UNIVERSITÀ degli STUDI di CATANIA

DIPARTIMENTO DI INGEGNERIA ELETTRICA, ELETTRONICA E INFORMATICA

PH.D. COURSE IN SYSTEMS, ENERGY, COMPUTER AND TELECOMMUNICATION ENGINEERING
XXXI CYCLE

Ph.D. Thesis

## ENHANCING INTEROPERABILITY IN INDUSTRY 4.0

MARCO STEFANO SCROPPO

Coordinator
Prof. Ing.
P. ARENA

Supervisor
Prof. Ing.
S. CAVALIERI

*...in everliving memory of my beloved grandmother M., recently passed away. I really miss you so much...*

# CONTENTS

# LIST OF FIGURES

ix

# LIST OF TABLES

# LISTINGS

# ONE

# INTRODUCTION

Since few years, Industry has been featured by a revolution, the fourth one. The fourth industrial revolution has been coined with different names in different countries; the most known is *Industry 4.0*. It features the application of modern Information & Communication Technology (ICT) concepts, such as *Internet of Things* (IoT) [1], in industrial contexts to create more flexible and innovative products and services leading to new business models and added value [2, 3]. The emerging *Industrial Internet of Things* (IIoT) [4] is one of the main results of this revolution.

Realisation of this novel vision may be achieved only if a big effort is really put to make interoperable the interchange of information between different industrial applications [5]. In order to provide for interoperability, definition and adoption of communication standards are of paramount importance [6]. For this reason, during the last few years, different organisations have developed reference architectures

to align standards in the context of the fourth industrial revolution.

Among the various standards taken in consideration by these reference architectures, *OPC UA international standard* (IEC 62541) [7, 8] seems to be one of the leading candidates to become a reference standard in this new era.  As for example, the *Reference Architecture Model for Industrie 4.0* (RAMI 4.0) [9], which defines a three-dimensional matrix that can be used to position existing standards in the fourth industrial revolution, indicates for OPC UA the role to standardise machine-to-machine communication.  Another example is the *Industrial Internet Reference Architecture* (IIRA), which is a standards-based open architecture defined by the Industrial Internet Consortium (IIC) [10].  The objective of IIRA is to create a capability to manage interoperability, to map applicable technologies, and to guide technology and standards development.  Also in this case, OPC UA plays a strategic role as it is one of the core connectivity standards inside IIRA.

Even though OPC UA already combines features coming both from industrial and ICT contexts, several activities have been carried on during these last years.  These activities have the aim to introduce ICT enhancements into OPC UA in order to further improve its usability inside the fourth industrial revolution and, in particular, in the IIoT. As for example, literature presents some proposals to combine OPC UA and *REpresentational State Transfer* (REST) architecture style [11]; in particular, the usage of *RESTful Web Services* [12] to access OPC UA is widely described [13, 14, 15].  This is due to many advantages of RESTful Web Services in industrial settings as shown in [13, 16] and to its usage in the context of IoT architectures [17].

Interoperability is an imperative requirement also in the IoT, as

no universal language exists for the Internet of Things. In the IoT market, device makers have to choose between either using different frameworks (e.g., provided by Apple, Amazon, or Google), which limit their market share, or developing their own framework for standardisation across multiple IoT ecosystems, which increases their costs. This might also create a challenge for end users to ensure that the products they use are compatible with the ecosystem they have acquired or alternately, to find some other ways to integrate their devices into the network. To overcome this, industry players have come together to form associations/foundations and consortiums of standards around the various IoT components, including connected buildings, connected home and industry IoT.

*Open Connectivity Foundation* (OCF) is one of the biggest industrial connectivity standards organizations for IoT [18]. It is an industry group whose aim is to develop specification standards to ensure a set of secure interoperability guidelines for consumers, businesses, and industries by delivering a standard communication platform and providing a certification program for devices associated with the IoT. Very recently, OCF has defined a set of specifications which leverage existing industry standards and technologies, provide connection mechanisms between devices and between devices and the cloud, and manage the flow of information among devices, regardless of their form factors, operating systems, service providers or transports. At this moment, OCF specification are under ISO/IEC standardisation under the project FDIS 30118 by ISO/IEC JTC1 Information Technology committee.

As pointed out in this introduction, one of the main goal of Industry 4.0 is the interoperability of industrial applications and the use

of enabling ICT technologies such as IoT. Due to the presence of different communication protocols in the current Industry 4.0 and IoT scenarios, interoperability may be achieved only through integration of these protocols. This integration must enable information exchange between the several industrial applications built on them. Integration of different protocols can be realised in several ways achieving various levels of interoperability according to the main features of the integration itself [19]. As for example, interoperability between two different ecosystems can be achieved by mapping the data model structures of the ecosystems to be integrated.

Taking into account OPC UA, current literature present several examples of interoperability between this standard and other protocols. Among them, [20] describes a solution for enabling interoperability between OPC UA and DPWS. Another example is the draft version of the mapping between OPC UA and DDS [21] (which is another core standard of IIRA [10]).

Given these premises and acknowledged the importance of OPC UA in this fourth industrial revolution, the research carried out during the Ph.D. course and described in this thesis was focused on the investigation about the enhancement of interoperability in Industry 4.0, IoT and IIoT. The research aim was to propose interoperability improvements based on OPC UA and, in particular, on the mapping of its information model in a different ecosystem.

First of all, the research was focused on the realisation of a proposal enabling the interoperability between OPC UA and generic users not compliant with OPC UA standard; in particular, it has been assumed to propose a way to allow the interaction between OPC UA and devices or applications using web technologies and without any knowledge of

the standard. The realization of this integration has been achieved through the definition of a novel data model mapping the OPC UA Information Model, based on common web data-formats (e.g. JSON). In the following, this proposal will be called *Integration between OPC UA and the Web*.

After that, the research focus has been enlarged to propose a solution enabling the interoperability between OPC UA and IoT/IIoT ecosystems. Among the current IoT/IIoT ecosystems, OCF has been chosen for the integration with OPC UA, as it seems a promising solution to standardise the exchange of information into IoT as explained before. The solution mainly aims to realise a mapping between OPC UA and OCF information models, called in the following *Integration between OPC UA and OCF*. Through this mapping, information maintained by an OPC UA Server may be used to populate a device compliant to OCF specifications which acts as a server, allowing it to expose this information to whatever client device in the OCF ecosystem. Vice versa, information maintained by an OCF Device may be published by an OPC UA Server allowing to make this information available to whatever OPC UA-compliant device.

The thesis is organized as follows. First of all, an overview about the fundamental elements used in this research is provided. In particular, the OPC UA standard is described in Chapter 2, the JSON data format is analysed in Chapter 3 and the OCF ecosystem is described in Chapter 4. After these overviews, the results of the research will be provided: Chapter 5 describes the *Integration between OPC UA and the Web* and Chapter 6 describes the *Integration between OPC UA and OCF*.

CHAPTER

# TWO

## OPC UA - IEC 62541

The aim of this chapter is to deepen some features of the OPC UA international standard (IEC 62541) needed to understand the content of the thesis. OPC UA is mainly based on a Client/Server communication model. Very recently, an extension of OPC UA called PubSub [22] has been released and defines a communication model based on Publish/Subscribe pattern [23] in addition to the Client/Server one.

## 2.1 OPC UA AddressSpace and Information Model

Inside an OPC UA Server, *OPC UA Nodes* are used to represent any kind of information, including variable instances and types. The set of OPC UA Nodes inside an OPC UA Server is called *AddressSpace* [24].

Inside the OPC UA AddressSpace, OPC UA Nodes may be organised into different subsets, called *Information Models* [25]; each of them is identified by a unique *Namespace URI*. An Array named *NamespaceArray* contains the URIs relevant to the Information Models of an OPC UA AddressSpace; each URI in a NamespaceArray is accessed through an integer index, called *NamespaceIndex*. Information Model relevant to index 0 is mandatory and refers to the native OPC UA Information Model. Each Node inside an Information Model is univocally identified by an *Identifier*. A Node inside an OPC UA AddressSpace is identified by an attribute named *NodeId*, which is a couple composed by a *NamespaceIndex* (ns) and an *Identifier* (i).

## 2.1.1 OPC UA NodeClass

An OPC UA Node belongs to a *NodeClass* which defines the attribute of the OPC UA Node. Each NodeClass is derived from the *Base NodeClass* which defines the common attributes of OPC UA Nodes, among which: *NodeId* (which identifies the OPC UA Node inside the OPC UA AddressSpace), *Description* (which is a localised textual description of the OPC UA Node) *BrowseName* (used to identify the OPC UA Node when browsing the OPC UA AddressSpace) and *DisplayName* (which contains the name of the OPC UA Node that should be displayed in a user interface). OPC UA defines the following NodeClasses:

- *Variable NodeClass*, used to model values of the system. Two types of Variable are defined: *Property* and *DataVariable*. A Property contains server-defined metadata characterising what other OPC UA Nodes represent. A DataVariable represents the

data of an OPC UA Object and it may be made up by a collection of other OPC UA DataVariable Nodes. The Variable NodeClass features the *Value* attribute containing its current value; another attributed called *DataType* contains the NodeId of a Node providing type definition for the Value attribute of the Variable Node.

- *VariableType NodeClass*, used to provide type definition for Variables. OPC UA standard defines the *BaseVariableType* which all the VariableTypes must be extended from. OPC UA already defines several standard VariableTypes derived from BaseVariableType. Among them there are the *BaseDataVariableType* and the *PropertyType*. The former is used to define a DataVariable Node, whilst the latter defines a Property Node.

- *DataType NodeClass*, used to provide type definition for the Value attribute of a Variable Node.

- *Object NodeClass*, used to represent real-world entities like system components, hardware and software components, or even a whole system. An OPC UA Object is a container for other OPC UA Objects, DataVariables and Methods. As the Object Node does not provide for a value, an OPC UA DataVariable Node can be used to represent the data of an Object. For example, an Object modelling a file uses a DataVariable to represent the file content as an array of bytes. As another example, function blocks in control systems might be represented as OPC UA Objects: the parameters of the function block (e.g., its setpoints) may be represented as OPC UA DataVariables. The Object

function block might also have properties that describe its execution time and its type.

- *ObjectType NodeClass*, used to hold type definition for OPC UA Objects. OPC UA defines the *BaseObjectType* which all the ObjectTypes must be extended from. OPC UA already defines several standard ObjectTypes derived from BaseObjectType. Among them there is the *FolderType ObjectType* to model hierarchy among OPC UA Nodes. Instances of FolderType ObjectType are used to organise the AddressSpace into a hierarchy of OPC UA Nodes; they represent the root Node of a subtree, and have no other semantics associated with them. As for example, OPC UA defines the OPC UA Node named *Objects* with NodeId given by NamespaceIndex=0 and Identifier=85 of FolderType ObjectType: it is the entry point for the entire set of Objects and Variables in an AddressSpace.

- *Method NodeClass*, used to model callable functions that initiate actions within an OPC UA Server. Methods are lightweight functions, whose scope is bounded by an owning OPC UA Object, similar to the methods of a class in object-oriented programming. Methods can have a varying number of input arguments and return resultant arguments. Each Method is described by a Node belonging to the Method NodeClass. This Node contains the metadata that identifies the arguments of the Method and describes its behaviour. OPC UA Method NodeClass features two boolean attributes named *Executable* and *UserExecutable*. The Executable attribute indicates if the Method is currently executable; it does not take any user access rights into account,

i.e. although the Method is executable this may be restricted to a certain user/user group. The UserExecutable attribute indicates if the Method is currently executable taking user access rights into account. OPC UA Method NodeClass features two OPC UA Properties named *InputArguments* and *OutputArguments*. The InputArguments Property is used to specify the arguments that shall be used by a client when calling the Method. The OutputArguments Property specifies the result returned from the Method call. In order to invoke a Method, it is necessary to use a specific OPC UA Service named Call in which actual parameters are passed [26].

- *View NodeClass*, used to allow OPC UA Servers to subset the AddressSpace into Views to simplify OPC UA Client access.

- *ReferenceType NodeClass*, used to define different Reference types. In the following, description of References will be given.

It is worth noting that only for the NodeClasses defining types, the boolean attribute *isAbstract* is present. A "true" value means that no instances of the type can be created and the type is called *abstract*: instances may exist only for the relevant subtypes. A type with a "false" value for this attribute is called *concrete*: instances of concrete types can be realised.

## 2.1.2 OPC UA Events

OPC UA also defines *Events*, which represent specific transient occurrences; system configuration changes and system errors are examples of Events. *Event Notifications* report the occurrence of an Event.

Events are not directly visible in the OPC UA AddressSpace. OPC UA Objects can be used to subscribe to Events. In particular, OPC UA defines a *Notifier* as an OPC UA Object that can be subscribed by OPC UA Clients to get Event Notifications.

## 2.1.3   OPC UA References

A relationship may be defined between two OPC UA Nodes and is called *Reference* [8, 24, 25]. References may be classified into: *Hierarchical* and *NonHierarchical*.

The semantic of a Hierarchical Reference is that it spans a hierarchy. Hierarchical References does not forbid loops. For example, starting from Node "A" and following Hierarchical References it may be possible to browse to Node "A", again. It is not allowed to have self-references using Hierarchical References. OPC UA foresees several Hierarchical Reference among which:

- *HasComponent* and *HasOrderedComponent.* If the source OPC UA Node is an Object, the target Nodes may be OPC UA Objects, DataVariables and Methods; the meaning is that the source Object is made up by the target OPC UA Nodes. If the source OPC UA Node is a DataVariable, the target Nodes may be other OPC UA DataVariables; the meaning is that the source variable is made up by a set of other variables.

- *HasProperty.* This Reference may connect a source OPC UA Node to a target OPC UA Property; the meaning is that the source Node features a property described by the target Node. In the following, in the case of a generic OPC UA Node linked to

an OPC UA Property Node by a HasProperty Reference, it will be said for short that the OPC UA Node features the Property related to the Property Node.
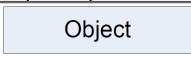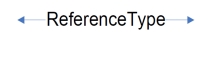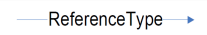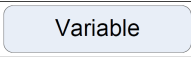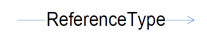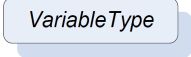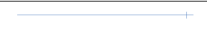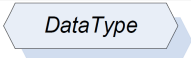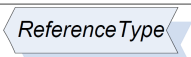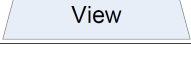
- *Organizes.* This Reference may connect a source OPC UA Object of FolderType ObjectType to other OPC UA Objects and/or Variables; the meaning is that the source Node organises (i.e. acts like a folder) the target Nodes.

- *HasChild.* It models a non-looping hierarchy between OPC UA Nodes.

- *Aggregates.* This reference is used to indicate that an OPC UA Node belongs to another OPC UA Node.

- *HasEventSource.* This reference may connect two OPC UA Nodes; the semantic of this reference is to relate Event sources to a Notifier in a hierarchical, non-looping organisation.

- *HasNotifier.* It relates Notifier OPC UA Objects, in order to establish a hierarchical organisation of event notifying Objects.

