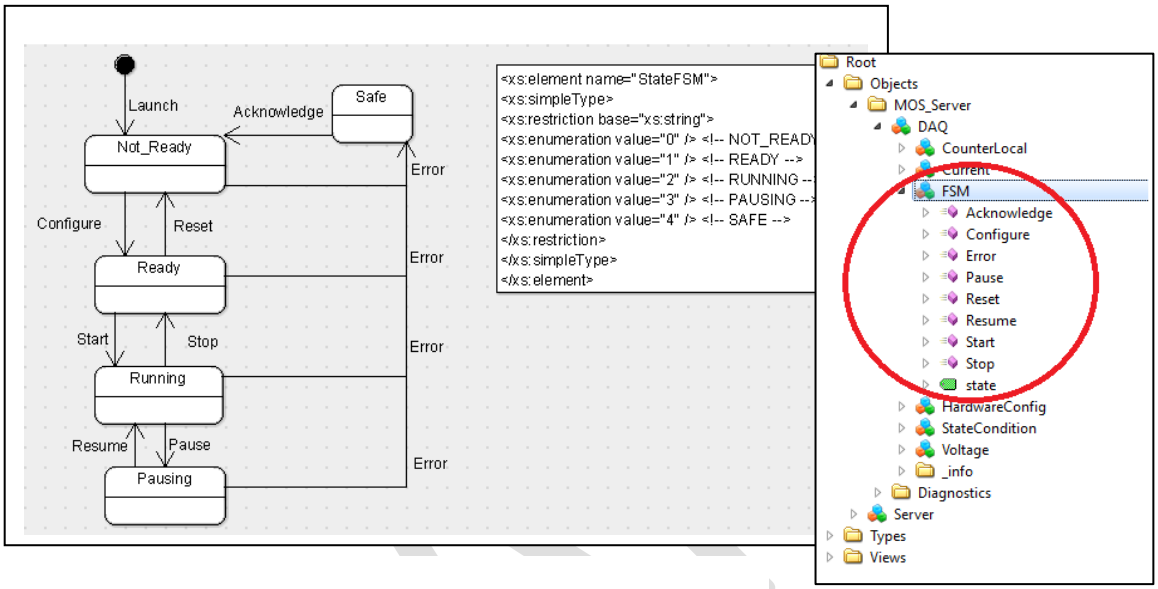


Figure 9 : FSM diagram

The transitions from a state to another are managed by method call in the OPCUA server. By default, all the method are created event if no action is connected to these transitions.

On the right, a view of the FSM organization in the OPCUA name space.

In the device description, it is also possible to override the FSM transitions methods. It means that the transitions method can be enriched by adding new parameters but also by associating device instructions to be sent to the device during the transition method call. The overriding is done with the “<FSM>”tag.

The table below describes the keywords used to override the FSM transition methods.

Keyword	Type	nb	Description
FSM			Start the FSM method overriding
NameFSM	Mandatory	1	Used to identify the concerned method. The name need to be imperatively in the following list: Configure,Start,Stop,Pause,Resume,Reset,Acknowledge
Sequence	Optional	N	Used to set a sequence of methods calls. See chapter on sequence.
Argument	Optional	N	Used to add a new argument to this method. See “Argu-

			ment" chapter.
DeviceInstruction	Optional	N	Used to set the command sent to the device during this method call. The value corresponds to an mnemonic in the device instruction set described in the device by the "<Instruction_Set>". See chapter on « instruction_set ».

Table 24 : "FSM" method overriding description table

Example of "FSM" keywords usage.

```
<FSM>
  <NameFSM>Start</NameFSM>
  <DeviceInstruction>
    <Value>FSMstateInstruction_Start</Value>
    <DeviceInstruction>
</FSM>

<FSM>
  <NameFSM>Reset</NameFSM>
  <DeviceInstruction>
    <Value>FSMstateInstruction_Reset</Value>
    <DeviceInstruction>
</FSM>
```

```
<FSM>
  <NameFSM>Configure</NameFSM>
  <Argument>
    <Name>ConfigId</Name>
    <Type>string</Type>
    <Access>Input</Access>
    <DefaultValue>defaultConfigId</DefaultValue>
  </Argument>
  <DeviceInstruction>
    <Value>FSMstateInstruction_Configure</Value>
    <DeviceInstruction>
</FSM>
```

Example: overriding method with parameters.

Remember: You can affect a default value for each parameter.

6. ANALYSIS OF FRAMES AND DISTRIBUTION INTO SEVERAL DATAPOINTS.

When the OPCUA server receives information from a device (example reception of data on the fly "like a weather station"), it can store the information in the data point associated. For this, it's necessary to have an association between the information from the hardware (ID) and the namespace of the data point.
This is the "Id" tag, which associates the datapoint with the frame received

In the description of a server, it's also possible to use a feature that analyzes the content of a datapoint when it was updating in order to store information's in the different sub-variables associated (parsing frames).
Example of use: allows, when updating a data point, corresponding to a 32-bit register, to store it in explicit variables. Each sub-variable corresponds to a bit of this register.

For this, it's necessary to report additional information to indicate that the data point should be treated, and it will store information correctly in the sub-variables of this datapoint.
Therefore, at the device level (simpleDevice tag), add the following tag: "DataIdDescriptorRef" and add the reference (name of XML file) that will contain the information for these particular datapoint.

The table below describes the keywords used for these particular datapoints and who is integrated in this new XML file.

Keyword	Type	nb	Description
Frames_description			Starting point for this XML file. Allow to describe the automatic update of the sub-variables

CompoundDatapoint Et SimpleDatapoint	Optional	1	The statement is performed conventionally (see simpleDatapoint and CompoundDatapoint)
In the "SimpleDatapoint" part, we will add the following statements			
Frame_element	Optional	N	Allows declaring and making the division of this data-point into sub-variables. **Allows the division of datapoint with the type "int" or array of "int"**
Frame_elementString	Optional	N	Allows declaring and making the division of this data-point into sub-variables. **Allows the division of datapoint with the type "string".**

Table 25 : "Frames_description " description table

Keyword	Type	nb	Description
Frame_element			Allows declaring a sub-variable of the datapoint
Name	Mandatory	1	Name of the sub-variable. This name is used to identify the data point in the OPCUA name space. This object will be created accordingly to the hierarchy of the XML description.
Type	Mandatory	1	Allows defining the type of the sub-variable into this list. • int32 • int8 • int16
Index	Mandatory	1	Allows defining the index of the array (datapoint) which this sub-variable is associated. Index = 0 if the datapoint is not an array. Index = 0..N if the datapoint is an array.
Pos	Optional	1	Allows defining (if needed), the starting position (datapoint) which this sub-variable is associated. Example : if this sub-variable is associated at the bit n°3, then Pos =3 <Pos>3</Pos>
Nb	Optional	1	Allows defining (if needed), the number of bits (data-point) which this sub-variable is associated. Example : if this sub-variable is associated at the bit n°3, and the length is 2 bits (bit 3 and 4) then Pos =3 and Nb=2 <Pos>3</Pos> <Nb>2</Nb>

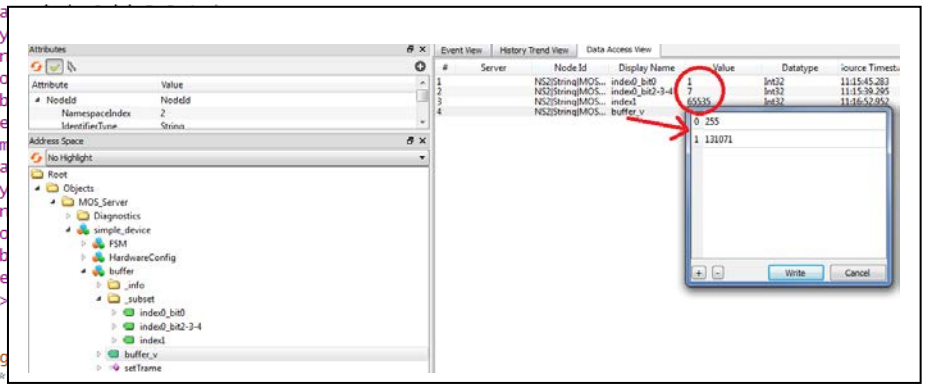
Table 26 : "Frames_element " description table

Keyword	Type	nb	Description
Frame_elementString			Allows declaring a sub-variable of the datapoint **The analysis is formed as string.**
Name	Mandatory	1	Name of the sub-variable. This name is used to identify the data point in the OPCUA name space. This object will be created accordingly to the hierarchy of the XML description.
Type	Mandatory	1	Allows defining the type of the sub-variable into this list. • string • xml (avoir)
EndDelimiter	Mandatory	1	Allows defining the separator character for the analyses.

Table 27 : "Frames_elementString " description table

Example of analysis frames.

```
<!-- ********************************************************************************************************** -->
<!-- **** file : ../example/0_Tutorial/0_firstApplication/17_exempleFrameAnalysis.xml        *** -->
<!-- **** description : declare the configuration for the hardware device                    *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file is using with the file : 17_declare_FrameAnalysis.xml -->
<!--  declare a simpledatapoint call "buffer" with the type int32 -->
<!--   this datapoint represent for example a buffer of 2*32 bits so ArraySize=2 -->
<!--   -->
<!--   declare some sub_variable for this buffer who is automaticaly updated -->
<!--   when the datapoint is updated -->
<!--  declare the sub-variable call "index0_bit0 who represent the bit0 of the index=0 of the "buffer" -->
<!--  declare the sub-variable call "index0-bit2-3-4" who represente 3 bits of the index=0 of the "buffer" -->
<!--  declare the sub-variable call "index1" who represent 16 bits of the index=1 of the "buffer"  -->
<!-- ************************************************************ -->

<Frames_description xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">

        <SimpleDatapoint>
                <Name>buffer</Name>
                <Type>int32</Type>
                <Frame_element>
                        <Name>index0_bit0</Name>
                        <Type>int32</Type>
                        <Index>0</Index>
                        <Pos>0</Pos>
                        <Nb>1</Nb>
                </Frame_element>
                <Frame_element>
                        <Na
                        <Ty
                        <In
                        <Po
                        <Nb
                </Frame_ele
                <Frame_elem
                        <Na
                        <Ty
                        <In
                        <Po
                        <Nb
                </Frame_ele
                <ArraySize>
        </SimpleDatapoint>
</Frames_description>
<?xml version="1.0" encoding
<!-- ********************
<!-- **** file : ../example/0_Tutorial/0_firstApplication/17_declare_FrameAnalysis.xml        *** -->
<!-- **** description : declare some sub-variable automaticaly updated                       *** -->
<!-- **** author      : panazol@lapp.in2p3.fr                                                *** -->
<!-- **** date        : 28/08/14                                                             *** -->
<!-- ********************************************************************************************************** -->


<!-- ************************************************************ -->
<!-- **** This file start a OPCUA server call MOS_server   -->
<!-- **** and add a simple device call "simple_device"-->
<!-- ****   declare some specific datapoints and sub-variables -->
<!-- ****   in the "17_exempleFrameAnalysis.xml" file -->
<!-- ************************************************************ -->
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
        <Name>MOS_Server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                <DataIdDescriptorRef>17_exempleFrameAnalysis.xml</DataIdDescriptorRef>
        </SimpleDevice>
</OPCUA>
```



7. AUTOMATIC DATA REFRESHING(PUSH/PULL).

A device emits the data in different way. In some cases, the device is responsible to sent the data to the outside (PUSH mode) and in some other cases, the connected client has to trigger the device data emission by itself (PULL mode). Because of these 2 possibilities of data emission, it is essential to provide a way of describing this mechanism in order to explain how a data point is refreshed by the data coming out of the device. However, in a "SimpleDataPoint" context, it is possible to enable an automatic refresh independently of the device mode. For using this option, the keyword "***MonitoringRate"*** has to be used to define the data point value-refreshing rate. The refreshing mode is based on a "get" method and obviously this method needs to be described with all its parameters. If a plugin is associated to the MOS, the corresponding "get" method of the plugin interface will be called by using the "get" method description (mainly parameters: see plugin chapter). If no "get"

method is found in the description, and error will be displayed during the MOS launching time.