- *HasSubtype.* It expresses a subtype relationship of types.

NonHierarchical References do not span a hierarchy and should not be followed when trying to present a hierarchy. The NonHierarchical References are: *HasTypeDefinition*, *HasEncoding*, *HasDescription*, *HasModellingRule*, *HasParentModel* and *GeneratesEvent*. HasTypeDefinition is used to bind an OPC UA Object or Variable to its ObjectType or VariableType, respectively. The meaning of HasModellingRule, HasEncoding and HasDescription References will be ex-

13

plained in the remainder of this chapter.  Description of HasParent-
Model and GeneratesEvent References can be obtained by [8, 24].

### 2.1.4   OPC UA Graphical Notations

OPC UA specifications define graphical symbols to represent Nodes
and References [24]. They are shown by Figure 2.1a and Figure 2.1b,
respectively.



**(a)** *Graphical Notation of Nodes*   **(b)** *Graphical Notation of References*

**Figure 2.1:** *OPC UA Graphical Notations*

It is worth noting that instead of using the HasTypeDefinition
Reference to point from an Object or Variable to its ObjectType or
VariableType, the name of the TypeDefinition can be added to the
text used in the Node.  The TypeDefinition shall either be prefixed
with "::" or it is put in italic as the top line as shown in Figure 2.2.
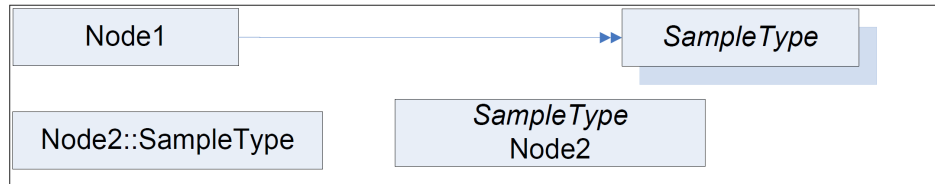
**Figure 2.2:** *Example of using a TypeDefinition inside a Node*

### 2.1.5   OPC UA DataType

The Value attribute of an OPC UA Variable is described by a DataType. OPC UA DataType may be *Built-in*, *Enumeration* or *Structured*.

Built-in DataTypes provide for base types like Int32, Boolean, Double; see [27] for the complete list of the Built-in DataTypes available.

Enumeration DataTypes are used to represent a discrete set of named values. The Value attribute is of Built-in Int32, i.e. an integer, which allows to identify the enumeration value. Enumeration values are maintained by the Enumeration DataType using two mutual exclusive OPC UA Properties named *EnumStrings* and *EnumValues*. The Value attribute of the EnumStrings Property Node is an array of LocalizedText (for a formal definition of this type see [24]) each of which represents the human-readable representation of an enumerated value; the integer representation of the enumeration value points to a position of the array. The Value attribute of the EnumValues Property Node is an array too and allows to represent enumerations with integers that are not zero-based or have gaps (e.g., 1, 2, 4). In this case each element of the array is a structure made up by the elements

shown by Table 2.1.

**Table 2.1:** *Structure of each element of the array relevant to Value attribute of EnumValues Property Node*

| Name | Type | Description |
| --- | --- | --- |
| Value | Int64 | The integer representation of an enumeration. |
| DisplayName | LocalizedText | A human-readable representation of the value of the enumeration. |
| Description | LocalizedText | A description of the enumeration value. |

Structured DataTypes represent structured data; they are the most powerful construct allowing to specify user-defined (i.e. vendor-specific) complex types. Generally, they are made up by a structure that may contain other Structured and/or Built-in DataTypes.

Transmission of information between OPC UA Client and Server may occur using OPC UA Binary, XML or JSON data encoding [27].

For Built-in DataTypes, the encodings are well defined by the OPC UA specification which defines standard data types giving the encoding rules corresponding to the Built-in DataTypes [25].

For Structured DataTypes, the encodings are exposed in the AddressSpace, so that OPC UA Clients can obtain them in order to decode or encode the data during the communication with the OPC UA Servers. For this reason, different encoding rules may be available for a Structured DataType, depending on the profile used for data transmission; as for example, *DefaultBinary* encoding is used for the binary data transmission (i.e. OPC UA Binary data encoding). In the following, details about representation of encoding of the Structured

DataType will be given considering only the DefaultBinary.

In order to provide the encoding rules, each DataType Node representing a Structured DataType points to an Object of *DataType-EncodingType* ObjectType representing the encoding for the specific data transmission (e.g. *DefaultBinary* Object in the case of binary data transmission).

Figure 2.3 shows an example of a vendor-specific Structured DataType named *MyType*. As it can be seen, this Node is linked to the DefaultBinary Object through an HasEncoding Reference.
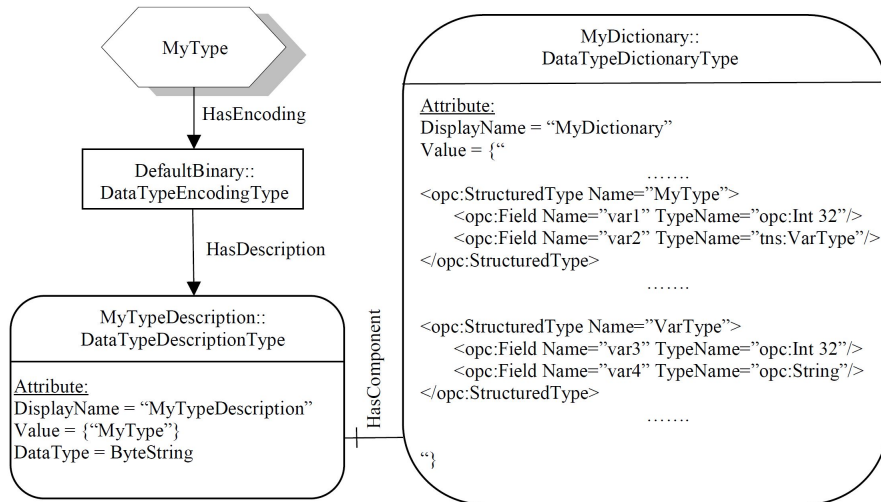


**Figure 2.3:** *Example of OPC UA Structured DataType description*

HasDescription Reference allows to link the DefaultBinary Object to an OPC UA Variable of the *DataTypeDescriptionType* VariableType, named in the example *MyTypeDescription*. Among its attributes, the Value contains the entry point of the Variable of

*DataTypeDictionaryType* VariableType which contains the description of the Structured DataType (i.e. its encoding). In the figure, this OPC UA Variable is named *MyDictionary*. HasComponent Reference is used to link the two variables MyTypeDescription and MyDictionary.

As shown by Figure 2.3, the Value attribute of the MyDictionary Variable contains one or more entries called *StructuredType*. A StructuredType contains in turn several *Field* elements. A Field refers to a component of the Structured DataType and features the attributes *Name* and *TypeName* describing the relevant component.

In the example shown by Figure 2.3, only the StructuredType relevant to MyType is shown. It is easy to understand that MyType is a structure having two elements named "var1" and "var2". The first element is of a Built-in DataType (i.e. Int32), and the second one is of a Structured DataType, named *VarType*, defined again in the same MyDictionary Variable as shown by Figure 2.3. As shown, this nested Structured DataType is made up by two elements: "var3" of Int32 Built-in DataType and "var4" of String Built-in DataType.

Array of elements belonging to Built-in, Enumeration and Structured DataTypes are allowed for the Value attribute of OPC UA Variable Nodes; for this reason, the Variable NodeClass defines two attributes named *ValueRank* and *ArrayDimensions* which are used to indicate whether the Value attribute of the Variable Node is a scalar or an array and, in the latter case, how many dimensions the array owns; ArrayDimensions is used to specify the maximum supported length for each dimension specified in ValueRank, only in case the Value attribute contains an array. It is worth noting that OPC UA allows only array values with elements of the same type, which is specified by the DataType attribute of the OPC UA Variable Node.

## 2.1.6   OPC UA ModellingRules

HasModellingRule Reference is used to describe how instances of types should be created. The source of this Reference is an *InstanceDeclaration*. An InstanceDeclaration is an Object, Variable or Method that references a *ModellingRule* Object with an HasModellingRule Reference and is the TargetNode of a Hierarchical Reference from a type Node or another InstanceDeclaration.

A ModellingRule Object specifies what happens to the InstanceDeclaration with respect to instances of the OPC UA type. For instance, a *Mandatory* ModellingRule for a specific InstanceDeclaration specifies that instances of the OPC UA type referencing the InstanceDeclaration must have a counterpart of that InstanceDeclaration. An *Optional* ModellingRule for a specific InstanceDeclaration, instead, specifies that instances of the OPC UA type may have a counterpart of that InstanceDeclaration but it is not required. Mandatory and Optional ModellingRules require that the counterpart of the InstanceDeclaration has the same BrowseName of the InstanceDeclaration.

Other two ModellingRules exist named *MandatoryPlaceholder* and *OptionalPlaceholder*. The differences with the previous ModellingRules is that the counterparts of InstanceDeclaration may be more than one, regardless of the BrowseName of the InstanceDeclaration. For this reason, the BrowseName of InstanceDeclarations having the OptionalPlaceholder and MandatoryPlaceholder ModellingRule will be enclosed within angle brackets (e.g. <InstanceDeclarationNames>). Furthermore, in the grapichal representation of OPC UA Nodes used in this thesis, a ModellingRule Object and rel-

evant HasModellingRule Reference are represented inside the source InstanceDeclaration Node only by the kind of ModellingRule Object in square brackets (i.e. [Mandatory], [Optional], [OptionalPlaceholder], [MandatoryPlaceholder]). Figure 2.4 shows an example of the graphical notation just described.
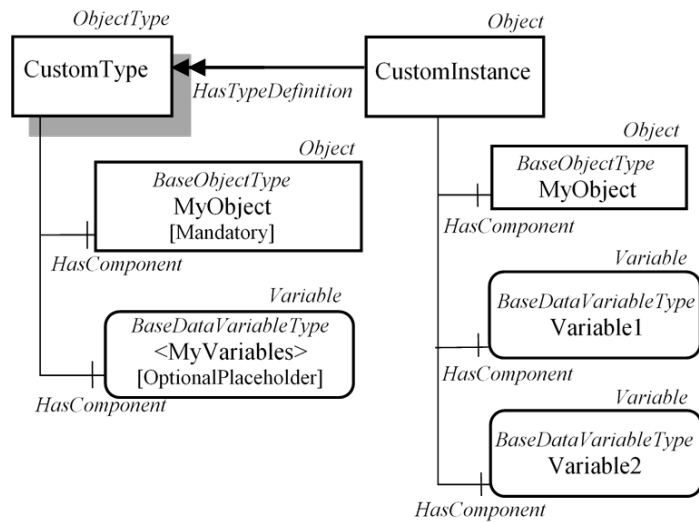


**Figure 2.4:** *ModellingRule graphical notation adopted*

## 2.2   OPC UA Services

OPC UA offers many services to allow an OPC UA Client to access to the AddressSpace of OPC UA Servers [26].

### 2.2.1   OPC UA Discovery Service Set

OPC UA Servers may register themselves with a *Discovery Server* using the OPC UA *RegisterServer Service*. OPC UA Clients can later discover any registered Servers by calling the OPC UA *FindServers Service* on the Discovery Server [26].

### 2.2.2   OPC UA Session Service Set

Once an OPC Client has found a specific OPC UA Server, it has to create a *Session* in order to exchange data with the Server [8]. OPC UA Client can establish a Session with the OPC UA Server by using the OPC UA *CreateSession Service*. Once a Session has been established, it must be activated using the OPC UA *ActivateSession Service* [26].

### 2.2.3   OPC UA Browse, Read and Write Services

Once a Session has been created and activated, access to the AddressSpace maintained by the OPC UA Server can occur.

The simplest way to allow an OPC UA Client to explore the AddressSpace of an OPC UA Server is using the OPC UA *Browse Service* [26]; given a particular OPC UA Node it allows to discover the References and the relevant target Nodes.

OPC UA *Read* and *Write Services* allow an OPC UA Client to access to a specific OPC UA Node inside a Session [8, 26].

The *Read Service* is used to read one or more attributes of one or more Nodes. The OPC UA Client specifies a list of Nodes and the relevant attributes to be read. The function returns a list of the attribute values read. Other elements are passed together with each attribute value; among them, there is the *StatusCode*. The StatusCode is produced by the OPC UA Server to indicate the conditions under which an attribute value was generated, and thereby can be used as an indicator of the usability of the value. OPC UA defines three values for the StatusCode (which contain a *SubCode*): *Good* (it assures that the value is reliable), *Uncertain* (it indicates that the quality of the value is uncertain for reasons indicated the SubCode), *Bad* (it means that the value is not usable for reasons indicated by the SubCode). Values that have an uncertain status associated with them shall be used with care since these results might not be valid in all situations. Values with a bad status shall never be used.

The *Write Service* is used to write values to one or more attribute of one or more Nodes.

## 2.2.4   OPC UA Subscription and MonitoredItem Service Sets

*Subscriptions* and *MonitoredItems* represent a more sophisticated way to exchange data from OPC UA Server to OPC UA Client. They allow an OPC UA Client to receive cyclic updates of OPC UA Variable values, Node attributes and Events [8, 26].

A Subscription is the context needed to realise this cyclic exchange;

it is associated to a OPC UA Session and is created using the *Create-Subscription Service.*

MonitoredItems are created using the OPC UA *CreateMonitored-Items Service.* Each MonitoredItem identifies the item (i.e. Variable value, Node Attribute or OPC UA Event) to be monitored and the Subscription to use to send *Notifications* when a change is detected. The content of a Notification depends on the changes detected; for example, in the case of changes of OPC UA Variable value, the Notification contains the new value updated. Notifications are put in a queue defined inside each MonitoredItem. Size and queuing policy may be defined by the OPC UA Client for each MonitoredItem queue. Notification are packaged into *NotificationMessages* for transfer to the OPC UA Client.

MonitoredItems have several settings among which there is the *Sampling Interval* which defines the rate at which it checks for changes in the Variable values, Node attributes and Events. Only for the OPC UA Variable Node, an attribute called *MinimumSamplingInterval* is defined specifying the lower bound value that the sampling interval can assume for each single Variable value.

Considering a MonitoredItem associated to an OPC UA Variable, in order to detect a Variable value change, several filters are available. Among them, the *DataChangeFilter* exists. DataChangeFilter features an important parameter, called *trigger*, which specifies the conditions under which a Notification must be produced. The default value of the trigger parameter is that a Notification is produced if a change in the Variable value or a change in the associated Status-Code is detected. Changes in the value may be generic (i.e., whatever changes), or it may be specified by another parameter of the

DataChangeFilter, called *DeadbandType*; it may assume three possible values: *None* (i.e., no rule is defined for the changes of Variable values), *AbsoluteDeadBand* and *PercentDeadband*. According to the AbsoluteDeadband [26], a change in the OPC UA Variable value is detected when the absolute value of the difference between the last value (that stored in the queue) and the current value of the OPC UA Variable is greater than a parameter called AbsoluteDeadband, fixed by the OPC UA Client. In other terms, a Notification is produced if (2.1) is verified.

$$|lastcachedvalue - currentvalue| > AbsoluteDeadband \qquad (2.1)$$

The AbsoluteDeadband can be applied only to Variable of Numeric type (i.e., Integer, Unsigned Integer, Float and Double).