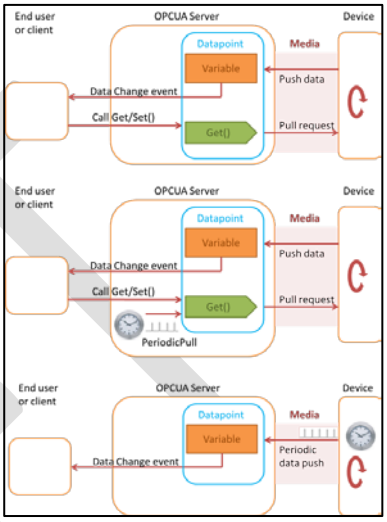


Figure 10 : push/pull : data refreshing

Keyword	Type	nb	Description
MonitoringRate			Used to set the data-refreshing rate. When the timeout is reached, the OPC UA server will send to the device the associated command in order to get the current value of the data point. The device will send back the data containing the data point value. The timeout value has to be initialize with the following values: 1 : 1 time / second 10 : 1 time / 10 seconds 30 : 1 time / 30 seconds 60 : 1 time / 1 minute 900 : 1 time / 15 minutes 1800 : 1 time / 30 minutes 3600 : 1 time / hour

Table 28 : "MonitoringRate" description table

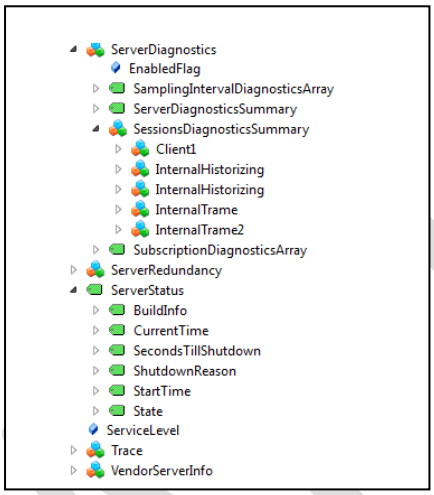
8. EXTRA SERVER INFORMATIONS.

This paragraph describes the extra information automatically added to the OPCUA namespace to control the server status itself. These information are divided in 2 parts:

The first part corresponds to a general aspects linked to the OPCUA specification and the second part dedicated to the MOS server.

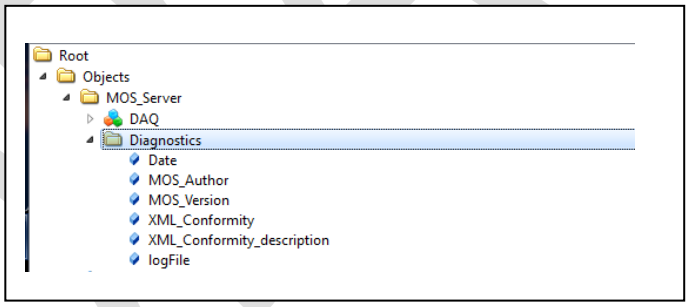
8.1 DIAGNOSTICS INFORMATIONS

As shown in the diagram below, a set of information is available to characterize the current stats of the OPCUA server.



8.2 M.O.S DIAGNOSTICS INFORMATIONS

In MOS context, a set of information is added to follow the server status as shown in the diagram below.



WRITING A PLUGIN

In some cases, a device is associated to software libraries provided by the device vendor. These libraries are used to facilitate the communication with the device. In this chapter, we will describe how to use these external libraries in a plugin and how to provide the plugin description.

1. WRITING A PLUGIN

Writing a plugin for MOS consists in in implementing an interface to delegate the data transfer from and to the device to this implementation. Basically, each method of the interface described below, will receive a string of characters corresponding to the command to send to the device and a set of values corresponding to the result of the command. The developer or the integrator in charge of the plugin implementation will have to write the implementation part for each method composing the interface. Of course, all the method do not

need to be implemented and the most appropriate ones need to be written. The main points to consider is that the plugin (thru the methods of the interface) is responsible to connect the device (open method), to close the connection (close method) and send the command to the device to get the value of the associated data point (Get method), to send the command to the device to set or modify the devoice status or configuration (Set method).

2. WHY A PLUGIN?

As explained above, it can be easier to use software libraries dedicated to the device mainly for the data encoding and decoding (.i.e. XBee protocol) instead of hardcoding and describing a complex protocol. In this paragraph, we will show how to implement the MOS plugin interface and how integrate this software in the MOS description. Developing a plugin is usually under the integrator responsibility but it does not need any OPCUA knowledge since the XML description is used to establish the connection between the plugin and the MOS.

3. HOW IT WORKS?

When a plugin is used in a MOS, all the communication and data exchanges are done thru the plugin. This means that the plugin receives via parameters the device instruction to be sent and also the parameters to be used to build the correct command (with the right parameters). As already explained above, a method associated to a data point is described with its parameters. During a method call and if a plugin is associated to the device management, the “cmd” (see below) is called. The parameter, named “chaine”, is built why concatenating the device instruction associated to the method and all the parameter figuring in the method description. A blank character separates each element in the result string. Then, the analysis of this string and the communication with the device (command, values...) is under the responsibility of the plugin developer.

4. PLUGIN IMPLEMENTATION

The plugin implementation can be done in 2 phases. The first one consists in writing a piece of software (by implementing the MOS plugin interface) and plugin unitary tests. The second one consists in describing the plugin in the MOS XML file. (plugin location, entry point in the plugin library)

3.1 IMPLEMENTATION

Basically, writing a MOS plugin is implementing the MOS interface. Obviously, this implementation can be integrated in a larger environment but this implemented class is the connection point with the MOS server.

3.2 L’INTERFACE DU PLUGIN

The C++ interface to implement is the following:

```
class PluginsInterface {
protected:
public:
        PluginsInterface() {
        }
        ;
        virtual int init(std::string chaine)=0;
        virtual int close()=0;

        virtual int cmd(std::string chaine, int commandStringAck, std::string *result)=0;

        virtual int set(std::string chaine,int commandStringAck)=0;
        virtual int set(std::string chaine,int commandStringAck, std::vector<Byte> tabValue)=0;
        virtual int set(std::string chaine,int commandStringAck, std::vector<short int> res)=0;
        virtual int set(std::string chaine,int commandStringAck, std::vector<int> res)=0;
        virtual int set(std::string chaine,int commandStringAck, std::vector<long> res)=0;
        virtual int set(std::string chaine,int commandStringAck, std::vector<float> res)=0;

        virtual int get(std::string chaine,int commandStringAck, std::string *result)=0;
        virtual int get(std::string chaine,int commandStringAck, std::vector<Byte> *tabValue)=0;
        virtual int get(std::string chaine,int commandStringAck, std::vector<short int> *res)=0;
        virtual int get(std::string chaine,int commandStringAck, std::vector<int> *res)=0;
        virtual int get(std::string chaine,int commandStringAck, std::vector<long> *res)=0;
        virtual int get(std::string chaine,int commandStringAck, std::vector<float> *res)=0;

};
```

It is an abstract class to be implemented by inheriting of this class. Each method of this class has to be implemented. However, only the methods corresponding the device behaviour need to be implemented and the other ones can be empty. Remarks: generally, the first parameter (string named “chaine”) contains the instruction to be sent to the device, and all the parameters

Méthode	Description
init(…)	This method is automatically called by the MOS when the OPCUA server starts. This method is used to initialize the plugin. For example, it can be used to initialize the connection with the device and to initialize connection to external resources(via configuration file, ...)
close(…)	This method is automatically called at the end of MOS program. This method is used to close the plugin. It means, for example, close the connection to the device and to all the external connection established during the plugin lifecycle.
cmd(…)	This method is generally called by the MOS when a client calls one method in the OPCUA Server (like open/start/etc…). This method is used to address a command to the device. The “value” parameter contains the instruction to be sent to the device. The “commandStringAck” parameter means that the command send to the device has to be acknowledged. In case of a command acknowledgement, the plugin is responsible to wait the device answer.
get (…)	This method is specifically called by the MOS when a client wants to set the value currently in the variable of one datapoint in the device. This method is used to get, from the device, the value associated to the data point. The returned value is done by using an array of values (the type of the array element is the type used to defined the data point). For the simple value (not represented by an array), the returned value will be stored a the position #0 f the array. The content of the array is then used to update the data point value(s)

set(…)	This method is specifically called by the MOS when a client wants to set the value currently in the variable of one datapoint in the device. This method is used to set the value of a data point. As a data point is associated to a device command, this method call will also send the instruction to the device. The function parameters are used: • To store the instruction parameters in order to build a complete device instruction. The information received by the plugin is built by using the associated device instruction described on the data point and all the parameters used in the method description. • The value currently in the datapoint is passed to the plugin by a vector. If the value is not an array, the value is stored in the position #0 of the array.

Table 29 : plugin function description

Set and Get method call:

The methods « set » and « get « of the plugin are called by the MOS depending on the Data Point type described in the XML file. All the methods are not mandatory (methods are kept empty) and only those, corresponding to the right data type, have to be implemented.

The “Get” and “Set” methods are used in the plugin to respectively get and set a value of a data point.

“Set” method can also be used to configure a device

These methods are called by the MOS only if “get” and/or “set” methods have been described in the XML file. (Interface description to the client)

These methods can be parameterized

The input parameters value (given by the client or default value) are concatenated as a string

Each element of the string represents the input value and are coded as a named-value pair (each element of the string are separated with a blank character)

MOS selects the method to be called accordingly to the type of the concerned data point

If a return value is expected by the client, the method has to have at least one output parameters. In case of many output parameters, the first one is used to contain the return value.

C++ plugin implementation example:

```
#ifndef COM_PLUGIN_H_
#define COM_PLUGIN_H_

#include <iostream>
#include <vector>

#include "../../../genericServerLib/include/plugins/cta_slc_pluginsInterface.
#include "com.h"

class COM_Plugin: public PluginsInterface , public Com {
public:
        COM_Plugin();
        int close();
        int init(std::string chaine);
        int cmd(std::string chaine, int commandStringAc

        int set(std::string chaine,int commandStringAck);
        int set(std::string value, int commandStringAck,std::vector<Byte> res);
        int set(std::string value, int commandStringAck,std::vector<short int> res);
        int set(std::string value, int commandStringAck,std::vector<int> res);
        int set(std::string value, int commandStringAck,std::vector<long> res);
        int set(std::string value, int commandStringAck,std::vector<float> res);

        int get(std::string chaine, int commandStringAck,std::string *result);
        int get(std::string value, int commandStringAck,std::vector<Byte> *res);
        int get(std::string value, int commandStringAck,std::vector<short int> *res);
        int get(std::string value, int commandStringAck,std::vector<int> *res);
        int get(std::string value, int commandStringAck,std::vector<long> *res) ;
        int get(std::string value, int commandStringAck,std::vector<float> *res);
private:
        void makeArgv(std::string chaine, int *argc, char** *argv);
};

typedef COM_Plugin *(*maker_protocole1)();
#endif //  COM_Plugin_H_
```

```
int COM_Plugin::init(std::string chaine) {
        int ret=0;
        int argc;
        char **argv;
        makeArgv(chaine, &argc, &argv);

        if(argc>1) setPort(argv[1]) ;
        if(argc>2) setSpeed(atoi(argv[2]));
        ret = Sopen();
        return ret;
}
```

```
int COM_Plugin::cmd(std::string chaine, int commandStringAck,std::string *result)
        int ret=0;
        ret = Swrite(chaine.c_str());
        return ret;
```

3.3 PLUGIN COMPILATION

In the MOS distribution, there is a directory containing the code sources already implemented and installed in MOS. These sources can be used as example to write a new plugin. In each directory, there is a Make file to be used for a direct compilation (with the make command) and also “CmakeList” files to do a pre-compilation with a Cmake command. A README.txt file is also available to explain the different steps of a “Cmake” compilation.

5. MOS PLUGIN DESCRIPTION

Writing a plugin is in some sense, providing a dynamic library. To be used, this library has to be loaded during the launching phase of a MOS application. So, if a plugin is used to manage a device, it has also to be described in the MOS description file (XML) to characterize the plugin (plugin library location, library entry point).

Keyword	Type	nb	Description
Plugins			Used to start a plugin description
Location	Mandatory	1	Used to define the location (access path) of the shared library containing the plugin.
Name	Mandatory	1	Used to define the entry point in the referenced library.

Table 30 : "Plugins" **Keywords description table**

```
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">

        <Name>MOS_Server</Name>
        <SimpleDevice>
                <Name>simple_device</Name>
                        <Plugins>
                                <Location>../plugins/libDNS.so</Location>
                                <Name>make_protocole1</Name>
                        </Plugins>
```

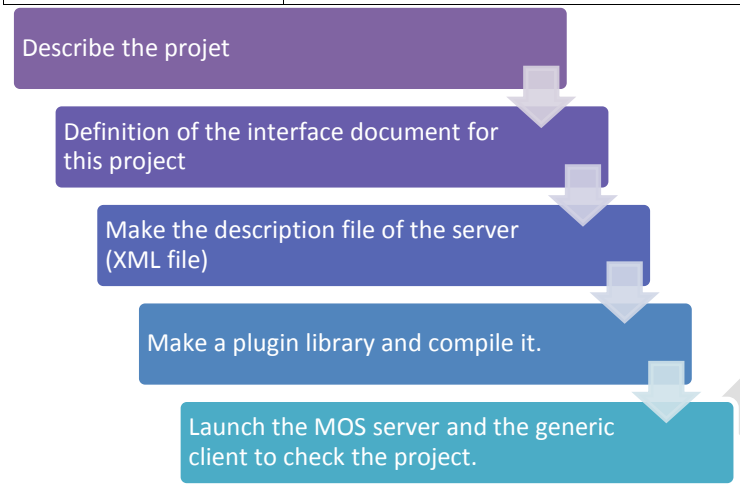
APPENDIX A : APPLICATION EXAMPLE

INTRODUCTION

The purpose of this appendix is to quickly understand with an example, the implementation of an OPCUA MOS server that meets your need. For this, we describe a simple but complete example, which will allow build an OPCUA MOS server with all the features that have been described in detail in the reference document.

1. PROCEEDING OF THIS DOCUMENT

In this document, we will proceed step by step as we would do when designing a real system. In our example the project is very simple. We just describe few lines for the different parts.



2. DESCRIBE THE PROJET

In order to choose a simple example but who contains all the functional aspects of a OPCUA MOS server, we offer an example of control and monitoring (with event management and alarms) of an equipment that must manage the closing and opening shutters (eg the shutters that protect the cameras CTA).
In order to simulate the system without actually having the hardware (shutters, motors, sensors etc...), we propose to integrate this application into an embedded ARM board with I/O board to simulate sensors and actuators of the system.
We will now describe more precisely the project.

OPCUA the server must be able to control the hardware remotely but it is also possible to drive the hardware locally via control buttons.

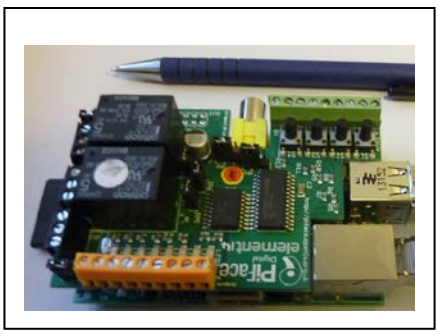
- A button allows the opening.
- A button allows the closing.
- A sensor detects the closing of the shutters
- A sensor detects the full opening of the shutter.

Clients can log onto the server for driving or monitoring the system.

DESCRIPTION OF HARDWARE IN ORDER TO SIMULATE THIS APPLICATION.

Because we don't really have the hardware parts, we will simulate with an embedded board the different Inputs and outputs of the system. For this, we will use a RaspberryPI embedded CPU board and a PIFACE board. The two boards communicate by an SPI bus

.



The OPCUA MOS server will run in the embedded system. The server must contact the PIFACE board via the SPI bus. So we will write a plugin to make an interface between the OPCUA server and the SPI bus.

3. THE SPECIFICATION DOCUMENT FOR THIS APPLICATION

Once the project described, we must now define the interface points that we want to control / monitor. In this chapter we define a specification on which will be based the OPCUA server.

Here the description is very simple.
We want:

- Know the state of the shutter in real time
- Be informed of any action on the shutter (closing by another client or local)
- Know the history of the state of the shutter.
- Know the history of actions on the shutter.
- Having an alarm when the shutter is not closed.

- Open / close the shutter

The OPCUA MOS server must make all these services.

4. THE XML FILE WHO DESCRIBE THIS SERVER.

Now that we have the necessary information, we can build the associated XML file.
In this chapter, we will see all the descriptive parts necessary to build the OPCUA server.

First step: start with the starting tag of our description file

```
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
  <Name>MOS_server</Name>

</OPCUA>
```

We describe a server that manages one shutter. We will declare a "Simpledevice" that we call "shutter"

```
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
<Name>MOS_server</Name>
<SimpleDevice>
                <Name>shutter</Name>
</SimpleDevice>
</OPCUA>
```

Because the access to the hardware will be realized by the SPI bus, we will write a plugin library (see below). It's necessary to declare this plugin in the description file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
  <Name>MOS_server</Name>
  <SimpleDevice>
    <Name>shutter</Name>
      <Plugins>
          <Location>../plugins/libRaspberry_PiFace.so</Location>
          <Name>make_protocole1</Name>
      </Plugins>
  </SimpleDevice>
</OPCUA>
```

The access at the hardware is therefore made by the plugin by some instructions. The list of instructions should be described in the tag <instruction_set>

```xml
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
  <Name>MOS_server</Name>
  <SimpleDevice>
    <Name>shutter</Name>
      <Plugins>
          <Location>../plugins/libRaspberry_PiFace.so</Location>
          <Name>make_protocole1</Name>
      </Plugins>

      <Instruction_set>
        <Instruction>
          <Name>Openshutters</Name>
          <Cmd>shutters v 1</Cmd>
        </Instruction>
        <Instruction>
          <Name>Closeshutters</Name>
          <Cmd>shutters v 0</Cmd>
        </Instruction>
        <Instruction>
          <Name>getState</Name>
          <Cmd>getState</Cmd>
        </Instruction>
      </Instruction_set>

  </SimpleDevice>
</OPCUA>
```

We want to drive the shutter through two actions from a client interface: Open and Close. We will declare two methods. Close and Open through the tag <method>

```xml
<?xml version="1.0" encoding="UTF-8"?>
<OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
  <Name>MOS_server</Name>
  <SimpleDevice>
    <Name>shutter</Name>
      <Plugins>
          <Location>../plugins/libRaspberry_PiFace.so</Location>
          <Name>make_protocole1</Name>
      </Plugins>

      <Instruction_set>
        <Instruction>
          <Name>Openshutters</Name>
          <Cmd>shutters v 1</Cmd>
        </Instruction>
        <Instruction>
          <Name>Closeshutters</Name>
          <Cmd>shutters v 0</Cmd>
        </Instruction>
        <Instruction>
          <Name>getState</Name>
          <Cmd>getState</Cmd>
        </Instruction>
      </Instruction_set>

      <Method>
        <Name>Open</Name>
      </Method>
      <Method>
        <Name>Close</Name>
      </Method>

  </SimpleDevice>
</OPCUA>
```

For each method, we must make the association between this method and the good instruction among the list of instructions.

```xml
      <Method>
        <Name>Open</Name>
        <DeviceInstruction>
          <Value>Openshutters</Value>
        </DeviceInstruction>
      </Method>
      <Method>
        <Name>Close</Name>
        <DeviceInstruction>
          <Value>Closeshutters</Value>
        </DeviceInstruction>
      </Method>
```

We also want that when an action occurs, an event is associated. We will declare the message associated with this action.