The PercentDeadband [28] can be applied only for OPC UA Variables of Numeric Type for which a range of the possible values has been defined inside the OPC UA Server. This range is called EU-Range. In the case of Numeric Variable with an EURange, the OPC UA Client specifies a *deadbandValue* and a Notification is produced if (2.2) is verified.

$$|lastcachedvalue - currentvalue| > \frac{deadbandValue}{100} * EURange$$
$$(2.2)$$

Each Subscription features a *PublishingInterval*, which defines the interval at which the OPC UA Server clears all the MonitoredItem queues contained in the Subscription and conveys their contents (i.e., Notifications) into a NotificationMessage to be sent to the OPC UA Client.

Transmission of NotificationMessages by OPC UA Server is triggered by *Publish requests* sent by OPC UA Client [8, 26]. The OPC UA Server enqueues all the Publish requests received until a NotificationMessage is ready (according to the PublishingInterval, as said before). When this occurs, the NotificationMessage is sent back to the OPC UA Client through a *Publish response*. For each Publish request sent by the OPC UA Client, exactly one NotificationMessage is transmitted by the OPC UA Server through a Publish response.

## 2.3   OPC UA Security

OPC UA gives a lot of importance to the secure communication between OPC UA Client and Server. A *Secure Session* must be created on the top of a *Secure Channel*, which may be featured by different levels of security, through the choice of the so-called *Endpoints* [8, 29].

Secure Channel is established to guarantee confidentially, integrity and application authentication. When a Secure Channel has been established, a Secure Session may be created on its top between OPC UA Client and Server to guarantee user authentication and authorization.

Creation of both Secure Channel and Secure Session is based on the use of X.509 certificates issued by a certification authority (CA). Among these certificates, there is the *OPC UA Application Instance Certificate*, which identifies the installation of an OPC UA product inside an OPC UA Client and Server. It is used during the creation of a Secure Channel.

Each OPC UA Server offers several Endpoints, each of which groups different attributes. Among them, there are: the *URL* of the

Endpoint, the *Application Instance Certificate* of the OPC UA Server, the *Security Policy* (which is the set of algorithms available for the implementation of the secure mechanisms for confidentially and/or integrity like Basic128Rsa15 or Basic256) and the *Security Mode* (which may enable both digital signature and encryption mechanisms, only digital signature mechanism or none of them). Before the creation of the Secure Channel, the OPC UA Client chooses the Endpoint of the OPC UA Server with the desired features. For example, let us consider an OPC UA Client who wants to use both digital signature and encryption, and let us assume that he wants to realise the digital signature with sha1 algorithm and aes128 for encryption [29]. In this case, the OPC UA Client must look for an Endpoint featuring Basic128Rsa15 as Security Policy, and a Security Mode which enables both digital signature and encryption (it is important to recall that Basic128Rsa15 is a suite of security algorithms that include aes128 for encryption and sha1 for authentication). In order that an OPC UA Client may discover the available Endpoints of a specific OPC UA Server, OPC UA *GetEndpoints Service* can be used [26]; it gives the list of the Endpoints and the relevant features available for a particular OPC UA Server.

OPC UA Client create a Secure Channel using the OPC UA *OpenSecureChannel Service*, specifying the URL of the preferred Endpoint [8, 26]; this service also allows the OPC UA Client to send its Application Instance Certificate to the Server. If the certificate is approved by the OPC UA Server, the Secure Channel is created. It is a logical connection between a single OPC UA Client and a single OPC UA Server, which maintains a set of keys known only to the Client and Server; these keys are used to authenticate and encrypt messages

sent across the network.

After Secure Channel has been established, OPC UA Client starts to establish a Secure Session using the services described in subsection 2.2.2. First of all, the Session is created using the OPC UA CreateSession Service. Once the Session has been established, the OPC UA Client uses the OPC UA Service ActivateSession to send the user credentials to the Server. It is worth noting that user credentials may be represented by another certificate or by a pair username/password.

## 2.4   OPC UA for Devices

In the current automation systems, devices from many different manufacturers must be integrated resulting in effort for installation, version management and device operation. This challenge can be faced best with an open and standardised device model. For this reason, OPC UA has defined a specification called *OPC UA for Devices - Companion Specification* [30], to define the information model associated with Devices; it is made up by three models which build upon each other:

- The (base) *Device Model* is intended to provide a unified view of devices irrespective of the underlying device protocols.

- The *Device Communication Model* adds Network and Connection information elements so that communication topologies can be created.

- The *Device Integration (DI) Host Model* allows reflecting the topology of the automation system with the devices as well as the connecting communication networks.

In the following, an overview of the Device Model will be given in order to better understand the content of the thesis.

### 2.4.1  OPC UA Device Model

The Device Model defines several elements, as shown by Figure 2.5 taken from the OPC UA Specification [30].
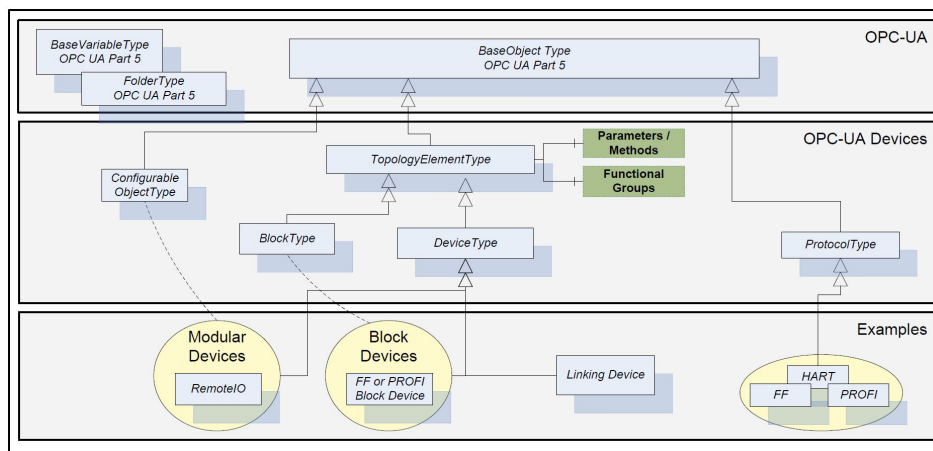


**Figure 2.5:** *OPC UA Device Model*

In the following, only those necessary for the proper understanding of the thesis will be deepened.

#### 2.4.1.1  TopologyElementType ObjectType

*TopologyElementType* is the base ObjectType for elements in a device topology. This ObjectType is abstract and defines the basic information components for all configurable elements in a device topology. Figure 2.6 shows the TopologyElementType component.
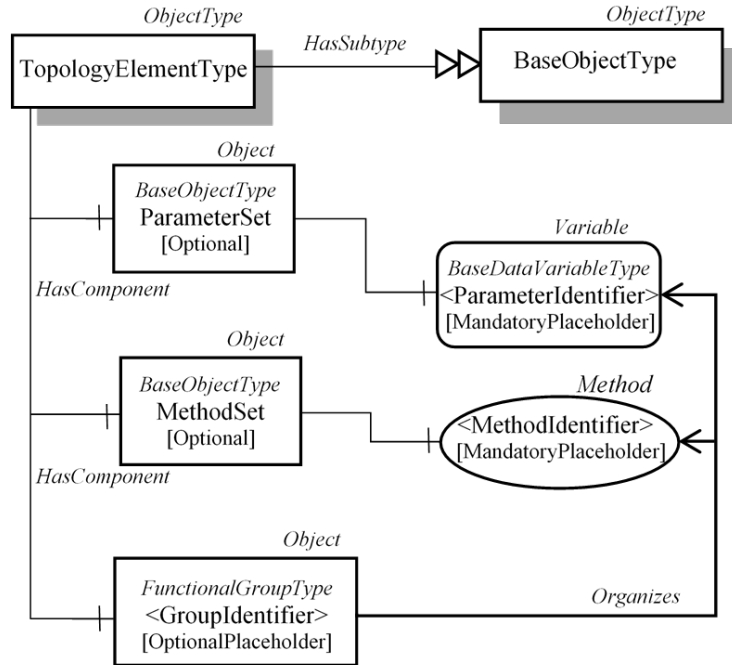
**Figure 2.6:** *OPC UA TopologyElementType ObjectType*

All elements in a topology may have *Parameters* and *Methods*. Parameters are modelled with OPC UA DataVariable nodes. If such an element has Parameters they are kept in an Object called *ParameterSet* as a flat list of Parameters. If it has Methods they are kept the same way in an Object called *MethodSet*. Both ParameterSet and MethodSet are target of a an HasComponent Reference starting from the TopologyElement Object.

*FunctionalGroups* can be used to organise the Parameters and Methods to reflect the structure of the TopologyElement. A FunctionalGroup Node is an instance of the *FunctionalGroupType* Object-

Type, a subtype of FolderType ObjectType. TopologyElements may have an arbitrary number of FunctionalGroups to organise Parameters and Methods. As for example, a FunctionalGroup called *Identification* shall be used to organise Parameters for identification of this TopologyElement. It is worth noting that the same Parameter or Method might be referenced from more than one FunctionalGroup.

### 2.4.1.2   DeviceType ObjectType

The abstract *DeviceType* ObjectType provides a general type definition for any Device. It is a subtype of TopologyElementType ObjectType. A Device Node (i.e. an instance of a concrete subtype of DeviceType) may have Parameters, Methods, and FunctionalGroups as defined for the TopologyElementTypeDevices.

DeviceType defines several OPC UA Properties, providing a way for a Client to get common Device information, among which: *SoftwareRevision* (which provides the revision level of the software/firmware of the device), *Model* (which provides the model name of the Device) and *Manufacturer* (which provides the name of the company that manufactured the device).

### 2.4.1.3   ConfigurableObjectType ObjectType

*ConfigurableObjectType* is used as a general means to create modular topology units. If needed, an instance of this type will be added to the head object of the modular unit.

OPC UA specification [30] defines a generic pattern to expose and configure components, named *Configurable Component* pattern. This pattern is based on the following principles:

- A ConfigurableObject shall contain a folder called *Supported-Types* that references the list of subtypes of BaseObjectType available for configuring components. The Types are referenced using Organizes References.

- The instances of the available Types shall be components of the ConfigurableObject (through HasComponent References).

Figure 2.7 shows the ConfigurableObjectType ObjectType.
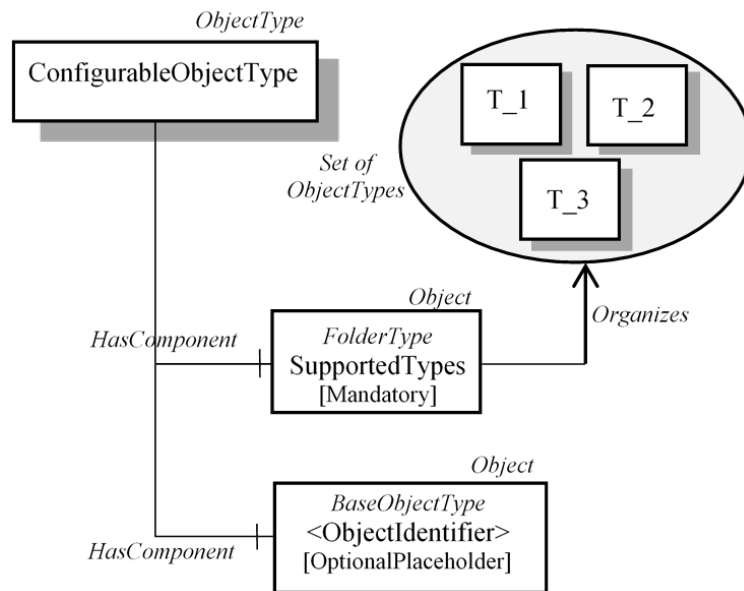


**Figure 2.7:** *OPC UA ConfigurableObjectType ObjectType*

# THREE

# JAVASCRIPT OBJECT NOTATION (JSON) DATA INTERCHANGE FORMAT

In the last few years, JSON (JavaScript Object Notation) [31, 32] has achieved remarkable popularity as the main format for the representation and the exchange of information over the modern web.

## 3.1   JSON base types

JSON is a data format based on the data types of the JavaScript programming language. JSON can represent four primitive types (string, number, boolean and null) and two structured types (object and array). In particular, the following base types are defined [31, 32]:

- *String*: a sequence of Unicode character included between double quotes.

- *Number*: numerical values. A number is represented in base 10

using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. A fraction part is a decimal point followed by one or more digits. An exponent part begins with the letter E in upper or lower case, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

- *Literal names*: can assume only the values *true*, *false* and *null*. This JSON base type is used to represent both boolean (using *true* and *false*) and null (*null*) primitive types.

- *Object*: a sequence of key/value pairs between curly brackets where key and value are separated by colon and pairs are separated by commas; a key must be a string whilst a value must be of a JSON base type.

- *Array*: an ordered collection of values between square brackets where values are separated by commas. As for JSON object, a value must be of a JSON base type.

### 3.1.1   JSON Example

As an example, let us consider the JSON document shown by Listing 3.1. It contains a JSON object representing the temperature values measured during to the last 24 hours in Rome (Italy), assuming a sampling interval of 6 hours (i.e. only four temperature values are produced every day).

The JSON object is made up by four key/value pairs. The keys of the pairs are: "country", "city", "isReliable" and

"last24hTemperatures"; the relevant values are of JSON string base type for "country" and "city", JSON literal names base type for "is-Reliable" and JSON array base type for "last24hTemperatures". The boolean value in "isReliable" indicates if the temperature measurements contained in the document are reliable or not. The JSON array base type relevant to the "last24htemperatures" key contains the temperatures of the last 24 hours, measured every 6 hours. For this reason, the array contains four JSON objects, each of which is made up by two key/value pairs where the keys are "timestamp" and "temperature" and the JSON base types are string and number, respectively. They allow to represent the timestamp and the temperature value for each measurement done. This example illustrates the simplicity and readability of a JSON document, which partially explains its fast adoption.

```
{
  "country": "Italy",
  "city": "Rome",
  "isReliable": true,
  "last24hTemperatures": [
    {
      "timestamp": "14/05/2018 00:30:00",
      "temperature": 21
    },
    {
      "timestamp": "14/05/2018 06:30:00",
      "temperature": 22
    },
    {
      "timestamp": "14/05/2018 12:30:00",
```

```
      "temperature": 30
    },
    {
      "timestamp": "14/05/2018 18:30:00",
      "temperature": 26
    }
  ]
}
```

**Listing 3.1:** *JSON Example*

## 3.2   JSON Schema

With the popularity of JSON it was soon noted that in many scenarios one can benefit from a declarative way of specifying a schema for JSON documents. For instance, in the public API scenario one could use a schema to avoid receiving malformed API calls that may affect the inner engine of the application. A declarative schema specification would also give developers a standardised language to specify what types of JSON documents are accepted as inputs and outputs by their API.

*JSON Schema* [33, 34, 35] is a simple schema language that allows users to constrain the structure of JSON documents and provides a framework for verifying the integrity of the requests and their compliance to the API.