```
<Method>
  <Name>Open</Name>
  <DeviceInstruction>
    <Value>Openshutters</Value>
  </DeviceInstruction>
  <Event>
    <Message>open the shutters</Message>
  </Event>

</Method>
<Method>
  <Name>Close</Name>
  <DeviceInstruction>
    <Value>Closeshutters</Value>
  </DeviceInstruction>
  <Event>
    <Message>close the shutters</Message>
  </Event>

</Method>
```

We want to know the status of shutter given by the hardware that will be read periodically.
We therefore declare a "simpleDatapoint" which we call "state".
We can define three values:
0: shutter closed.
1: shutter not closed but not completely opened.
2: shutter opened.
We can declare the type as "integer"

```
</Method>
<Method>
  <Name>Close</Name>
  <DeviceInstruction>
    <Value>Closeshutters</Value>
  </DeviceInstruction>
  <Event>
    <Message>close the shutters</Message>
  </Event>
</Method>

<SimpleDatapoint>
  <Name>state</Name>
  <Type>int32</Type>
</SimpleDatapoint>
```

Because we want to have a history on the state of the shutter we declare the tag: <Historizing> 1

```
<SimpleDatapoint>
  <Name>state</Name>
  <Type>int32</Type>
  <Historizing>1</Historizing
</SimpleDatapoint>
```

Because we want an alarm when the shutter is not closed, we must associate an alarm with this data point. For that, we declare in the tag<Alarm_NotEqual> the type of message and the alarm conditions.

```
47        <SimpleDatapoint>
48          <Name>state</Name>
49          <Type>int32</Type>
50          <Historizing>1</Historizing>
51
52          <Alarm>
53            <Alarm_NotEqual>
54              <Value>0</Value>
55              <Message>shutternotclose</Message>
56              <Severity>500</Severity>
57              <Enable>true</Enable>
58            </Alarm_NotEgal>
59          </Alarm>
```

View the complete description file:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- ********************************************************************************************** -->
3  <!-- **** file : ../example/0_Tutorial/0_firstApplication/18_simpleProjectExample.xml        *** -->
4  <!-- **** description : simple project describe into the appendix of the MOS document        *** -->
5  <!-- **** author      : panazol@lapp.in2p3.fr                                                 *** -->
6  <!-- **** date        : 28/08/14                                                              *** -->
7  <!-- ********************************************************************************************** -->
8
9
10 <!-- ********************************************************************** -->
11 <!-- **** This file describe a little project of shutter simulation      --> 
12 <!-- **** this project use an external library to manage the SPI bus --> 
13 <!-- **** so we integrate this library with a plugin --> 
14 <!-- **** We declare 2 methods and monitore the state of the shutter --> 
15 <!-- **** we manage some events when a method is called --> 
16 <!-- **** we manage an alarm when the shutter is not closed --> 
17 <!-- ********************************************************************** -->
18 <OPCUA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
19   xsi:noNamespaceSchemaLocation="http://lappweb.in2p3.fr/~panazol/xsd/v1_0_0/dict_cta.xsd">
20   <Name>MOS_server</Name>
21   <SimpleDevice>
22     <Name>shutter</Name>
23       <Plugins>
24           <Location>../plugins/libRaspberry_PiFace.so</Location>
25           <Name>make_protocole1</Name>
26       </Plugins>
27
28       <Instruction_set>
29         <Instruction>
30           <Name>Openshutters</Name>
31           <Cmd>shutters v 1</Cmd>
32         </Instruction>
33         <Instruction>
34           <Name>Closeshutters</Name>
35           <Cmd>shutters v 0</Cmd>
36         </Instruction>
37         <Instruction>
38           <Name>getState</Name>
39           <Cmd>getState</Cmd>
40         </Instruction>
41       </Instruction_set>
42
43       <Method>
44         <Name>Open</Name>
45         <DeviceInstruction>
46           <Value>Openshutters</Value>
47         </DeviceInstruction>
48         <Event>
49           <Message>open the shutters</Message>
50         </Event>
```

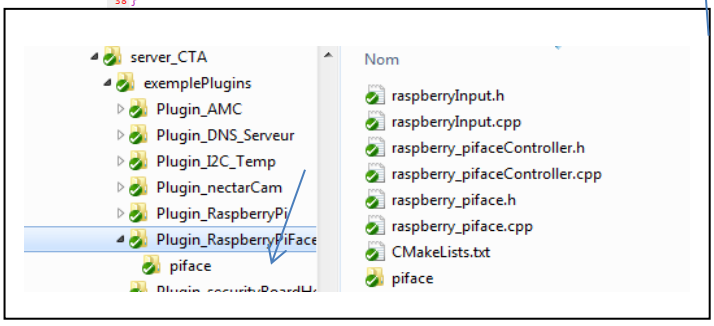
```
52       </Method>
53       <Method>
54         <Name>Close</Name>
55         <DeviceInstruction>
56           <Value>Closeshutters</Value>
57         </DeviceInstruction>
58         <Event>
59           <Message>close the shutters</Message>
60         </Event>
61       </Method>
62
63       <SimpleDatapoint>
64         <Name>state</Name>
65         <Type>int32</Type>
66         <Historizing>1</Historizing>
67
68         <Alarm>
69           <Alarm_NotEqual>
70             <Value>0</Value>
71             <Message>shutternotclose</Message>
72             <Severity>500</Severity>
73             <Enable>true</Enable>
74           </Alarm_NotEqual>
75         </Alarm>
76
77       </SimpleDatapoint>
78   </SimpleDevice>
79 </OPCUA>
```

5. MAKE THE PLUGIN FILE

Cette partie n'est pas obligatoire en fonction de vos besoins, mais nous allons ici construire un plugin spécifique pour l'accès à la partie SPI de la carte PIFACE.
Dans ce chapitre nous allons décrire la façon les éléments nécessaire afin de répondre à la norme d'un plugin MOS.
Nous verrons aussi la partie compilation de ce plugin dans un environnement ARM afin de la porte dans la carte RASPBERRYPI.

Dans notre projet nous pilotons le bus SPI en utilisant une librairie externe. Nous intègrerons cette librairie et l'appel de fonction afin de communiquer avec la PIFACE. Il faut de l'autre côté intégrer cette nouvelle librairie afin de la rendre plugin compatible avec MOS.
Il faut intégrer le fichier d'entête et définir toutes les fonctions virtuelles nécessaires.
Nous allons

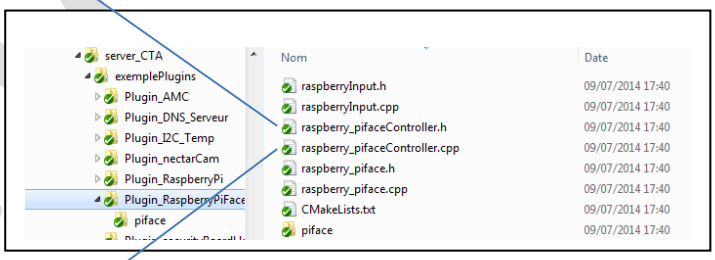
```cpp
#include <unistd.h>

#include "raspberry_piface.h"
#include <cstring>
#include <cstdlib>
#include <getopt.h>

using namespace std;

void raspberryPlugin::makeArgv(std::string chaine, int *argc, char** *argv) {
        std::string chaine2 = chaine;
        int flag = 1;
        int cpt = 1;
        std::size_t found;
        int pos = 0;
        *argv = (char**) malloc(20 * sizeof(char*));
        int i = 0;
        while (flag) {
                (*argv)[i++] = &chaine[pos];
                found = chaine2.find(' ');
                if (found == std::string::npos)
                        flag = 0;
                else {
                        cpt++;
                        if (chaine2[0] == '"') {
                                std::string chaine3 = chaine2;
                                chaine3 = chaine3.substr(1);
                                found = chaine3.find('"');
                                found = found + 2;
                        }
                        pos = pos + found + 1;
                        chaine[pos - 1] = '\0';
                        chaine2 = chaine2.substr(found + 1);
                }
        }
        (*argv)[i] = NULL;
        *argc = cpt;
}
```




```cpp

int raspberryPlugin::cmd(std::string chaine, int commandStringAck, std::string *result) {
        int ret = 0;
        int argc;
        char **argv;

        std::string cmd;

        makeArgv(chaine, &argc, &argv);
        cmd = argv[0];

        if (cmd.compare("openInstruction") == 0) {
                ret = m_raspberryController->open();
                return ret;
        }
        if (cmd.compare("closeInstruction") == 0) {
                ret = m_raspberryController->close();
                return ret;
                        }

        if (cmd.compare("kill") == 0) {
                ret = m_raspberryController->stop();
                delete m_raspberryController;
                return ret;
        }
        return ret;
}

int raspberryPlugin::stop() {
        int ret=0;
        ret  = m_raspberryController->stop();
        return ret;
}

int raspberryPlugin::init(std::string chaine) {
        int ret=0;
        m_raspberryController = new RaspberryController();
        ret = m_raspberryController->startRun();
        return ret;
}


int raspberryPlugin::get(std::string value, int commandStringAck,std::vector<int> *res){
                int ret=0;
                res->resize(1);
                (*res)[0]=m_raspberryController->getStateShutter();
                return ret;
};


extern "C" {
raspberryPlugin *make_protocole1() {
        return new raspberryPlugin();
}
```

```cpp
#ifndef RASPBERRY_H_
#define RASPBERRY_H_

#include <iostream>
#include "map"
#include "vector"
#include "piface/pfio.h"
#include "raspberry_pifaceController.h"

#include "../../genericServerLib/include/plugins/cta_slc_pluginsInterface.h"


class raspberryPlugin: public PluginsInterface {
public:
  raspberryPlugin() {};

        int close(){return 0;};
        int init(std::string chaine);
        int cmd(std::string chaine, int commandStringAck,std::string *result);

        int set(std::string chaine,int commandStringAck){return 0;};
        int set(std::string value, int commandStringAck,std::vector<Byte> res){return 0;};
        int set(std::string value, int commandStringAck,std::vector<short int> res){return 0;};
        int set(std::string value, int commandStringAck,std::vector<int> res){return 0;};
        int set(std::string value, int commandStringAck,std::vector<long> res){return 0;};
        int set(std::string value, int commandStringAck,std::vector<float> res){return 0;};

        int get(std::string value, int commandStringAck,std::vector<int> *res);
        int get(std::string chaine, int commandStringAck,std::string *result) {return 0;};
        int get(std::string value, int commandStringAck,std::vector<Byte> *res){res->resize(1); (*res)[0]=0;return 0;};
        int get(std::string value, int commandStringAck,std::vector<short int> *res){res->resize(1); (*res)[0]=0;return 0;};
        int get(std::string value, int commandStringAck,std::vector<long> *res) {res->resize(1); (*res)[0]=0; return 0;};
        int get(std::string value, int commandStringAck,std::vector<float> *res){res->resize(1); (*res)[0]=0; return 0;};

private:
        void makeArgv(std::string chaine, int *argc, char** *argv);
        int stop();

        RaspberryController* m_raspberryController;

};

typedef raspberryPlugin *(*maker_protocole1)();
#endif //  raspberryPlugin_H_
```

```cpp
#ifndef __RASPBERRYCONTROLLER_H__
#define __RASPBERRYCONTROLLER_H__

#include "string"
#include "../../../include/uabase/uathread.h"


typedef unsigned char Byte;

class RaspberryController: public UaThread {
public:
        RaspberryController();
        ~RaspberryController();
        void run();
        int  stop();
        void pause();
        void resume();
        int  startRun();
        int  open();
        int  close();
        int getStateShutter();
private:
        RaspberryController* m_raspberryController;
        void cmdopen();
        void cmdclose();

        int m_stop;
        int m_pause;
        int m_openShutter;
        int m_closeShutter;
        int m_stateShutter;
};

#endif // __RASPBERRYCONTROLLER_H___
```



```cpp
#include "raspberry_pifaceController.h"
#include "./piface/pfio.h"

RaspberryController::RaspberryController() {
        m_pause = OpcUa_False;
        m_stop = OpcUa_False;
        m_closeShutter = OpcUa_False;
        m_openShutter = OpcUa_False;
        m_stateShutter = 0;
}

RaspberryController::~RaspberryController() {
        m_pause = OpcUa_True;
        m_stop = OpcUa_True;
        wait();
}

void RaspberryController::pause() {
        m_pause = OpcUa_True;
}

void RaspberryController::resume() {
        m_pause = OpcUa_False;
}

int RaspberryController::stop() {
        int ret=0;
        m_stop = OpcUa_True;
        return ret;
}

int RaspberryController::getStateShutter() {
        return m_stateShutter;
}

int RaspberryController::open() {
        int ret=0;
        m_closeShutter = 1;
        return ret;
}

int RaspberryController::close() {
        int ret=0;
        m_openShutter = 1;
        return ret;
}
```

```cpp
void RaspberryController::cmdclose() {
        printf("sonde jl RaspberryController::shutter cmd : close\n");
        char patterns[] = { 0x1, 0x3, 0x7, 0xF, 0x1F, 0x3F, 0x7F, 0xFF };
        int i;
        m_stateShutter = 80;
        for (i = 0; i < 8; i++) {
                pfio_write_output(patterns[i]);
                m_stateShutter -= 10;
                sleep(1);
        }
        m_closeShutter = OpcUa_False;
}

void RaspberryController::cmdopen() {
        printf("sonde jl RaspberryController::shutter cmd : open\n");
        char patterns[] = { 0xFF, 0x7F, 0x3F, 0x1F, 0xF, 0x7, 0x3, 0x1 };
        int i;
        m_stateShutter = 0;
        for (i = 0; i < 8; i++) {
                pfio_write_output(patterns[i]);
                m_stateShutter += 10;
                sleep(1);
        }
        m_openShutter = OpcUa_False;
}



void RaspberryController::run() {
        int val;
        if (pfio_init() < 0)
                return;

        while (m_stop == OpcUa_False) {
                if (m_pause == OpcUa_False) {
                        usleep(1000000);
                        val = pfio_read_input();
                        printf("sonde jl val = %d\n", val);
                        if (val & 0x1) cmdclose();
                        if (val & 0x2) cmdopen();
                        if (m_closeShutter == OpcUa_True) cmdclose();
                        if (m_openShutter == OpcUa_True)  cmdopen();
                }
        }
        pfio_deinit();
}

int  RaspberryController::startRun() {
        int ret=0;

        start();
        return ret;
}
```