JSON schema is represented as JSON objects. A JSON object may be empty (i.e., "{}") or it may contain a number of key/value pairs having special meanings. An empty JSON object validates every JSON document, whilst the second format validates a JSON document

according to the rules associated to each key specified in the JSON Schema. Among these keys there are:

- ”$schema”: it is used to specify if the JSON document is a JSON Schema. It also specifies which version of the JSON Schema specification is used.

- ”properties”: the list of key/value pairs that could be present in the JSON document. For each pair, the related JSON Schema is provided.

- ”type”: the type of the JSON Schema or the value type for each key/value pair listed in ”properties”. When it specifies the type of the JSON Schema, it may assumes several values, among which the “object” value; the relevant meaning is that the JSON Schema is a collection of key/value pairs. When ”type” is used to define the value type for each key/value pair, it may indicate if a value is an *object*, an *array* or a JSON base type. It is worth noting that specification allows to use *integer* to restrict numeric values only to integer numbers (number type identifies both integer and real numbers).

- ”enum”: an array containing the allowed enumerated values.

- ”required”: an array that specifies which properties must be present in a JSON document compliant with the schema. The properties here defined must be a subset of the content of ”properties”.

- ”additionalProperties”: a boolean that indicates if different properties from those specified in ”properties” are allowed.

As an example, let us consider the JSON Schema shown by Listing 3.2.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "country": {"type": "string"},
    "city": {"type": "string"},
    "isReliable": {"type": "boolean"},
    "last24hTemperatures": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "timestamp": {"type": "string"},
          "temperature": {"type": "number"}
        },
        "required": ["timestamp", "temperature"],
        "additionalProperties": false
      }
    }
  },
  "required": ["country", "city", "isReliable", "
      last24hTemperatures"],
  "additionalProperties": false
}
```

***Listing 3.2:*** *JSON Schema Example validating the JSON Example shown in Listing 3.1*

It validates any JSON document made up by key/value pairs where keys are "country", "city", "isReliable" and "last24hTemperatures"

and values are string for "country" and "city", boolean for "isReliable" and array for "last24hTemperatures". In turn, the JSON Schema specifies that each element of the array "last24hTemperatures" is an object made up by two key/value pairs where keys are "timestamp" and "temperature" with string and number values, respectively.

All the four properties specified by the JSON Schema (i.e. "country", "city", "isReliable" and "last24hTemperatures") are mandatory due to their presence in the "required" property. Furhtermore, other key/value pairs are not allowed due to the property "additionalProperties" set to false. For the same reason each object inside the array "last24hTemperature" must be made up by the two properties "timestamp" and "temperature" and no additional properties are allowed.

The JSON document seen before in Listing 3.1 is correctly validated by the JSON Schema of Listing 3.2.

CHAPTER

# FOUR

# OPEN CONNECTIVITY FOUNDATION (OCF)

OCF specifications [36] are based on the REpresentational State Transfer (REST) architecture style [11]. The OCF specifications enable interoperability between heterogeneous devices acting as OCF Clients and devices acting as OCF Servers. The notion of client and server is realised through roles; an OCF Server exposes hosted resources, whilst an OCF Client accesses resources on a server through RESTful operations. Data exchange between OCF Client and OCF Server occurs in JSON format [31].

## 4.1  OCF Resource Model

OCF defines the *OCF Resource Model* to enable the interoperability and to provide consistency between devices in OCF ecosystem [36]. The OCF Resource Model is based on the concepts of *Device* and *Resource*. According to the OCF Core Specification [37], a Device

models a logical entity (e.g., corresponding to a real device) whilst a Resource is the representation of a component of a Device (e.g., a sensor in a smartphone).

An OCF Resource is an instance of one or more OCF Resource Types. Each Resource Type defines a set of properties exposed by the Resource. A Resource is addressed using URI and contains properties. Properties are represented as key/value pairs of a JSON object and are defined using OCF data types derived from JSON base types. According to [37], OCF adopts the same JSON base types described in Section 3.1, with the exception of "true" and "false" value that in JSON are defined by the literal name type but in OCF are mapped in OCF boolean data type (values "true" and "false" are left unaltered). The properties of a Resource represent the state of the Resource itself. In addition, a Resource declare a set of OCF *Interfaces*. Each Interface specifies how is possible to interact with the Resource itself.

OCF specification defines several common properties which must be present in a Resource. They are specified by the "oic.core" Resource Type, and consist of an unique identifier for the Resource in the context of a Device (id), the name of the Resource (n), the Resource Types (rt) and the Interfaces (if) supported by the Resource. The properties of Resource Type "oic.core" must be present in every JSON object representing an OCF Resource and are summarised in Table 4.1.

***Table 4.1:*** *OCF Common Properties defined by "oic.core" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|:---:|:---:|:---:|:---:|
| id | string or uuid | No | Identifier for the Resource in the context of a Device |
| n | string | No | Name of the Resource |
| rt | array of strings | Yes | Defines the Resource Types of the Resource |
| if | array of strings | Yes | Interfaces supported by the Resource |

OCF specifies that a Resource can be related to another Resource through an OCF *Link* [37]. A Link consists of a set of parameters. The "rel" parameter specifies the kind of relationship of a Link; if it is not provided, a value of "rel" = "hosts" shall be assumed. The "href" parameter specifies the target URI of the Resource pointed by the Link. The context URI of a Link is implicitly the URI of the Resource containing the Link itself unless the Link specifies the "anchor" parameter. The "anchor" parameter is used to change the context URI of the Link, in the sense that the relationship with the target URI is based on the anchor URI. As an example, consider a Resource representing a floor containing Links pointing to the Resources representing the rooms in the floor. If each room contain lights, these may be defined in the Resource representing the floor as Links having the "anchor" parameter set to the URI of the Resources representing the rooms containing the lights. Another important parameter is "p" (Policy)

which defines rules for correctly accessing a Resource referenced by the target URI. One of these rules, named *Observable*, may be used to notify the OCF Client on state change of the target Resource.

A Link is always owned by the source Resource, and a Resource with properties and Links is named Collection. Properties of an OCF Collection are defined by the Resource Type "oic.wk.col". Among these properties, "links" is used to gather every Link having as source the Collection itself.

According to OCF specifications, a Device belongs to a *Device Type*. A Device Type is identified by a string; also a Device Name is associated to a Device Type for informative purpose. A Device Type mandates the list of minimum OCF Resources that a Device of this type must expose; a Device may include other Resources, but the implementation of those specified by its Device Type is mandatory.

In order to enable the functional interaction between OCF Client and OCF Server, OCF mandates a list of core Resources that must be supported and exposed by a Device. Specifically, OCF defines three well-known Resources in an OCF Device. These Resources are addressed in the context of the OCF Device using the predefined URIs "/oic/p", "/oic/d" and "/oic/res" and belongs to the OCF Resource Types named "oic.wk.p", "oic.wk.d" and "oic.wk.res", respectively.

The Resource addressed by "/oic/p" URI represents the physical platform hosting the physical device. It is used to expose information about platform like vendor name or software version.

The Resource addressed by "/oic/d" URI represents the device and its properties. Properties of this Resource are defined by the Resource Type "oic.wk.d". These properties provide information about the devices as: a localized description of the device in one or more language

(ld), the software version (sv), the manufacturer name (dmn) and the model number (dmno). It is worth noting that the "rt" property of this Resource includes also the Device Type of the device represented, alongside the Resource Type "oic.wk.d".

The Resource addressed by "/oic/res" URI is the entry point for all the Resources exposed by the OCF Device. It contains OCF Links to each Resource owned by the Device.

In the following, an example of an OCF Device Type will be given. Table 4.2 points out its description according to the formalism used in OCF specifications, giving a Device Name and the string which identifies the Device Type (i.e. "x.my.device").

*Table 4.2:* OCF Device Type Example

| Device Name | Device Type | Required Resource Name | Required Resource Type |
| --- | --- | --- | --- |
| MyDevice | "x.my.device" | MyResource | "x.my.resource" |

According to Table 4.2, each Device of this type must implement a Resource of "x.my.resource". Figure 4.1 shows an instance of this Device Type.

For each Resource, only URI and "rt" property are shown. The Resource containing the Device information (addressed by "/oic/d" URI) contains two elements in the "rt" property; one of these is the "x.my.device" string representing the Device Type in Table 4.2.

An OCF Device can represent a device made up by subdevices. In this case, an OCF Device can expose Resources representing the subdevices. A Resource of this kind belongs to a Device Type (i.e. its "rt" property must contain a Device Type) and shall expose the

OCF Device of "x.my.device" Device Type
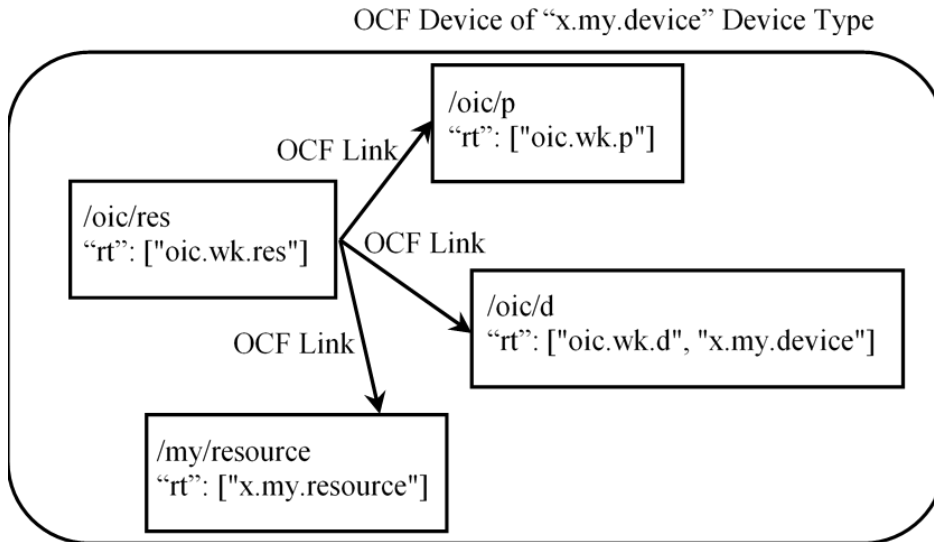


***Figure 4.1:*** *OCF Device instance of "x.my.device" Device Type*

properties defined by "oic.wk.d" Resource Type. Furthermore, if the Resource representing a subdevice is also a Collection (i.e. it has the "oic.wk.col" Resource Type in its "rt" property), it shall link mandatory Resources specified by the Device Type.

Figure 4.2 shows an example of a Device made up by a subdevice. The OCF Device represented in this figure is the same shown by Figure 4.1, with the addition of a Resource representing a subdevice and addressed by the URI "/my/subdevice". As shown, the "rt" property of this Resource contains three strings: "oic.wk.d", "x.my.device" and "oic.wk.col". The first element specifies that this Resource represents a Device whilst the second the relevant Device Type. The last element specifies that this Resource is also a Collection: for this reason it has to link the mandatory OCF Resources defined by the "x.my.device" De-

vice Type (i.e. a Resource belonging to the "x.my.resource" Resource Type as defined in Table 4.2).

OCF Device of "x.my.device" Device Type
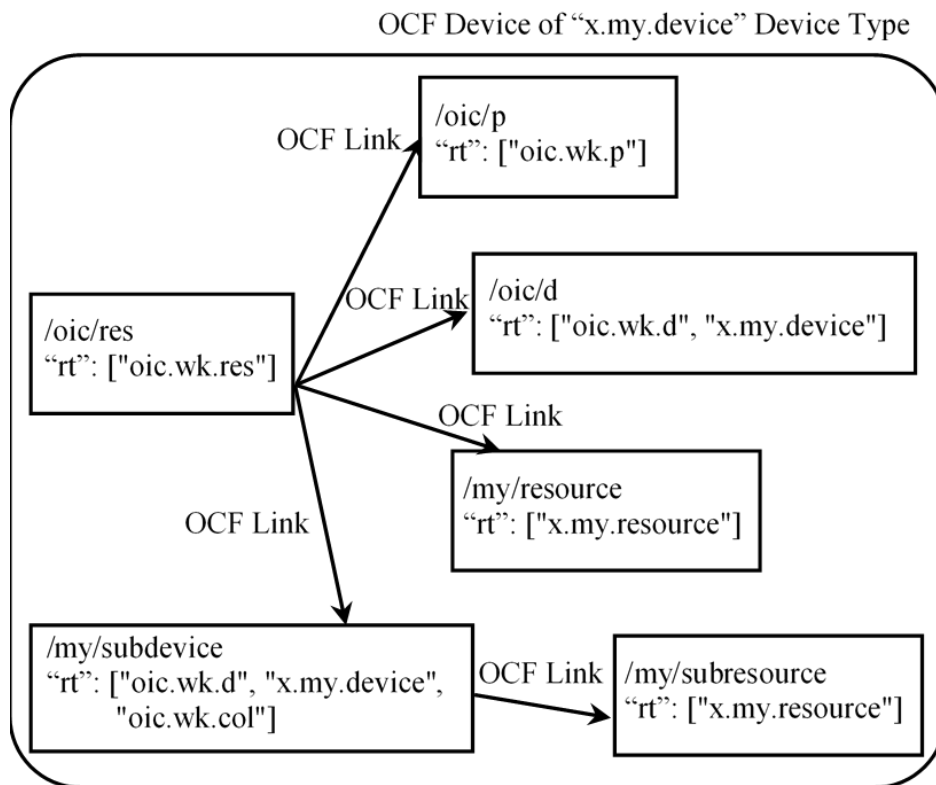


**Figure 4.2:** *OCF Device made up by a subdevice*

# FIVE

# INTEGRATION BETWEEN OPC UA AND THE WEB

As pointed out in the Introduction (Chapter 1), this chapter describes a proposal of integration between OPC UA and the Web, enhancing the interoperability between OPC UA and generic users which do not have any knowledge of the standard.

The proposal consists in the definition of a novel data model based on common web data-formats and mapping the OPC UA Information Model. This data model has been defined and used in the implementation of a web platform, called *OPC UA Web Platform*, able to offer to a generic client a lightweighted interface to OPC UA Servers; lightweight interface is considered both in terms of messages exchanged between client and server and in terms of basic knowledge to be held by a client. The web platform proposed is based on a REST architecture, due to the relevant advantages pointed out in [13].

Several papers and articles describe definition and realization of the OPC UA Web Platform and the relevant data model, from early stages [38, 39, 40] to the actual version described in this chapter [41]. The OPC UA Web Platform has been implemented and the relevant code is available on GitHub [42].

## 5.1   OPC UA Web Platform Architecture

OPC UA Web Platform is based on the adoption of a REST architecture and is shown by Figure 5.1.
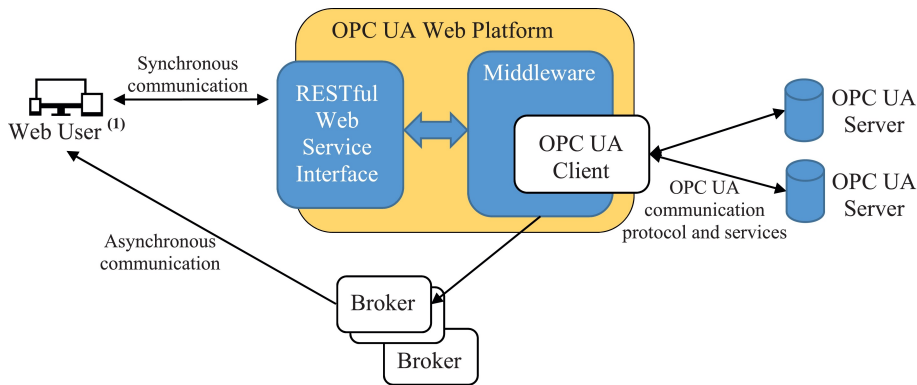


Note [1]: Web User is a generic application running on a generic (smart) device which consumes the services offered by the OPC UA Web Platform. It is constrained neither to be OPC UA compliant nor to implement OPC UA communication stack.

**Figure 5.1:** *OPC UA Web Platform Architecture*

As it can be seen, OPC UA Web Platform offers to a Web User the access to one or more OPC UA Servers. The generic term Web User will identify a generic application which consumes the services offered by the OPC UA Web Platform; these services allow the Web User to access information maintained by OPC UA Servers, without

knowledge of the OPC UA standard and without a real awareness of the presence of OPC UA Servers behind the OPC UA Web Platform, which are seen just as generic servers publishing information. Web User is neither required to be an OPC UA Client nor to implement the OPC UA communication stack (i.e., OPC UA protocol and services). Web User is only constrained to have knowledge of basic concepts detailed in the following subsection.

The OPC UA Web Platform consists of two main modules:

- *RESTful Web Service Interface.* It accepts requests submitted by an authenticated Web User. Communication between Web User and the OPC UA Web Platform is synchronous as for each Web User's request through the RESTful Web Service Interface, the OPC UA Web Platform will send a relevant response; it is realised by communication protocols described in the remainder of this section. Due to the adoption of REST, communication between Web User and OPC UA Web Platform is stateless. This means that each Web User's request is independent from any stored context on the OPC UA Web Platform, and each Web User's request must contain all the information necessary to the OPC UA Web Platform to accomplish the requested service and to generate the relevant response.

- *Middleware.* It performs all the operations needed to fulfil each request coming from a Web User; these requests may require data exchange between the Middleware and the available OPC UA Servers. For this reason, the Middleware includes an OPC UA Client used for the access to the OPC UA Servers. Communication between OPC UA Client and OPC UA Servers occurs

51

according the standard OPC UA communication protocol and
services (i.e. using the OPC UA communication stack), as shown
by Figure 5.1.


The RESTful Web Service Interface provides several service that
will be clearly described in the remainder of this chapter. Among
these services it has been foreseen a particular one, called *Monitor*. It
allows Web User to request the activation of a particular asynchronous
communication between the platform and the Web User based on
Publish/Subscribe Pattern [23]. For this reason, Figure 5.1 shows
the presence of a set of *Brokers*, needed to handle each asynchronous
communication requested by a Web User.

Each Web User needing to use the OPC UA Web Platform must
be previously registered. During registration, user credentials in terms
of username and password must be stored in the OPC UA Web Plat-
form. Credentials are issued by a Web User only one time at his first
access to the OPC UA Web Platform. The platform validates these
credentials and generates a signed token which is returned to the Web
User. He will use the signed token received, in each next web service
synchronous request issued to the OPC UA Web Platform.

In order to guarantee a secure communication between Web User
and OPC UA Web Platform, encryption of data information flow (in-
cluding the exchange of the signed token) has been realised by HTTPS
[43]. Secure communication between OPC UA Web Platform and each
OPC UA Server is realised by the OPC UA Client integrated in the
Middleware, through the OPC UA secure mechanisms [29] (explained
in Section 2.3).

The next two subsections will deal with two important items about

the basic knowledge and communication technologies that a Web User must hold to access the platform. These sections will allow the reader to realise how minimal are the requirements requested to access the platform, allowing a real enhancement of OPC UA interoperability with a very large set of applications.

### 5.1.1 Web User's basic knowledge needed to access the OPC UA Web Platform

It has been assumed that the Web User must be aware about the capability of the OPC UA Web Platform to make available one or more *data sets* to be accessed for both reading and writing operations. Each data set has an identifier called *dataset-id*, assigned by the OPC UA Web Platform as an integer positive number. Each data set is mapped by the OPC UA Web Platform to an OPC UA Server, although this mapping is hidden to the Web User who is not really aware about the presence of OPC UA Servers.

Each data set is seen by a Web User as a set of elements, called *Nodes* (not OPC UA Nodes but simply Nodes), each of which may refer to a variable, an object, a method and a folder (they will be called *Variable Node*, *Object Node*, *Method Node* and *Folder Node*, respectively). It is assumed that the common concept of variable, object, folder and method must be held by the Web User. About the object, Web User must be aware that an Object Node is made up by other Object, Variable and Method Nodes. Furthermore, an Object Node may publish an event and may be have an event source connected to it. The concept of property must be also held by the Web User; he must be aware that a property may be associated to

an Object or a Variable Node, giving additional information about it. OPC UA Web Platform is in charge to map OPC UA Variables, OPC UA Objects, OPC UA Methods and OPC UA FolderType Objects to Nodes; this map will be hidden to the Web User. Each Node has a *node-id* assigned by the platform and visible to the Web User.

On the basis of what said until now, concatenation of dataset-id and node-id allows the Web User to univocally identify each Node exposed by the OPC UA Web Platform.

Another Web User's basic knowledge is that Nodes are linked each other; the concept of link to be held is the same of an edge in a graph. Web User must be aware that for each Node one or more edges may exist, linking the source Node to a target Node. The edge is always oriented from the source Node to the target Node. Furthermore, each edge features an attribute which gives information about the relationship between source and target Nodes. The following values have been foreseen for the relationship attribute:

- *HasComponent.* If the source is an Object Node, the target may be an Object, a Variable or a Method Node; in this case, the meaning is that the source Node is made up by the target Nodes. If the source is a Variable Node, the target must be a Variable Node; the meaning is that the Variable Node is made up by a collection of the target Variable Nodes. This attribute is mapped by the OPC UA Web Platform to the OPC UA HasComponent and HasOrderedComponent Hierarchical ReferenceTypes.

- *HasProperty.* The source may be an Object or a Variable Node, whilst the target must be a Variable Node. It means that the Object or Variable Node features a property given by the value

of the target Variable Node. It is mapped by the OPC UA Web Platform to the OPC UA HasProperty Hierarchical Reference-Types.

- *Organizes.* The source is a Folder Node, whilst the target may be Object and Variable Node. It means that the folder organises (i.e. it is a directory containing) the target Nodes. It is mapped by the OPC UA Web Platform to the OPC UA Organizes Hierarchical ReferenceTypes.

- *HasChild.* It may connect whatever kinds of Nodes. Its meaning is that a non-looping hierarchy between these Nodes exist, according to the orientations of the edges. It is mapped by the OPC UA Web Platform to a vendor-specific subtype of the OPC UA HasChild Hierarchical ReferenceType.

- *Aggregates.* Target and source Nodes may be generic. It means that the target Node belongs to the source Node. It is mapped by the platform to a vendor-specific subtype of the OPC UA Aggregates Hierarchical ReferenceType.

- *HasEventSource.* An edge with this attribute will connect a source Node with a target Node to relate event sources in a hierarchical, non-looping organisation. It is mapped by the platform to the OPC UA HasEventSource Hierarchical Reference.

- *HasNotifier.* It is used to relate Object Nodes which are event notifiers. It is mapped by the OPC UA Web Platform to the OPC UA HasNotifier Hierarchical Reference.

On the basis of what said, the edges considered for the Nodes are mapped to the entire set of OPC UA Hierarchical References.  This is important to be pointed out because this information will be used in the remainder of the chapter.  It is also important to be pointed out that this mapping is internal to the OPC UA Web Platform and hidden to the Web User.

For each Node inside a data set, the Web User knows that he may access to its information, provided by the OPC UA Web Platform as properties of a JSON document.  The following basic properties have been defined for each Node: a *node-id*, a *Name* and a *Type*.  As said before, each Node in a data set has a node-id assigned by the platform and visible to the Web User; it has been assumed that OPC UA Web Platform produces the node-id as a string containing the concatenation between the NamespaceIndex and the Identifier of the OPC UA NodeId.  For example node-id="0-85" refers to the OPC UA Node belonging to NamespaceIndex 0 and Identifier 85 (i.e., OPC UA Node with ns=0; i=85).  Name is the text which can be displayed for a Node and is realised by the OPC UA Web Platform as a string containing the OPC UA DisplayName of the relevant OPC UA Node.  Type specifies the type of Node from the Web User's point of view and may assumes one of the following values: "folder", "object", "method", "variable".

Only in the case of Variable Node, Web User may access to the properties *Value*, *ValueTypeSchema*, *Status*, *MinimumSamplingInterval* and *Deadband*.  For each Variable Node, Web User knows that a Value is associated.  In order to correctly understand the value, Web User must be aware about the relevant type. For this reason, description of the type is given to the Web User through the ValueType-Schema, made up by a JSON Schema and describing how the Value

property is made. Status belongs to an enumerated type, which can assume three different values: Good, Uncertain, Bad. Their meanings are the same of StatusCode as defined in the OPC UA, and described in Chapter 2. This knowledge must be held by the Web User.

Web User must be aware that for each Variable Node, an asynchronous transmission mechanism may be realised. The Value of a Variable Node may be automatically sent to the Web User by the OPC UA Web Platform for each change in the value and/or the status of the Node. In order to provide to the Web User the information about the minimum sampling interval that he can request for the asynchronous transmission relevant to a specific Node Variable, the *MinimumSamplingInterval* property has been defined. Furthermore, the Value of a Variable Node may be sent, if requested by the Web User, if the change of the Node Variable Value occurs according to a percentage or absolute deadband, according to conditions (2.1) or (2.2) given in Section 2.2.4. For this reason, for each Variable Node the property *Deadband* has been defined. It specifies if the Web User is allowed to realise the asynchronous transmission based on the deadband on a specific Variable Node and, in this case, the deadband type allowed to the Web User; this property may assume one of the following value: "percentage", "absolute", "both", "none".

In order to better understand what introduced until now, the following example will be given. Figure 5.2 shows a simple OPC UA AddressSpace, pointing out the presence of several types of References and NodeClasses. On the top of the AddressSpace there is an OPC UA FolderType Object, which organises other OPC UA Objects. Some of the OPC UA Objects have several components, among which Methods and DataVariables. Among the OPC UA DataVariables, it is possi-

ble to see one featuring a Value attribute belonging to the MyType
DataType considered in Section 2.1.5 and shown by Figure 2.3. This
means that the OPC UA AddressSpace contains also the definition of
this DataType, i.e. all the OPC UA Nodes shown in Figure 2.3 (not
shown for space reason). Figure 5.3 shows the OPC UA AddressSpace
given by Figure 5.2 from the Web User's perspective.



*Figure 5.2:* *Example of OPC UA AddressSpace*

Note [1]: This Variable Node is relevant to the OPC UA Node with Value of MyType DataType

***Figure 5.3:*** *Web User's view of the OPC UA AddressSpace shown by Figure 5.2*

As it can be seen in Figure 5.3, the WebUser's view is limited to a set of Nodes of different types, connected by oriented edge with the relationship property. The Variable Node features a different background colour in order to point out that for them the Web User will receive additional information detailed in the following. As it is possible to see, the Web User's view of the OPC UA AddressSpace, allows him to capture the relevant structure in terms of objects, folders and methods and gives to him detailed information about variables.

Comparing Figures 5.2 and 5.3, it is possible to point out that all the information about OPC UA types are hidden to the user. This

does not means that information about types disappears, but they are used by the OPC UA Web Platform and passed to the Web User in a more friendly form, when needed.

As an example, let us consider the information about Objects and Folders. In the OPC UA AddressSpace shown by Figure 5.2, the BaseObjectType and FolderType ObjectTypes are used to specify if an OPC UA Object is a folder or not. In the Web User's view this information is passed through the Type property of the Node, specifying if the Node is an Object Node or a Folder Node, as shown by Figure 5.3.

Another example is the description of the MyType DataType which is included (but not shown for space limitation) in the OPC UA AddressSpace of Figure 5.2. This description is totally hidden to the Web User, as shown by Figure 5.3, but it is given to him through a JSON Schema document contained in the ValueTypeSchema property of the relevant Variable Node (the Node with the Note (1) shown in Figure 5.3); the remainder of this chapter will detail how JSON Schema is built and used to fill the ValueTypeSchema property.