6. LAUNCH THE SERVER AND CHECK THE PROJET.

Une fois toute les étapes précédentes effectuées, il nous reste à lancer et tester notre applicatif.

LAUNCH THE SERVER

LAUNCH THE GENERIC CLIENT

APPENDIX B : XML EDITOR TOOL USER GUIDE

1. INTRODUCTION

This document allows you to show, modify and create your XML description file.
This XML description file is absolutely necessary for launch the “Generic OPCUA server” application.

- You can use any Editor file to read and write this file.
- You can also use Eclipse to manage your Xml file. This application can help you to create files with the good syntax (according with the good XSD dictionary).

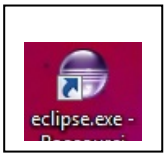
This document shows you how:

Install the Xml Editor Plugin for Eclipse.

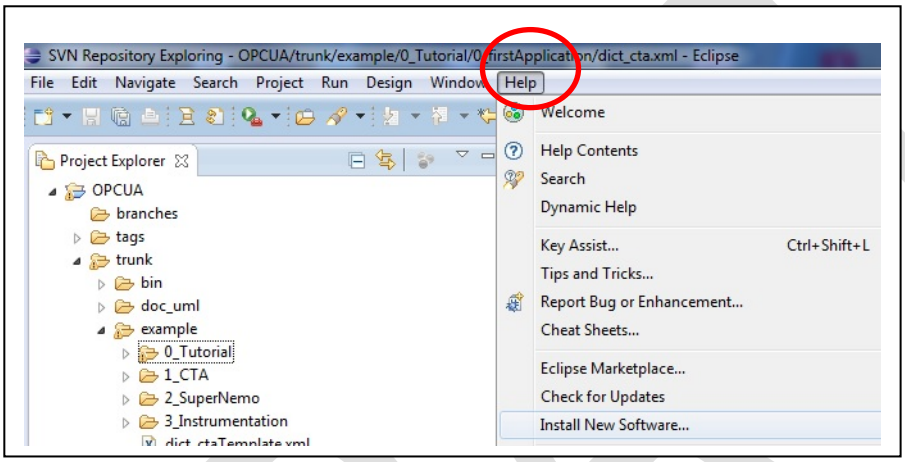
Tutorial for create a new XML file for your project.