The examples just presented allow to enforce the concept that although the Web User's view is simplified, information maintained by the OPC UA AddressSpace is not lost for the Web User, but it is given to the Web User in a more friendly way.

## 5.1.2   Web User's communication technologies needed to access the OPC UA Web Platform

Communication between Web User and the OPC UA Web Platform may be synchronous (based on RESTful web services [11]) and asynchronous (based on Publish/Subscribe Pattern [23]), as shown by Figure 5.1.

It has been assumed to realise synchronous communication through the HTTP protocol using a connection encrypted by the Transport Layer Security (TSL), i.e. HTTPS [43]. It allows to realise encryption of data information flow between Web User and OPC UA Web Platform, and vice versa.

About asynchronous communication, several technologies are currently based on the Publish/Subscribe Pattern; the technologies used for the implementation of the proposed platform are both Microsoft SignalR [44] and MQTT [45]. Web User is allowed to choose one of them to realise the asynchronous communication according to his preferred/available technology. Considering the SignalR-based communication, the Hub model has been adopted; it is basically based on the use of WebSocket, but it allows the use of HTTP if WebSocket technology is not supported by the Web User. Among available implementations of MQTT, the authors were interested in Mosca [46]. Mosca is a MQTT broker whose implementation is based on Node.js [47] and may use WebSocket or HTTP.

## 5.2 Improvements of the proposal over the related work present in literature

As pointed out in the Introduction of this thesis (Chapter 1), other solutions proposing a REST architecture for OPC UA exist in the current literature. The ones that seem closer to the proposal are described in [13, 14, 15], called in the following RESTful OPC UA [13] and Prosys [14, 15]. The aim of this section is to compare these solutions with the OPC UA Web Platform in order to highlight its improvements over these existing works. Comparison will be done from two points of view: web technologies used for the communication and OPC UA basic knowledge that the user has to hold. Tables 5.1 and 5.2 summarise these differences which will be better explained in the two following subsections.

Furthermore, in Chapter 2 it has been said that a novel OPC UA specification based on a Publish/Subscribe pattern has been defined. As the OPC UA Web Platform adopts the Publish/Subscribe pattern, a last subsection presents a comparison between the novel OPC UA PubSub specification and the OPC UA Web Platform, in order to give a better understanding about the potential relationship between the proposed approach and the OPC UA specification.

**Table 5.1:** *Web technologies adopted in the proposal versus existing solutions*

| RESTful OPC UA | | Prosys | | OPC UA Web Platform | |
|---|---|---|---|---|---|
| Sync | Async | Sync | Async | Sync | Async |
| Modified OPC UA Comm. Stack | N/A | HTTP | Server Sent Event over HTTP | HTTP | MQTT or SignalR over Web-Socket or HTTP |
| UDP | N/A | TCP/IP | TCP/IP | TCP/IP | TCP/IP |

**Table 5.2:** *Basic Knowledge held by Web User versus existing solutions*

| RESTful OPC UA | Prosys | OPC UA Web Platform |
|---|---|---|
| User is an OPC UA Client, so it must hold full knowledge of OPC UA Information Model and Services and must hold the modified OPC UA communication stack | Full Knowledge of OPC UA Information Model and Services; the client is constrained to implement a specific subset of OPC UA services and protocol | Basic Knowledge described in Section 5.1.1 and no need to implement OPC UA services and protocol (no need of OPC UA communication stack). |

## 5.2.1 OPC UA Web Platform versus RESTful OPC UA solution

RESTful OPC UA [13] provides a RESTful approach in the OPC UA Client/Server communication model. No middleware exists between a user and an OPC UA Server as the user itself is an OPC UA Client which directly exchange data with the OPC UA Server. The RESTful data exchange model is obtained using a modified OPC UA communication stack. The RESTful data exchange has been limited to the OPC UA synchronous services through the use of UDP as transport protocol; asynchronous ones are realised according to the original OPC UA standard. As consequence, from the communication point of view the only similarity with the proposal here presented is the REST architecture chosen in both approaches. In RESTful OPC UA the user must have the full knowledge of OPC UA Information Model and Services as it is an OPC UA Client and has to implement a modified OPC UA communication stack. The proposed solution has the aim to provide a RESTful approach in the communication with an OPC UA Server but it does not force the users to be OPC UA Clients but rather it enlarges them to a generic Web User as described before. This is obtained using a middleware which hides the OPC UA communication. For this reason, it is not requested to the Web User neither to implement OPC UA communication stack nor to have full knowledge about OPC UA Information Model and Services. In conclusion, the OPC UA Web Platform adopts a RESTful approach similarly to the solution proposed in [13] but it is able to improve the RESTful OPC UA eliminating the restriction on users, which are no longer forced to be OPC UA Clients but simple Web Users.

## 5.2.2   OPC UA Web Platform versus Prosys solution

Prosys solution [14] is mainly based on the Master's Thesis [15]. Although the platform presented in [15] and the one here proposed offer interfaces which seem similar, there are a lot of differences between them. In the following these differences will be pointed out, highlighting that the OPC UA Web Platform approach do not overlap the Prosys approach but improves it.

The main assumption of [14] is that a user is a web client compliant to basic OPC UA Client Facets [48]. In other words the web client must be an OPC UA Client implementing a specific subset of services and protocol of the OPC UA specifications. The OPC UA Web Platform proposal removes the restriction on users as they have to be simple Web User (i.e. it is required neither to be an OPC UA Client nor to have knowledge about OPC UA standard). This means a great simplification as a user is not forced to implement the OPC UA communication stack and specific services foreseen by the standard. The advantage of this choice is the enhancement of interoperability between OPC UA Servers and generic clients not compliant with OPC UA standard, increasing the set of devices and applications which can access the information maintained by the former. For this reason the OPC UA Web Platform improves the solution [14, 15] allowing access from the web to OPC UA Server not only for application compliant with OPC UA Facets, but also for generic applications not compliant to OPC UA standard which cannot have access to information owned by OPC UA Servers using solutions like those presented in [14, 15].

In [14] is clearly stated that the web client was designed to support

standard web browsers; in other words, the work presented in [14] aims to realise an OPC UA Client interface running on a web browser. The OPC UA Web Platform presented in the thesis offers an interface to a generic user without any constraints. For instance, Web User may be either a human operator using a web browser or an application running on a smart device. From this point of view, the proposed approach enlarge the set of applications/devices which may use the platform to access OPC UA Servers.

In order to improve interoperability of OPC UA Server into environments not compliant with the OPC UA standard, in the proposal here presented, particular effort has been put to the problem of data encoding of vendor-specific Structured DataTypes. As said in Section 2.1.5, OPC UA allows the definition of vendor-specific Structured DataTypes stored in an OPC UA Server. If an OPC UA Client needs to access a OPC UA Variable whose Value attribute belongs to a vendor-specific Structured DataType, the relevant encoding rules must be retrieved from the server. If a client is not compliant to OPC UA standard, it has no possibility to access such encoding rules. For this reason, the proposal foresees that for each OPC UA Variable of this kind, the generic client receives a JSON document containing its current value alongside a JSON Schema describing the structure of the value. This value is human-readable and is obtained by the proposed platform applying the encoding rules of the relevant DataType. It is worth noting that the JSON Schema sent to the generic client may be used for several purposes, among which to validate the value received and to allow the client to write new well-formed values to be sent to the platform. It is worth noting that the proposed platform gives to the Web User the human-readable value regardless of the OPC

UA DataType, in order to improve interoperability allowing whatever client to understand information received without any knowledge of the OPC UA DataTypes. In the approach presented in [14] the problem of distribution of the values encoded according to vendor-specific OPC UA DataTypes is not treated; as consequence, a web client is not allowed to understand the relevant values received from an OPC UA Server but it has to decode the data received applying the relevant encoding rules. In this case the OPC UA Web Platform improves the Prosys solution resolving the problem of decoding complex data, directly providing comprehensible data and avoiding the decoding of the data by the user.

Both the solutions presented in [14] and in this thesis feature an interface exposing several resources defined according to a REST architecture. Choice of the resources is totally different in the two approaches. In [14], it has been assumed that OPC UA items relevant to the OPC UA Client/Server communication model become REST resources; example of such resources are the OPC UA MonitoredItem, the OPC UA Session and the OPC UA Subscription. There are several consequences relevant to this choice. First of all, the web client must be aware of the meaning of these resources, and this explains why it must be compliant to OPC UA Client Facets [48]. A generic client not compliant with OPC UA cannot understand the meaning of these resources as they are strictly linked to the OPC UA standard. Another important consequence is that the access to a single piece of information maintained by the OPC UA AddressSpace could not be realised through the direct access to the relevant resource exposed by the interface, but through a preparatory sequence of requests to the interface according to the OPC UA Client/Server communication

model. For example, if a client of [14] wants to monitor an OPC UA Node, it is obliged to request the creation of new resources relevant to OPC UA Session, Subscription and MonitoredItem if they do not exist. The OPC UA Web Platform removes this overhead as the Web User is able to obtain exactly the single piece of information from a certain server without the need to perform preparatory accesses to resources different from the one containing the information desired. In fact, all the stateful tasks related to OPC UA Client/Server communication are hanled in a black-box approach, and the information must be obtained at any moment simply accessing a single resource. This improvements make possible the interoperability with applications running on very simple and resource-constrained devices (e.g., smart devices inside an IIoT environment) which do not know the OPC UA communication model. This results in a reduction of the complexity on the client side compared to the solution in [14].

The choice made in [14] to define a resource for each OPC UA item related to the OPC UA Client/Server communication model leads to address each resource by paths strictly bounded to a single client. This due to the OPC UA standard which requires that items like OPC UA Session and Subscription can be used only by the OPC UA Client which created them. For instance, in [14] an OPC UA Node is represented by the resource featuring the path "/api/sessions/$\{sessionId\}$/nodes/$\{nodeId\}$"; it is clear that $\{sessionId\}$ is the identifier of a Session created by the client accessing the node and can be used only by it. This restriction has been removed in the OPC UA Web Platform where each resource is uniquely identified by a path independent from the Web User; this improves the availability of the data that must be obtained by several users using the same path.

Another improvement of the OPC UA Web Platform is about stateless constraints of the REST architecture. In [14] is clearly stated that the current development of the proposed architecture is featured by the storing of sessions created by the web clients, breaking in this way the stateless constraint of the REST architecture. Sessions are handled through Cookies as the web client is assumed to be a web browser. Furthermore, use of Cookies limits the portability of the proposal presented in [14] into an environment different by a web browser. The lack of stateless constraint in the approach presented in [14] is also due to the delivery of the real-time push notifications to clients through the use of Server-Sent Events (SSE). In particular, this approach uses a heartbeats mechanism sent through a separate push endpoint; the absence of heartbeat messages informs the connected web clients that connection to the service has been lost. This allows the client user interface to display the connection status to the user. It is clear that maintaining the connection status is not compliant with the stateless concept of the REST architecture, as said before. The stateless principle has been respected in OPC UA Web Platform as clearly explained; furthermore, there is no constraints on the kind of client, which may be run inside a web browser or may be an application running on whatever (smart) device.

## 5.2.3 OPC UA Web Platform versus OPC UA PubSub Specification

OPC UA is a developing standard; at this time, the novel OPC UA PubSub Specification [22] has been defined by the OPC Foundation. It specifies a Publish/Subscribe model (in addition to the existing

Client/Server model) used to enable an asynchronous stream of information between Publishers and Subscribers through a generic *Message Oriented Middleware (MOM)*, as shown by Figure 5.4.
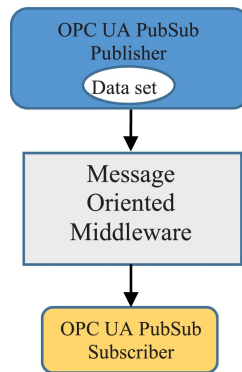


***Figure 5.4:*** *OPC UA PubSub Model*

This model seems close to the one here proposed to implement the asynchronous communication between Web User and the OPC UA Web Platform (as depicted by Figure 5.1). Comparing the two models, it is worth noting that Web User and the OPC UA Web Platform play the roles of subscriber and publisher, respectively. Although the two models seem similar, there are several differences between them explained in the following.

In OPC UA PubSub model, a Publisher may be pre-configured (configuration mainly involves the data set to send) or may be configured using the PubSub Configuration Model as defined by [22]; in this last case the OPC UA PubSub Model should be used in synergy with the OPC UA Client/Server model as specification defines particular OPC UA Methods to change this configuration. In particular, it is requested that, on behalf of one or more subscribers, an OPC

UA Client creates a Session with the OPC UA Server and then invokes such methods by the Call service of the OPC UA Client/Server communication model. In the approach here presented, a generic Web User is able to directly configure the data set to be published (specifying the rate and the Topic where the data will be issued), using the interface of the proposed platform.

OPC UA PubSub model specifies that a Subscriber can be a native OPC UA application or an application that has just knowledge about the rules used to decode and understand the messages exchanged with the Publisher. In particular, a Publisher envelopes data in a *DataSetMessage*, contained into a *NetworkMessage*, in turn contained in the payload of a transport protocol segment (e.g. UDP, AMQP, MQTT). Instead, the platform here presented issues encoded data directly in the payload of the protocol segment. About data encoding, the proposed approach adopts the same OPC UA JSON encoding rules defined by [27]. The Web User does not need any rule to extract data from the payload; furthermore, it is able to understand the data received using suitable JSON Schemas provided by the interface of the OPC UA Web Platform.

OPC UA specifies that the transport protocols OPC UA TCP, HTTPS and WebSocket could be used in the Client/Server communication model. OPC UA PubSub model could be used in two different variants: *broker-based* (in this case the MOM is a Broker and standard messaging protocols like AMQP or MQTT are used in the communication with it) and *broker-less* (in this case the MOM is the network infrastructure and datagram protocols like UDP are used in order to realise the communication between Publisher and Subscriber). In the proposal here presented, the synchronous communication is based on

HTTP requests, whilst the asynchronous communication scenario is based on the use of broker and MQTT and SignalR as messaging protocols (which may both rely on WebSocket or HTTP).

It is worth noting that the solution here presented can coexist with the OPC UA PubSub model; the OPC UA Web Platform may rely on both OPC UA Client/Server and OPC UA PubSub models. Figure 5.5 shows a possible use of the OPC UA PubSub model inside the proposed solution; the asynchronous communication currently implemented by the OPC UA Web Platform Middleware on the basis of the OPC UA Client/Server model may be realised through the novel OPC UA PubSub model.
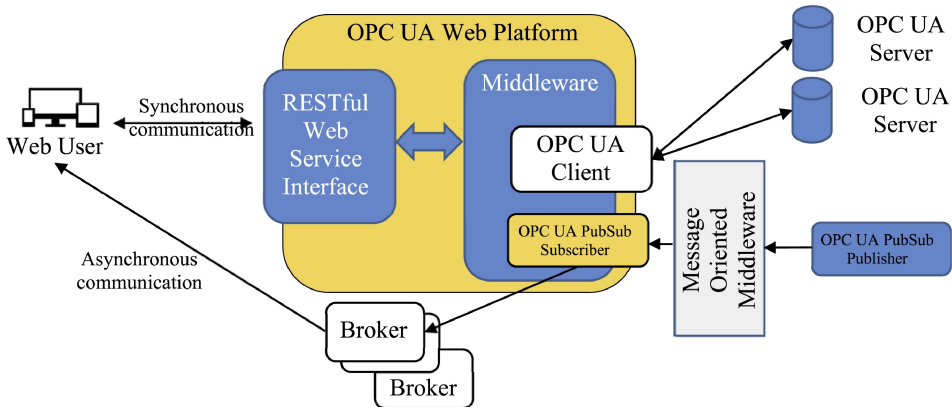


**_Figure 5.5:_** _Possible use of the OPC UA PubSub model inside the proposed platform_

## 5.3   RESTful Interface

The definition of OPC UA Web Platform Interface is based on the REST architectural approach.  The platform exposes resources on which Web User may perform a subset of CRUD (Create, Read, Update and Delete) operations mapped on HTTP methods. According to the classification of REST resources done in [49], it is possible to define the following kind of resources: *Collection*, *Document* and *Controller*.

The list of available data sets was modelled as a Collection resource, which allows the Web User to retrieve information about to the data sets offered by the OPC UA Web Platform.

A Document resource refers to a Node allowing to the Web User to retrieve information about the state of a single Node; definition of the state of a Node will be given in Section 5.3.3.  If the Document resource represents a Variable Node, Web User can also update the Variable Node value.

Controller resource models a procedural concept.  In the proposed platform two procedures have been implemented related to Web User authentication and to an asynchronous transmission mechanism of Variable Node values.

Resources are addressed using URI as described in [49]. For example the URI "/data-sets/1/nodes/2-75" refers to a resource representing a Node with node-id "2-75" and belonging to the data set with dataset-id "1".

It has been assumed that Node with node-id="0-85", corresponding to the OPC UA Object named *Object* with NodeId="ns=0; i=85" described in Section 2.2, will be the entry point of a data set.

Based on what said until now, Table 5.3 summarises the resources

provided by the platform and the HTTP methods used for interaction.

*Table 5.3: Resources and HTTP methods defined*

| Kind of Resource | Resource | GET | POST |
|---|---|---|---|
| Collection | /data-sets | Return the list of available data sets | N/A |
| Document | /data-sets/{dataset-id}/nodes/{node-id} | Return the state of the Node *node-id* in the data set *dataset-id* | Update the value of a Variable Node with *node-id* in the data set *dataset-it* |
| | /data-sets/{dataset-id}/nodes/ | Return the entry point of a specific data set (the state of the Node with *node-id="0-85"* in the data set with *dataset-id*) | N/A |
| Controller | /authenticate | N/A | Return the auth token |
| | /data-sets/{dataset-id}/monitor | N/A | Activate async transmission of the Variable Node values in a specified data set |
| | /data-sets/{dataset-id}/stop-monitor | N/A | Stop async transmission previously activated |

### 5.3.1 Web User Authentication

In order to guarantee a secure communication between Web User and OPC UA Web Platform, an ad-hoc Controller resource with URI ”/authenticate”, has been defined. Web User makes a POST request on this resource, specifying his credentials stored in the platform during registration phase. The OPC UA Platform validates the user credentials and generates a signed token for the Web User, which is returned to him in the POST response. An encryption mechanism based on HTTPS allows a secure transmission of the token. The Web User will use the signed token received, in each next request issued to the OPC UA Web Platform.

### 5.3.2 Information about Data Sets

The Collection resource with URI "/data-sets" allows a Web User to request information about available *data sets*. For each GET request made by Web User on this resource, the OPC UA Web Platform will invoke the FindServers Service [8, 26] to retrieve the list of the available (and registered) OPC UA Servers. The OPC UA Web Platform will associate a *dataset-id* (assumed to be a positive integer) to each OPC UA Server URL. The platform will send back the list of the couples *dataset-id*/OPC UA Server URL, by means of the GET response. In each following request involving a particular data set, the Web User will indicate the relevant dataset-id in the URI path.

### 5.3.3   Information about Nodes

Resource with URI *"/data-sets/{dataset-id}/nodes/{node-id}"* has been defined to represent a Node with node-id {*node-id*} in a specific data set with dataset-id {*dataset-id*}. Making a GET request on this resource, the Web User will receive information about the state of the Node specified. The authors assumed that the state of a Node is composed by two parts: the first one describes the Node itself, whilst the second one describes the edges having the Node as source.

The description of the Node depends on the kind of the Node. In the case of Variable Node, its description is made up by the properties: *node-id, Name, Type, Value, ValueTypeSchema, Status, Deadband, MinimumSamplingInterval.* For the other kinds of Node, only *node-id, Name* and *Type* are given to the Web User.

Description of edges having the Node as source, is made up by a list of elements, one for each target Node; each element describes the target Node (specifying only the properties node-id, Name and Type) and the relationship property of the edge connecting source and target Nodes.

Web User may optionally make a GET request on the URI *"/data-sets/{dataset-id}/nodes"*. In this case, the OPC UA Web Platform will transform this URI into *"/data-sets/{dataset-id}/nodes/0-85"*, allowing the Web User has to retrieve state of the entry point of the data set.

For each GET request on *"/data-sets/{dataset-id}/nodes/{node-id}"*, the OPC UA Web Platform will access to the OPC UA Node relevant to the Node with the node-id {*node-id*}, inside the OPC UA Server relevant to the data set with the dataset-id {*dataset-id*}. In the

following this OPC UA Node will be identified as OPC UA Starting Node. All the OPC UA attributes of the OPC UA Starting Node are retrieved by the OPC UA Web Platform and temporary stored inside it.

As Section 5.1.1 pointed out that the edges refer to the entire set of OPC UA Hierarchical Reference, the OPC UA Web Platform will follow the Hierarchical References starting from the OPC UA Starting Node. Following these references, the OPC UA Web Platform builds a temporary graph through a cut of the OPC UA AddressSpace with depth one starting from the OPC UA Starting Node, as shown by Figure 5.6.
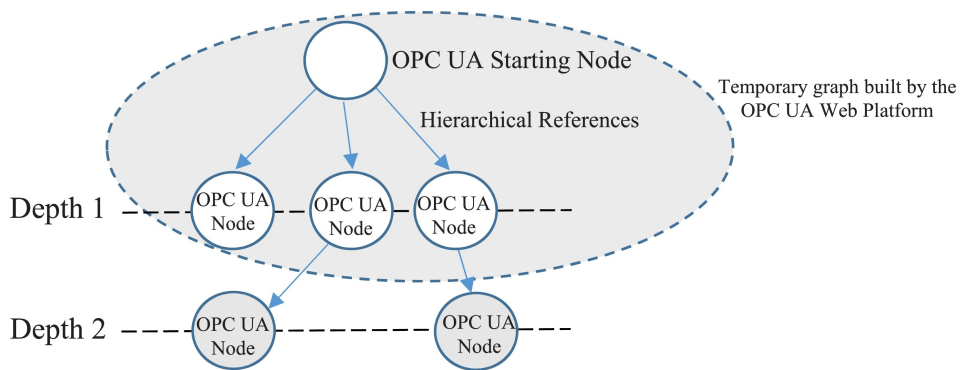


**Figure 5.6:** *Graph realised by cutting OPC UA AddressSpace*

Information relevant to the OPC UA Starting Node, to the other OPC UA Nodes contained in the temporary graph and to the relevant OPC UA Hierarchical References is used to build the Node state as a JSON document compliant to the JSON Schema shown by Listing 5.1.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "node-id": {"type": "string"},
    "Name": {"type": "string"},
    "Type": {"enum": ["folder", "object", "method", "
        variable"]},
    "Value": {},
    "ValueTypeSchema": {"type": "object"},
    "Status": {"enum": ["good", "uncertain", "bad"]},
    "Deadband": {"enum": ["absolute", "percent", "both", "
        none"]},
    "MinimumSamplingInterval": {"type": "number"},
    "Edges": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "node-id": {"type": "string"},
          "Name": {"type": "string"},
          "Type": {"enum": ["folder", "object", "method", "
              variable"]},
          "Relationship": {"enum": ["HasComponent", "
              HasProperty", "Organizes", "HasComponent", "
              HasChild", "Aggregates", "HasEventSource", "
              HasNotifier"]}
        },
        "required": ["node-id", "Name", "Type", "
            Relationship"],
        "additionalProperties": false
      }
```

```
    }
  },
  "required": ["node-id", "Name", "Type", "Edges"],
  "additionalProperties": false
}
```

***Listing 5.1:*** *JSON Schema for the description of the Node state*

As shown by Listing 5.1, the information relevant to node-id, Name, Type and Edges are mandatory. Only in the case of Variable Nodes, Value, ValueTypeSchema, Status, Deadband and Minimum-SamplingInterval properties are also present. Edges is an array of objects, each representing a single edge. For each edge information about target Nodes and the relationship is provided. JSON document describing the Node state is given to the Web User inside the body of the GET response.

### 5.3.3.1 Decoding Procedure of the OPC UA Variable Value

As said in Section 5.1.1, Web User has no visibility about Built-in and Structured DataTypes and related encodings for each OPC UA Variable. This requires that for each Variable Node, the Web User must receive from the OPC UA Web Platform the description of the type relevant to the Value property, in order to correctly validate each value received, and create well-formed value in case it wants to write a new value for the Variable Node. It has been assumed to send this description inside the ValueTypeSchema property shown by Listing 5.1.

Let us assume that the Web User makes a GET request on a Node relevant to an OPC UA Variable whose Value attribute belongs to

a Built-in DataType. In this case, the OPC UA Web Platform is in charge to retrieve information about the OPC UA standard data types used to encode the Built-in DataType; on the basis of the standard data type, the corresponding JSON base type, able to represent it, must be found. In the research carried on, it has been verified that for each OPC UA Built-in DataType, a JSON base type representing it exists. Finally, the OPC UA Web Platform will prepare the JSON Schema shown by Listing 5.4, where the value "XXX" is replaced by the JSON base type corresponding to the standard data type encoding the OPC UA Built-in DataType. This Schema is put into the ValueTypeSchema property of the JSON document describing the Node state.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "XXX"
}
```

**Listing 5.2:** *JSON Schema for ValueTypeSchema property in the case of Built-in DataType*

Now, let us assume that the Web User makes a GET request on a Node relevant to an OPC UA Variable whose value attribute belongs to a Structured DataType. In this case, the OPC UA Web Platform will consider the DataTypeDictionaryType Variable describing the Structured DataType. The ValueTypeSchema property will be filled with a JSON Schema containing several properties, one for each Field element of the StructuredType entry (see Figure 2.3). Listing 5.3 shows an example of the JSON Schema corresponding to a Struc-

tured DataType. In the case of a Field whose TypeName refers to a Built-In DataType, the relevant property is filled with the corresponding JSON base type. In the example of Listing 5.3, the "FieldName1" property corresponds to the Name of a Field whose TypeName refers to a standard data types; the value "XXX" is filled with the corresponding JSON base type. In the case of a Field whose TypeName refers to a StructuredType, a nested JSON document representing it, is inserted. The JSON document shown in Listing 5.3 presents a property "FieldName2" corresponding to the Name of a Field whose Type-Name refers to a StructuredType. As shown, in this case the property is filled with another JSON document modelling the StructuredType in term of Fields here present (e.g., the field "FieldName3").

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "FieldName1": {"type": "XXX"},
       ...
    "FieldName2": {
      "type": "object",
      "properties": {
        "FieldName3": {"type": "XXX"}
         ...
      }
    }
  }
}
```

**Listing 5.3:** *JSON Schema for ValueTypeSchema property in the case of Structured DataType*

### 5.3.3.2   Fulfilling GET request through OPC UA Services

The GET request on the resource */data-sets/{dataset-id}/nodes/{node-id}"* is accomplished by the OPC UA Web Platform using the OPC UA Browse and OPC UA Read Services [8, 26], as shown by Figure 5.7.

When the OPC UA Web Platform receives a GET request relevant to a Node resource, the first action performed is the call of a procedure named in Figure 5.7, "Check existence of OPC UA Client and Session". This procedure may be called also for other OPC UA Web Platform services, as it will be shown in the following subsections. The procedure is represented by Figure 5.8.
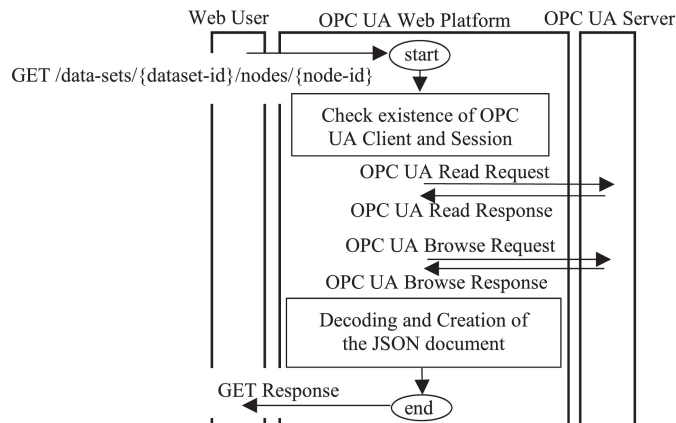


*Figure 5.7:* *Mapping the GET Request with OPC UA Services*

It checks if an OPC UA Client instance exists inside the Middleware. If this does not occur, the instance is made and it will be used in the future for all the Web Users requesting services from the platform. Then, the platform checks for the existence of a Secure Session
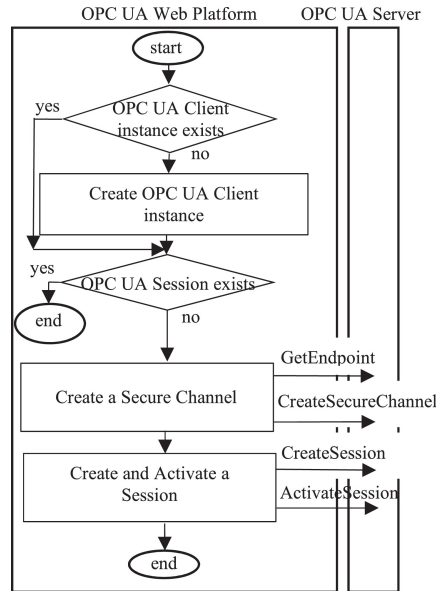
**Figure 5.8:** *Check Existence of OPC UA Client and Session*

between the OPC UA Client and the OPC UA Server relevant to the dataset-id specified by the Web User. If the OPC UA Session does not exist, it is created. According to what explained in Chapter 2 about Secure Session, the OPC UA Web Platform creates a SecureChannel adopting the best security options available [29]. To this aim, it uses the OPC UA GetEndpoint Service to retrieve the list of Endpoints of the OPC UA Server [26], choosing that offering the highest security options. Then the OPC UA CreateSecureChannel Service is invoked [26], as explained in Chapter 2. Finally, an OPC UA Session is created and activated through the OPC UA CreateSession and OPC UA ActivateSession Services [26].

Coming back to the GET request, after the check about existence

of OPC UA Client and OPC UA Session has been completed, the OPC UA Web Platform will invoke the OPC UA Services shown in Figure 5.7, i.e. OPC UA Read and OPC UA Browse, in order to build the graph shown by Figure 5.6.

OPC UA Read request allows the platform to retrieve the full set of information about the OPC UA Starting Node. OPC UA Browse request allows to retrieve the description of the References contained in the OPC UA Starting Node. For each Hierarchical Reference, the Browse Service allows also to retrieve information of the target OPC UA Node pointed by the OPC UA Starting Node using the Hierarchical Reference itself.

Figure 5.7 shows the last procedure named "Decoding and Creation of the JSON document". It aims to create the JSON document giving the Node state according to the JSON Schema shown by Listing 5.1. The main part of the information needed to create this document are given by the results achieved through the previous calls of OPC UA Read and Browse Services [26] as explained before. In the case of a Variable Node, the procedure described in Section 5.3.3.1 must be applied to fill the ValueTypeSchema property shown by Listing 5.1.

When the JSON document has been built according to the JSON Schema of Listing 5.1, it is inserted in the GET response sent to the Web User.

### 5.3.3.3 Case Study

In order to better understand the procedure behind the GET request on the resource *"/data-sets/{dataset-id}/nodes/{node-id}"*, in the following a practical example will be considered. It refers to the simple Information Model shown by Figure 5.9, made up by an OPC UA FolderType Object named Controllers and two OPC UA DataVariables named Controller1 and Controller2 organised by the folder.
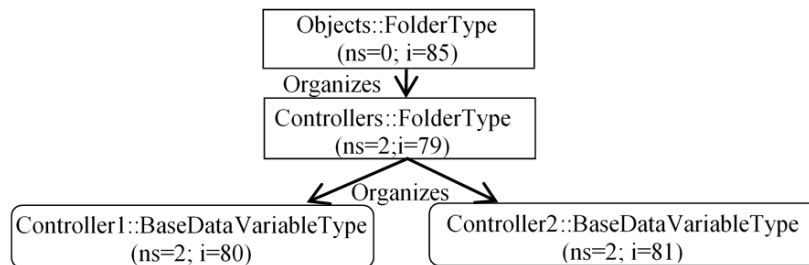


***Figure 5.9:*** *OPC UA AddressSpace used in the Case Study*

It has been assumed that the information model is identified by the NamespaceIndex number 2 (i.e., ns=2); the figure shows the Identifiers of three nodes inside this Information Model (i.e., i=79, 80 and 81). The same figure points out that this information model is reachable from the standard OPC UA Node *Objects* (ns=0; i=85).