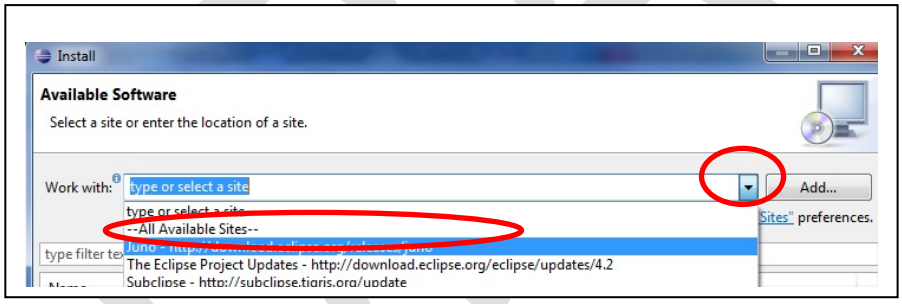
2. INSTALLATION ECLIPSE XML EDITOR

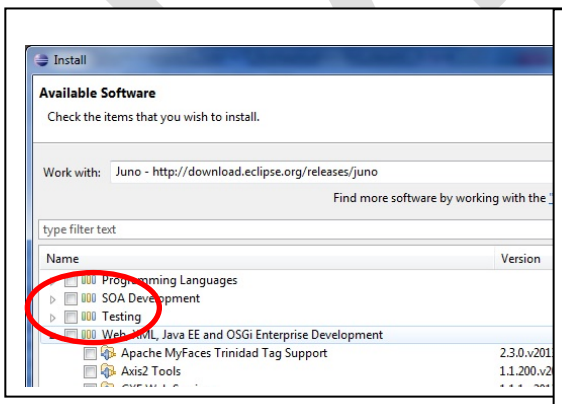
Launch your Eclipse Application

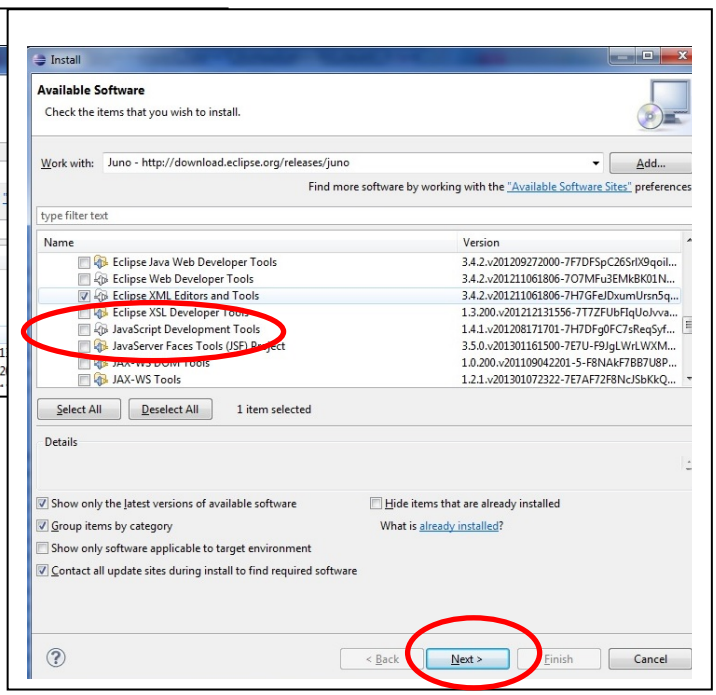


Install XML editor and Tools



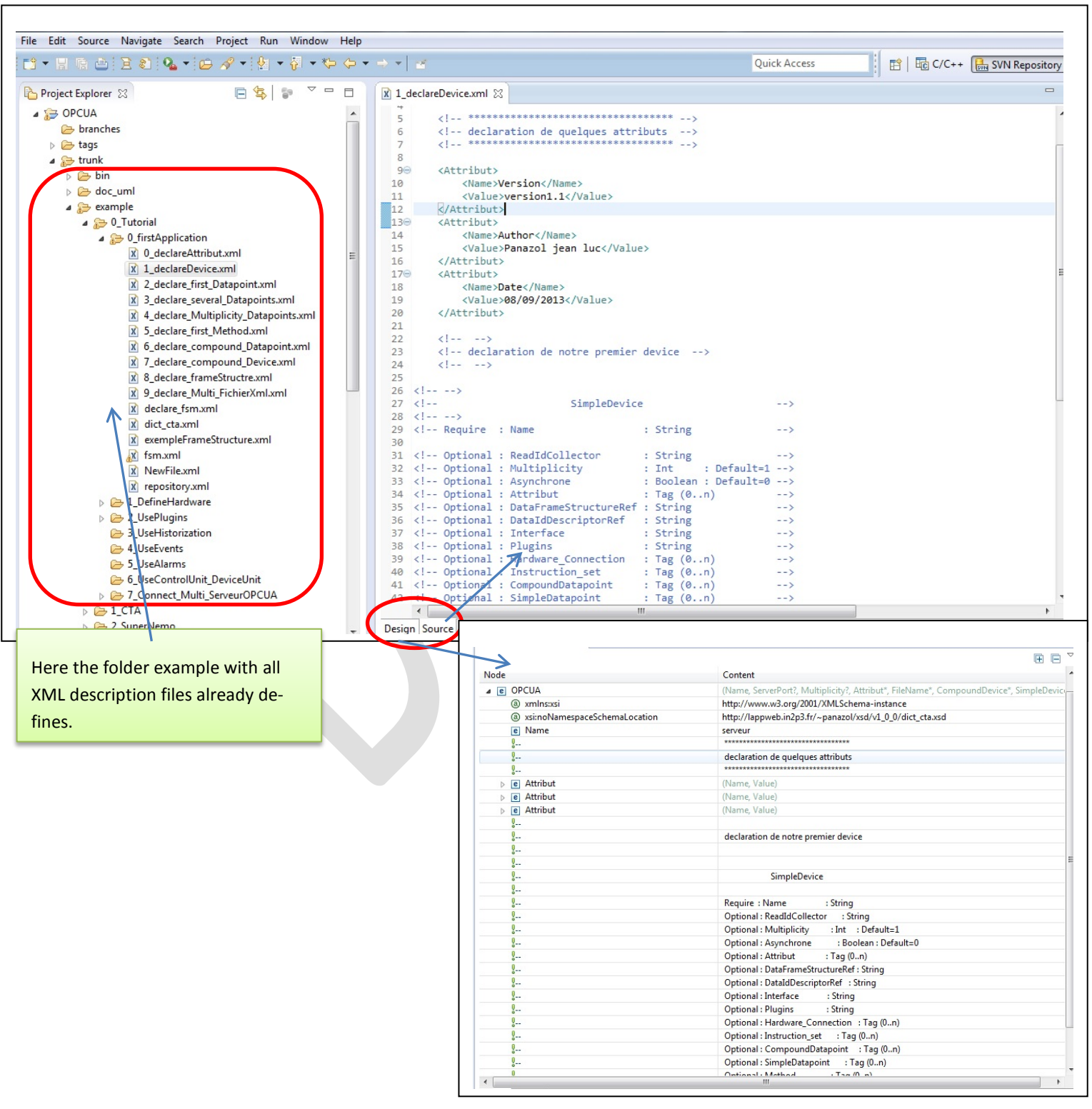






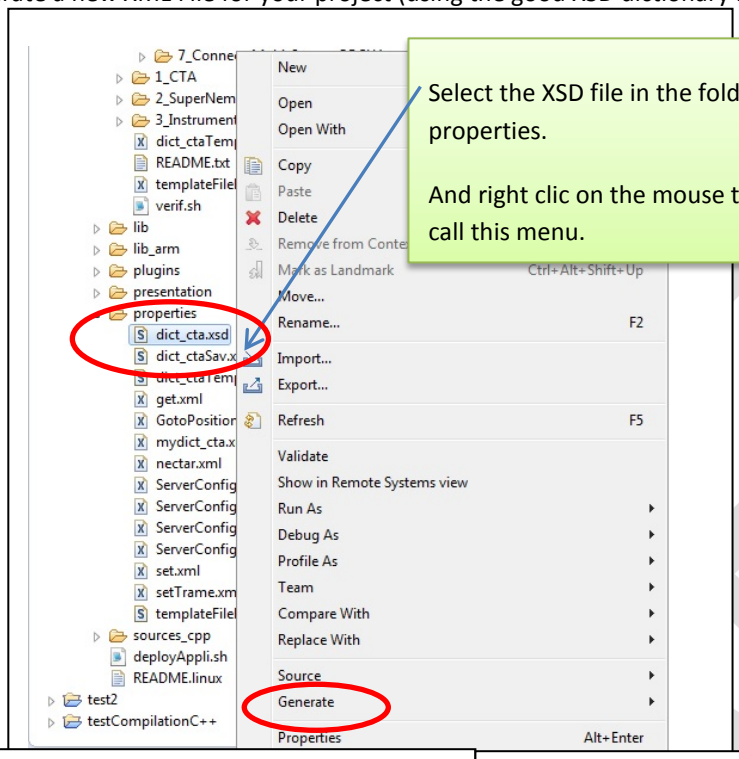
3. USING XML DESCRIPTION FILE FOR YOUR PROJECT

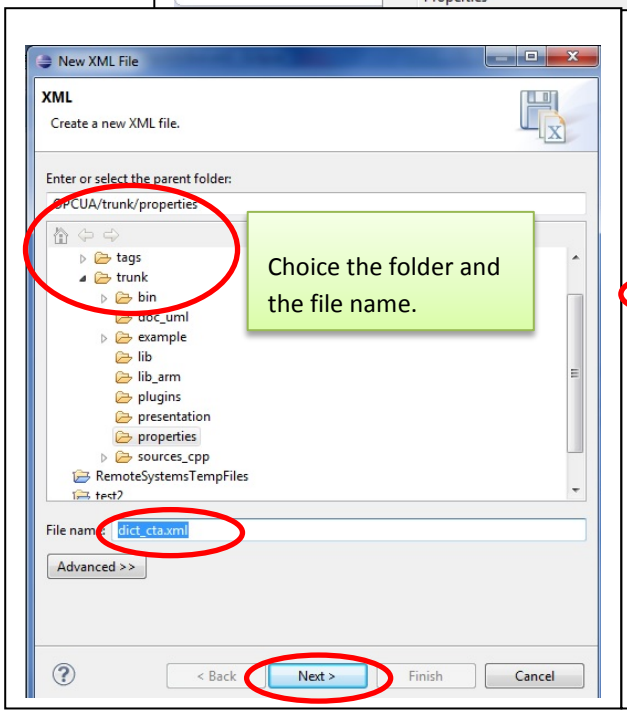
- Show and modify existing XML description file.

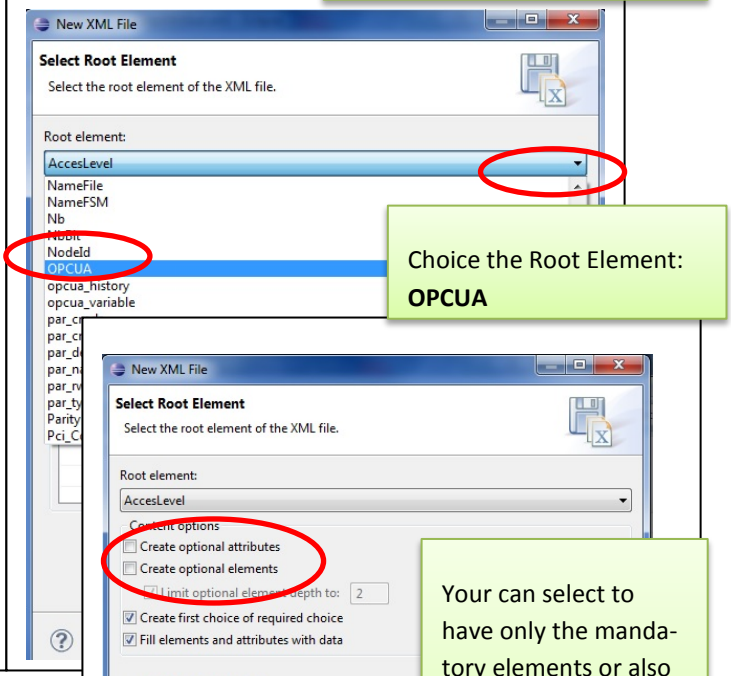


- Create new xml description file for your project.

Generate a new XML File for your project (using the good XSD dictionary file)

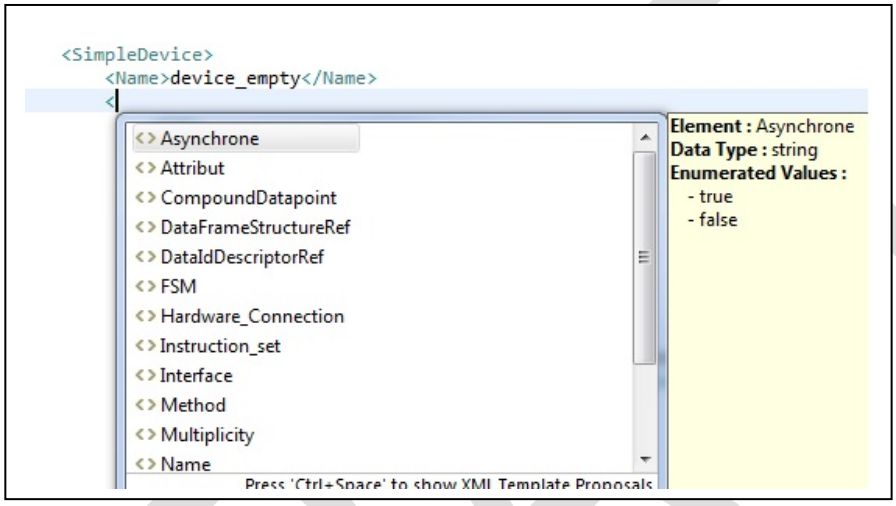




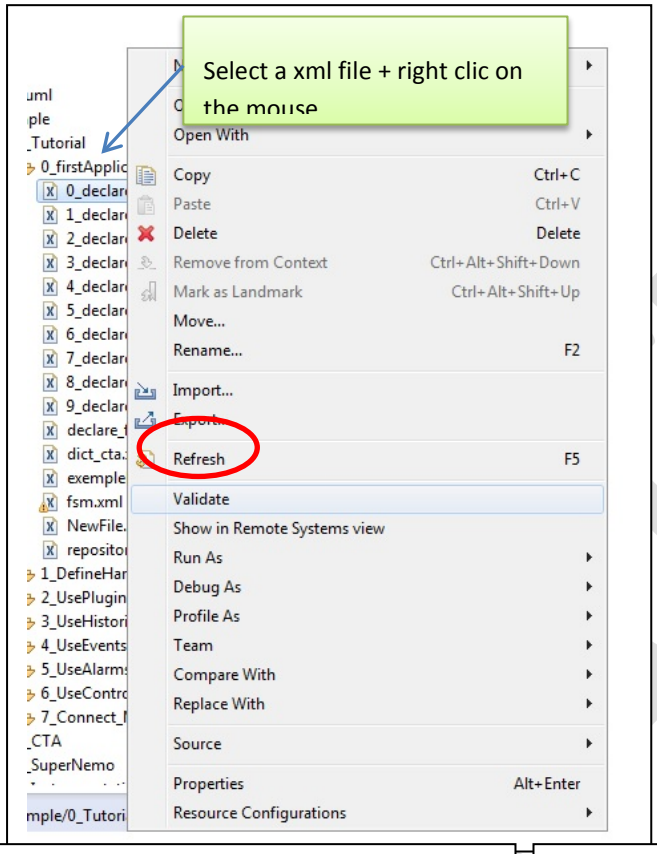


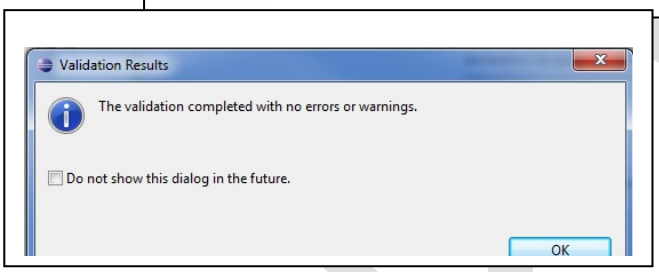
- Write a new tag in your xml file description.

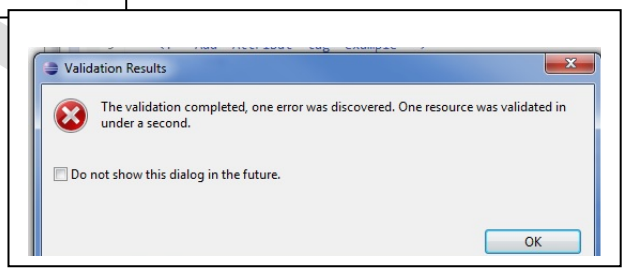
When you want write or modify your file, Eclipse help you and show you automatically the different tags possible.

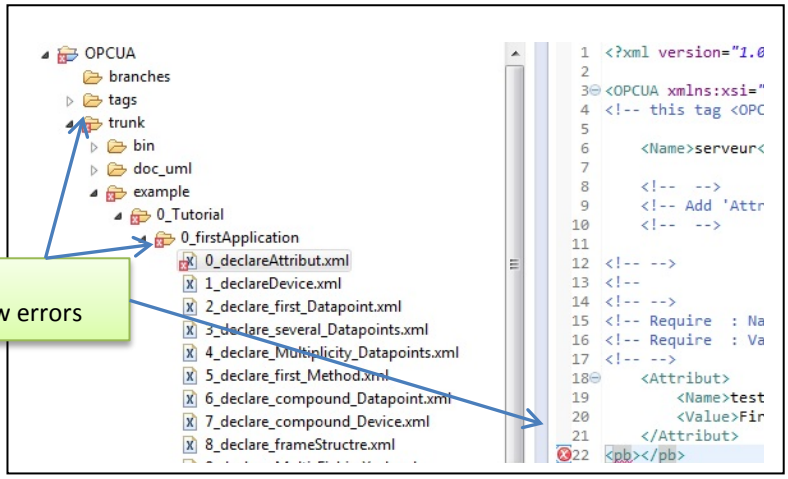


- Verify and validate XML files (file or folder) manually.









APPENDIX C : QUICK START USER GUIDE

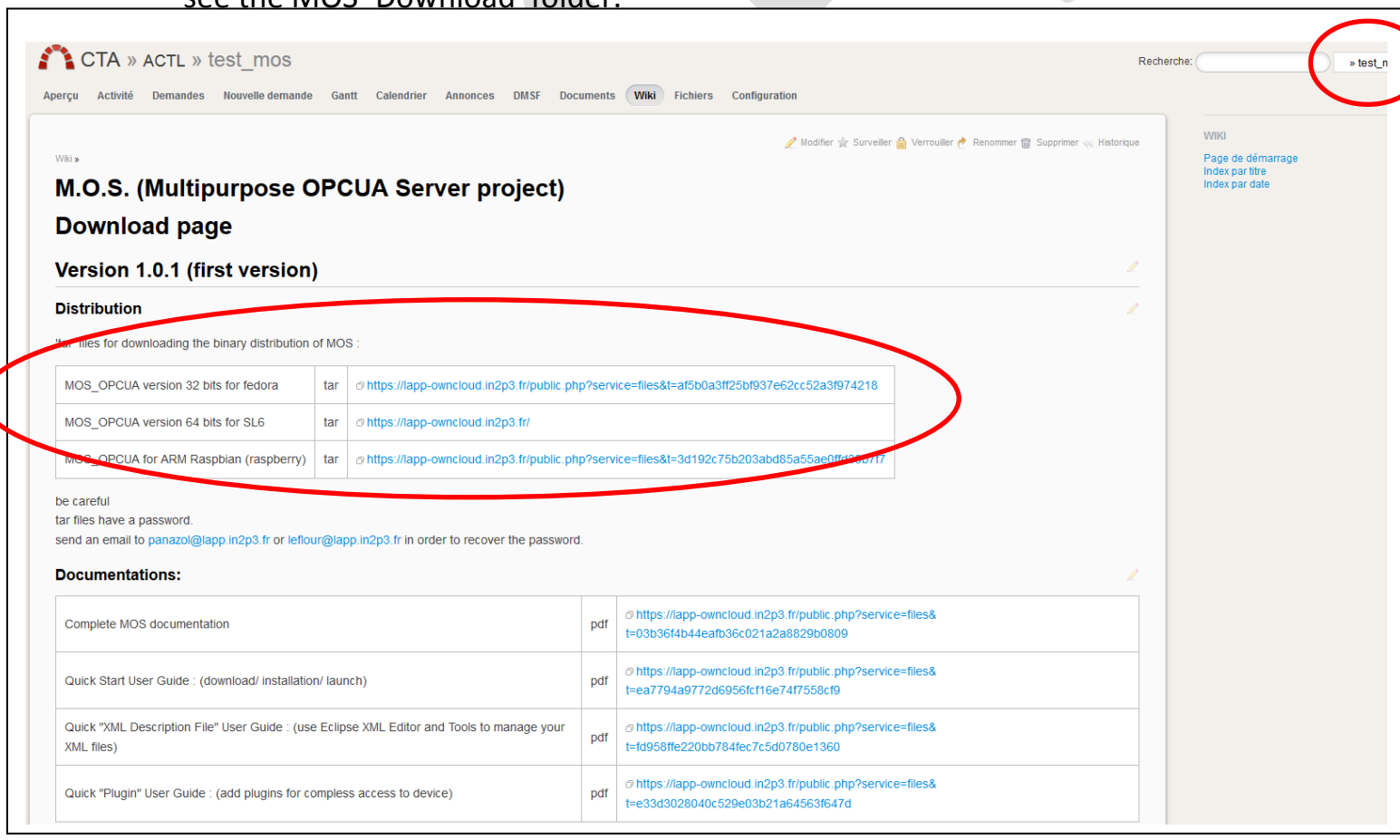
1. DOWNLOAD

You can download different binary application of “MOS” (Multipurpose OPCUA Server) for :

- Plateform Arm (be careful with HF library “HardFloat”)
- Plateform X86 32 bytes for fedora distribution linux
- Plateform X86 64 bytes for SL6 distribution linux

Here the address to download the zip tar file :
https://forge.in2p3.fr/projects/test_mos/wiki

see the MOS_Download folder.



2. INSTALLATION

Unzip the file

- gunzip OPCUA_<architecture>_<version>.tar.gz
- tar –xvf OPCUA_<architecture>_<version>.tar

- **Result :**
  - Application in <INSTALL_OPCUA_DIR>/bin
  - Libraries in <INSTALL_OPCUA_DIR>/lib
  - Plugin libraries in <INSTALL_OPCUA_DIR>/plugins
  - Example of XML description files <INSTALL_OPCUA_DIR>/example
- **Result :**
  - Application in <INSTALL_OPCUA_DIR>/bin
  - Libraries in <INSTALL_OPCUA_DIR>/lib
  - Plugin libraries in <INSTALL_OPCUA_DIR>/plugins

3. LAUNCH

You need to enable the ipv6

>sudo modprobe ipv6

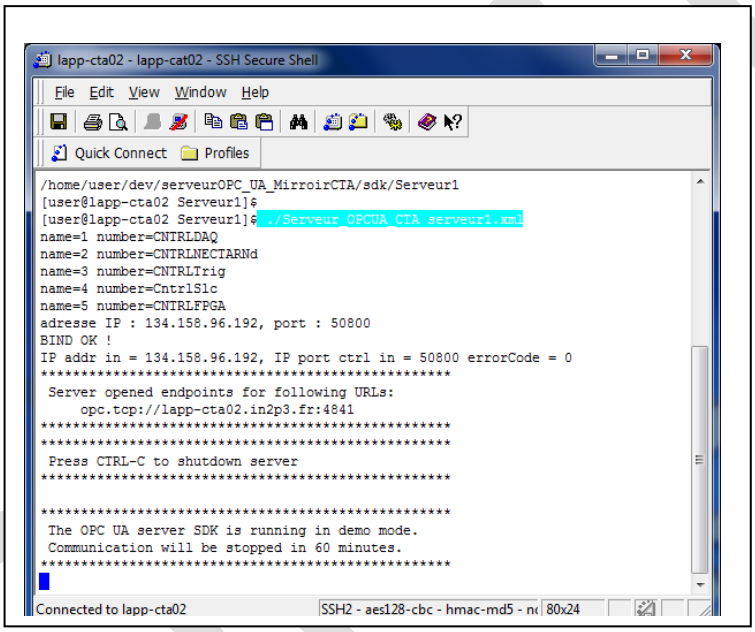
- **Start OPCUA with in command line**
  - cd <INSTALL_OPCUA_DIR>/bin/
  - ./ MOS_server –d <Description XML file>

Example :

./MOS_Server -d ../example/0_Tutorial/0_firstApplication/2_declare_first_Datapoint.xml

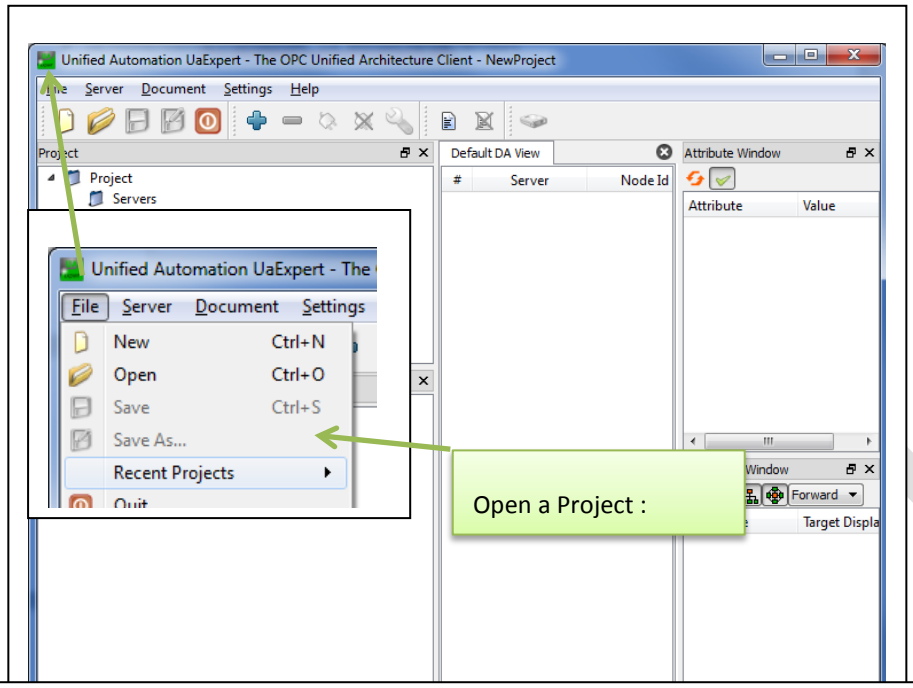
- **Arguments**

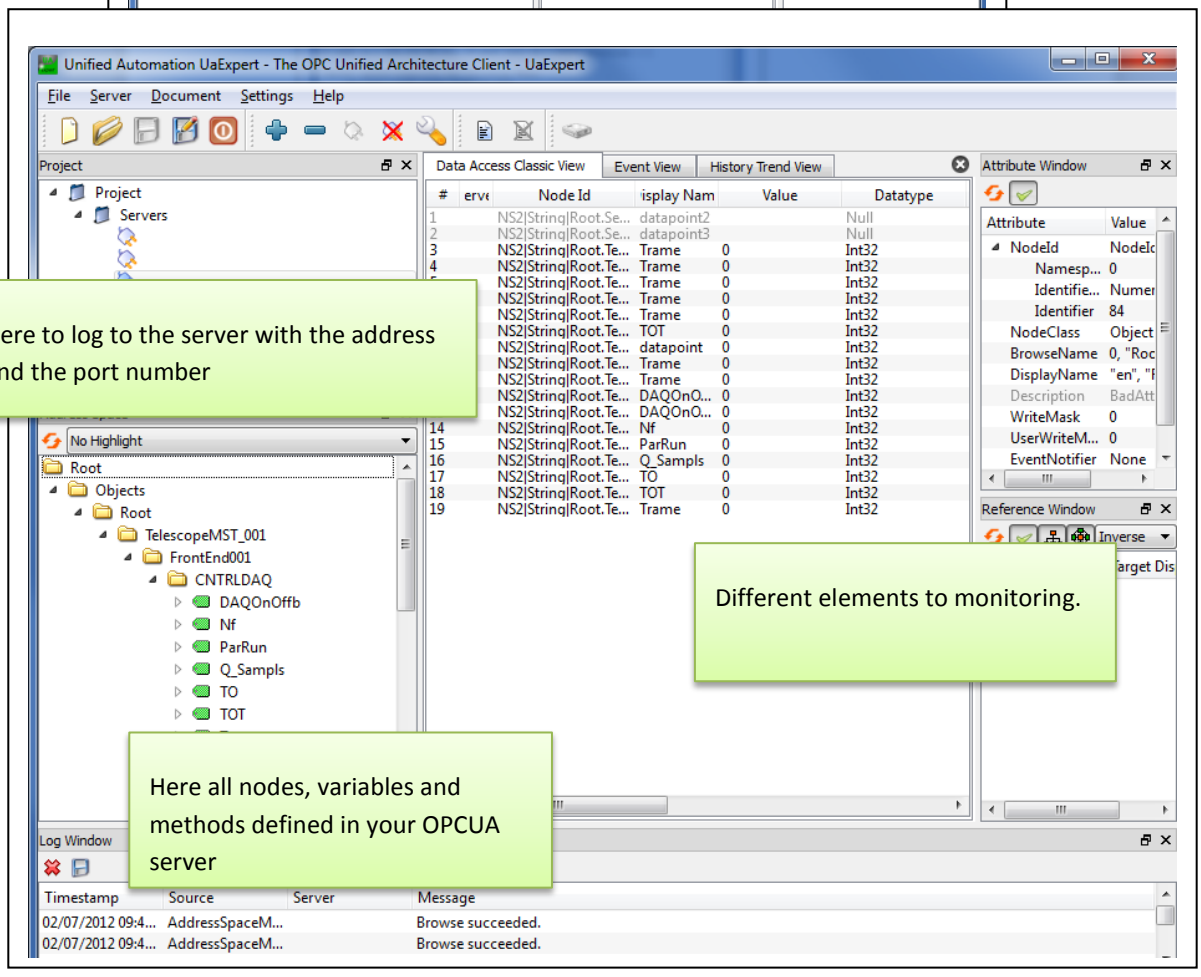
-d : (mandatory) : Description file
-r: (optional) : Repository server address. Allow to store into a central server this instance of OPCUA server.
-c: (optional) : Configuration element for the server (port number, max clients, …) (default = ../properties/ServerConfig.xml)



4. VERIFICATION WITH A GENERIC CLIENT

- Use UaExpert





APPENDIX D :QUICK PLUGIN USER GUIDE

1. INTRODUCTION

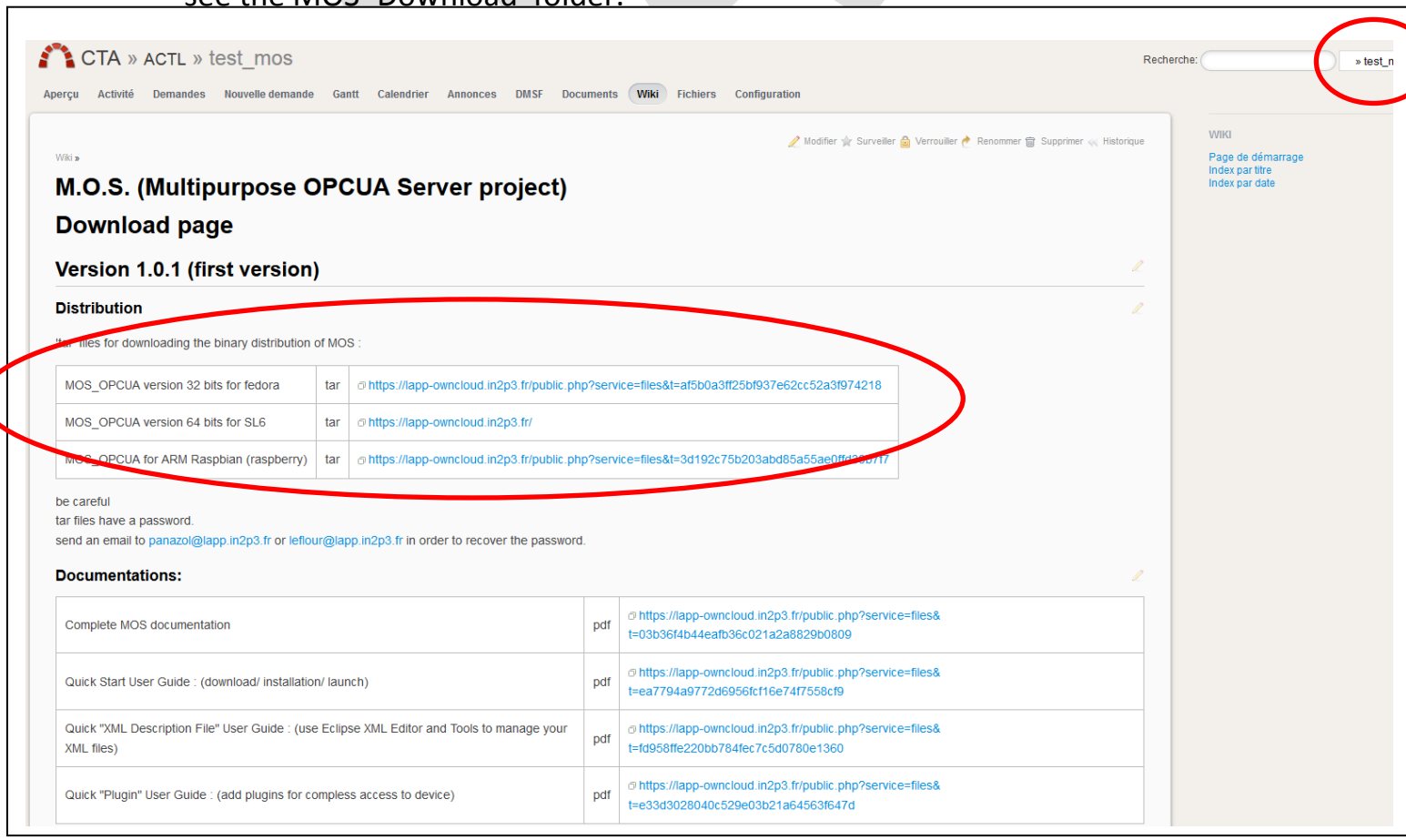
With the “generic OPUCA Server” application you can access to different device with different elements (USB/TCP/UDP/SERIAL etc…) with simple protocols.
But sometimes it’s possible that you need to use e specific library or driver to communicate with your device.
That why it’s possible to add a plugin program to access your device.

This document shows you how to create and add a plugin to access to your device.

2. DOWNLOAD SQUELETON PLUGIN SOURCE

https://forge.in2p3.fr/projects/test_mos/wiki

see the MOS_Download_folder.



3. INSTALLATION & FIRST COMPILATION

Installation with tar files

- gunzip Plugin_squeleton.tar.gz
- tar –xvf Plugin_squeleton.tar
- cd Plugin_squeleton

Compilation : (read README.txt)

Using MAKE

> make

Using CMAKE

Using a separate build directory:

>cd build

>cmake -DCMAKE_INSTALL_PREFIX=../.. ..

>make

---

4. PREPARE THE "SQUELETON PLUGIN" FOR YOUR PLUGIN (CHANGE THE NAME OF THE PLUGIN)

- Change the name of the folder.

```
>mv Plugin_squeleton Plugin_<your plugin name>
```

- Change in makefile files

```
> cd  Plugin_<your plugin name>

>vi makefile

Replace the squeleton by your pluginName

>:%s/squeleton/<your pluginname>/

>:wq

>vi CMakeLists.txt

Replace the Squeleton by your pluginName

>:%s/Squeleton/<your pluginname>/

>:%s/squeleton.cpp/<your pluginname code>.cpp/

>:wq
```

- Change the name of squeleton.cpp squeleton.h

```
>mv squeleton.cpp < your plugin name >.cpp

>mv squeleton.h <your plugin name>.h
```

```
vi <your plugin name>.h

Replace SQUELETON_H_ by <YOUR PLUGIN NAME_H_>

Replace "squeleton" by <your plugin name>


vi <your plugin name>.cpp

Replace "squeleton" by <your plugin name>
```

5. INTEGRATE YOUR CODE IN THE GOOD PLUGIN FUNCTIONS

>They are 3 functions in the plugin, who allow you to communicate between the OPCUA server and your code.

set() : This function is called when a client calls the method set() in a datapoint. The OPCUA server adds the variable define in the datapoint and pass the information on the plugin function “set()”

get() : This function is called when a client calls the method get() in a datapoint. The OPCUA server call the plugin function “get()” and puts the result into the variable define in the datapoint.

cmd() : : This function is called when a client calls others methods.

Example:

- Temperature (datapoint)
  - temperature_v (data associated at this datapoint)
  - set() (Method associated at this datapoint)
    When a client call the set() method
    1. The plugin function set() is called with the parameter “temperature_v”

  - get()(Method associated a this datapoint)
    When a client call the “get()” method.
    2. The plugin function get() is called
    3. The result is put to the variable “temperature_v”

  - init()(Method associated a this datapoint)
    When a client call the “init()” method :
    The plugin function cmd() is called.

Conclusion: They are 3 functions cmd(), set() and get(). For set() and get() functions, they are different sub functions for each type of parameter

- Set/get(result byte)
- Set/get(result short int)
- Set/get(result int)
- Set/get(result long)
- Set/get(result float)

- Set/get(result string)

The OPCUA server calls the good get()/set() plugin function according of the type of the variable (define in the description XML file)

Example:
"Temperature_v" is a float variable, so the method get() calls the plugin function "get(**result float**)"

APPENDIX E :CROSS COMPILER USER GUIDE (FOR ARM)

1. INTRODUCTION