Table 5.4 points out some attributes of Controller1 OPC UA DataVariable.

**Table 5.4:** *Attributes of Controller1 DataVariable*

| Attribute | Value |
|---|---|
| NodeId | ns=2;i=80 |
| DisplayName | Controller1 |
| Value | Byte[] |
| DataType | TController (NodeId) |
| MinimumSamplingInterval | 500 |

The Value attribute of Controller1 contains the most recent value of the Variable: in the example, its content has been assumed to be represented in Binary Encoding. Its type is defined by the DataType attribute; as shown by Table 5.4, this attribute specifies the NodeId of the OPC UA DataType Node defining the type of the Value attribute. In this example, the OPC UA DataType Node is named TController, which has been assumed to be a Structured DataType. Another attribute presented in Table 5.4 is the MinimumSamplingInterval; the table shows the relevant value assumed in this example.

Figure 5.10 shows the TController DataType Node. Its description is very similar to that shown by Figure 2.3 for the MyType DataType in Section 2.1.5. As it can be seen, the MyCustomDictionary Variable specifies that the Structure DataType is made up by two Fields named "var1" and "var2", of Int32 and Type2 DataTypes respectively. Int32 is a Built-in DataType; Type2 is another Structured DataType made up by two fields named "var3" and "var4", of Int32 and String DataTypes respectively, as shown by Figure 5.10.

Suppose that the Web User makes the GET request on the resource
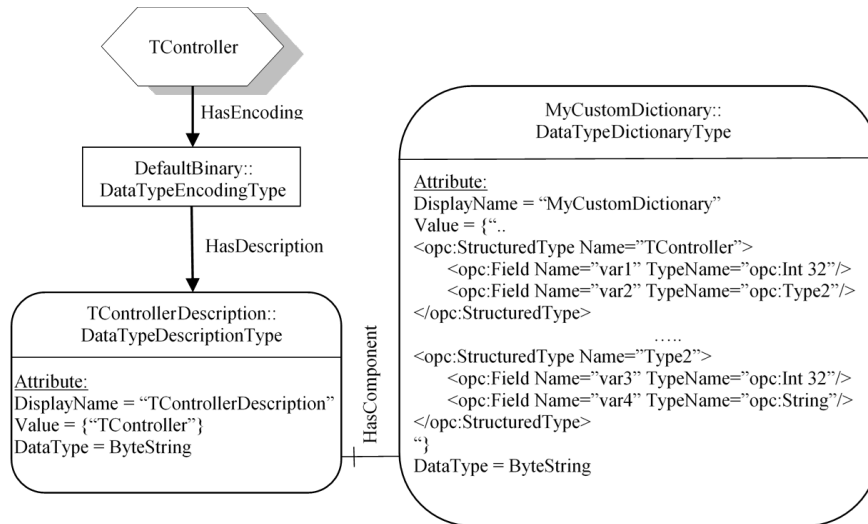
***Figure 5.10:*** *TController DataType*

*/data-sets/1/nodes/2-79*, assuming that the OPC UA Server owning the AddressSpace shown by Figure 5.9 refers to the dataset-id=1. As shown by Figure 5.7, after having assured the existence of an OPC UA Client instance and an active and secure OPC UA Session, the OPC UA Web Platform performs the OPC UA Read and Browse Service requests. Both OPC UA Services have the OPC UA NodeId with ns=2 and i=79 as argument.

OPC UA Read Service allows the platform to retrieve the information needed to fill the properties node-id, Name and Type (i.e. "2-79", "Controllers" and "folder", respectively).

OPC UA Browse Service allows the platform to retrieve the information about the Hierarchical References starting from the OPC UA Starting Node. In particular, the information obtained for each Hier-

87

archical Reference are node-id, Name and Type of the target Node and the Relationship between target and OPC UA Starting Node. Considering Figure 5.9, only the two Organizes References starting from Controllers OPC UA Node are found. For the Organizes Reference pointing the OPC UA Node Controller1, the information obtained to fill the properties node-id, Name Type and Relationship are "2-80", "Controller1", "variable" and "Organizes", respectively. For the Organizes Reference targeting the OPC UA Node Controller2, the information obtained are the same as the previous one with the exception of node-id which is equal to "2-81".

Information coming from OPC UA Read and Browse Services, are used to create a JSON document compliant to the schema shown by Listing 5.1. Considering the example just explained, the JSON document shown by Listing 5.1 is obtained and sent to the Web User inside the GET response.

```
{
  "node-id": "2-79",
  "Name": "Controllers",
  "Type": "folder",
  "Edges": [
    {
      "node-id": "2-80",
      "Name": "Controller1",
      "Type": "variable",
      "Relationship": "Organizes"
    },
    {
      "node-id": "2-81",
      "Name": "Controller2",
```

```
      "Type": "variable",
      "Relationship": "Organizes"
   }
  ]
}
```

***Listing 5.4:*** *JSON document received by the Web User for GET on*
*"/data-sets/1/nodes/2-79"*

In the following, another example of GET request will be pre-
sented, involving the Structured DataType TController described by
Figure 5.10. Let's suppose to make the GET request on the resource
"/data-sets/1/nodes/2-80". In this case, the OPC UA Starting Node
is a Variable Node. OPC UA Read Service on this OPC UA Node
allows the platform to fill the properties node-id, Name, Type with
"2-80", "Controller1" and "variable", respectively. As this OPC UA
Node is a DataVariable, its representation provided by the platform in
the GET response features the properties Status, MinimumSampling-
Interval, DeadBand, Value and ValueTypeSchema.  Status is set to
"Good" and MinimumSamplingInterval is set to 500. Deadband is set
to "none" as the platform recognise that the OPC UA DataVariable
Node has not a numeric DataType and does not feature an EURange.
In order to fill the ValueTypeSchema property, the following procedure
will be performed by the OPC UA Web Platform.

On the basis of the DataType attribute of the OPC UA Starting
Node, the TController NodeId is obtained (as shown by Table 5.4).
The OPC UA Browse Service is called by the OPC UA Web Platform,
starting from TController OPC UA Node, following the HasEncoding
Reference and looking for the DefaultBinary DataTypeEncodingType
Object.  Another call of OPC UA Browse Service from this node al-

89

lows to reach the TControllerDescription Variable; as said in Section 2.1.5, its Value attribute contains the entry point of the DataType-DictionaryType Variable (i.e., MyCustomDictionary in Figure 5.10) describing the structure of the TController DataType. On the basis of the information obtained about this structure, the JSON Schema shown by Listing 5.5 is built and used to fill the ValueTypeSchema property of the Variable Node.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "var1": {"type": "integer"},
    "var2": {
      "type": "object",
      "properties": {
        "var3": {"type": "integer"},
        "var4": {"type": "string"}
      },
      "required": ["var3", "var4"],
      "additionalProperties": false
    }
  },
  "required": ["var1", "var2"],
  "additionalProperties": false
}
```

***Listing 5.5:*** *JSON Schema for ValueTypeSchema property in the case of TController Structured DataType*

TController DataType structure (described by the Value attribute of MyCustomDictionary Variable in Figure 5.10) is used to decode the

binary stream representing the Controller1 Value attribute as seen in Table 5.4. In this example, it has been assumed that "var1" has a value of 10 and "var2" contains the field "var3" and "var4" with the values 15 and "config", respectively. Using these values, the JSON document shown by Listing 5.6 is built.

```
{
  "var1": 10,
  "var2": {
    "var3": 15,
    "var4": "config"
  }
}
```

***Listing 5.6:*** *The JSON document built using the current Controller1 Value*

Using all the information collected until now, the OPC UA Web Platform is able to build the JSON document describing the state of the Variable Node representing the OPC UA DataVariable Controller1, according to the JSON Schema of Listing 5.1. It is shown by Listing 5.7 and it is passed to the Web User through the GET response body.

```
{
  "node-id": "2-80",
  "Name": "Controller1",
  "Type": "variable",
  "Value": filled with the JSON document of Listing 5.6,
  "ValueTypeSchema": filled with the JSON Schema of
      Listing 5.5,
```

```
  "Status": "good",
  "Deadband": "None",
  "MinimumSamplingInterval": 500,
  "Edges": []
}
```

**Listing 5.7:** *JSON document received by the Web User for GET request on "/data-sets/1/nodes/2-80"*

As it is possible to see from Listing 5.7, the Edges property is an empty array, as no references start from the OPC UA Starting Node. Listing 5.7 also points out that the Value property is filled with the JSON document shown by Listing 5.6 and the ValueTypeSchema is filled with the JSON Schema of Listing 5.5.

### 5.3.4   Updating value of Variable Nodes

It has been assumed to allow the Web User to update only the Value property of a Variable Node. This update is possible sending a POST request to the resource *"/data-sets/{dataset-id}/nodes/{node-id}"*. Web User must specify the value to be updated in the body of the POST request.

It is required that Web User has previously received the state of the Variable Node through the GET request on the same resource *"/data-sets/{dataset-id}/nodes/{node-id}"*. In particular, the Web User has to know the JSON Schema contained in the ValueTypeSchema property of the Variable Node to update. The new value to be updated must be a JSON document compliant with this JSON Schema. A value not compliant will produce an error condition notified to the Web User by the OPC UA Web Platform.

For each POST request, the OPC UA Web Platform use to the OPC UA Write Service, passing the new value to be set. This value must be encoded by the OPC UA Web Platform according to the relevant DataType. To this aim, the OPC UA Web Platform has to obtain the NodeId of the OPC UA DataType Node (e.g., MyType or TController DataTypes in Figures 2.3 and 5.10). This is realised using the OPC UA Read Service on the OPC UA DataVariable Node and reading the DataType attribute. Once the NodeId has been obtained, the platform can access to the DataType Node and can understand if it is a Built-in or a Structured DataType.

If it is a Built-in DataType, the new value to be updated is converted from the JSON base type to the standard data type relevant to the Built-in DataType. The value obtained is inserted into the OPC UA Write Service request.

If the OPC UA DataType is a Structured DataType, OPC UA specification requires that the value to be inserted into the OPC UA Write Service request is encoded inside a particular object called *ExtensionObject* [27]. Furthermore, it is required that this object must contain the NodeId of the DataTypeEncodingType Object associated to the Structured DataType (e.g., DefaultBinary Object in Figures 2.3 and 5.10). This NodeId can be easily obtained starting from the OPC UA Structured DataType Node (e.g., MyType and TController Nodes in Figures 2.3 and 5.10, respectively) and following the HasEncoding Reference by the OPC UA Browse Service.

Once the OPC UA Web Platform receives the OPC UA Write response, the relevant state will be returned to the Web User in the body of the POST response.

### 5.3.5   Monitoring Variable Nodes

A Controller resource for each data set has been defined in order to allow the Web User to activate an asynchronous transmission mechanism of the value and/or the status relevant to a particular Variable Node; as said before, it has been assumed that the asynchronous transmission occurs when a change in the value and/or the status of the Node and, optionally, when a change in the value according to absolute or percent deadband is detected.

The controller resource has the URI *"/data-sets/{dataset-id}/monitor"*. A POST request on this resource allows the Web User to enable notification of changes of value and/or status of one or more Variable Nodes inside a specific data set. Notifications are published in a Topic on a specific Broker. The Web User has to specify the following information inside the body of the POST request:

- *ElementToMonitor*: it is an array of elements, each of which is made up by the following items:

  - *node-id*: it specifies the Variable Node in the specified data set, for which asynchronous transmissions are requested.

  - *SamplingInterval*: it specifies the desired sampling interval at which a Variable Node is checked to detect changes in the value and/or status, and optionally to detect changes in the value according to absolute or percent deadband.

  - Deadband: it allows to specify the requested deadband; it assumes one of these values: "absolute", "percent", "none".

  - DeadbandValue: it allows to specify the value of the absolute or percent deadband. It is considered only if Deadband

parameter is set to "absolute" or "percent".

- *BrokerURL*: it is a string containing the concatenation of the technology used (e.g., SignalR, MQTT) and the address of the Broker chosen by the Web User for asynchronous communication.

- *Topic*: it allows to specify the Topic/Group where notifications will be published; Web User will subscribe on the same Topic/Group on the Broker in order to receive the asynchronous updates.

It has been assumed that for each POST request, the OPC UA Web Platform is in charge to create an OPC UA Subscription inside the OPC UA Server relevant to the data set specified by the Web User; this OPC UA Subscription is associated to the couple Topic/Broker specified by the Web User. An internal table is used by the OPC UA Web Platform to this aim; in this table, information about OPC UA Subscription and relevant OPC UA Server is associated to each couple Topic/Broker.

Starting from the moment the POST request succeeds, all the Notifications produced inside the OPC UA Subscription are published on the Topic of the specified Broker. This asynchronous information flow may be cancelled by making a POST request on another controller resource described in the Section 5.3.6.

Fulfilment of the POST request is realised by the OPC UA Web Platform through the use of the OPC UA CreateSubscription and CreateMonitoredItems Services [8, 26], according to the procedure shown by Figure 5.11.
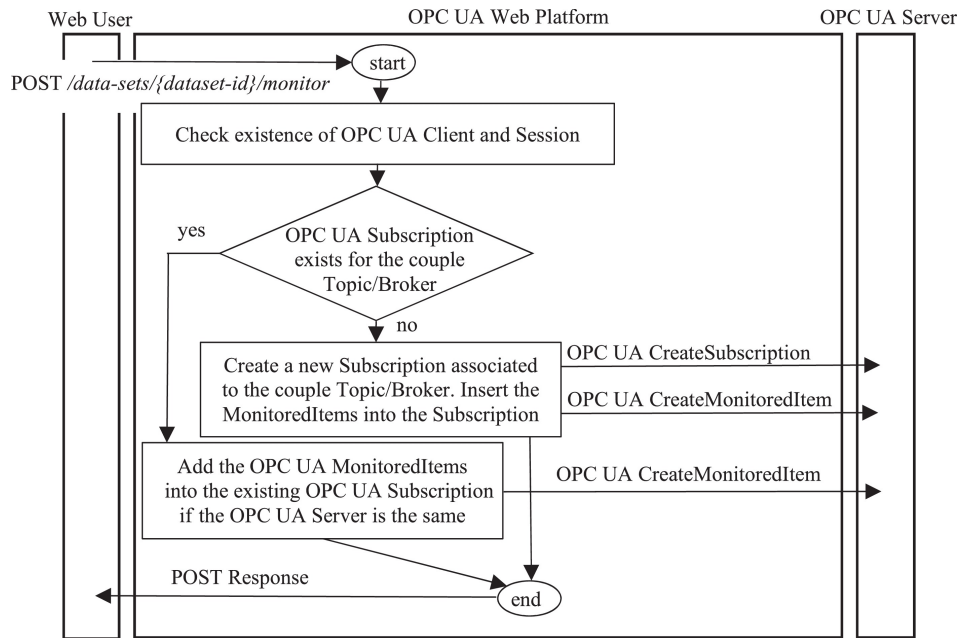
**Figure 5.11:** *Realization of Monitor through OPC UA Services*

The OPC UA Web Platform checks the existence of OPC UA Client and OPC UA Session, as shown by Figure 5.8. Then, it verifies the existence of the couple Topic/Broker in its internal table; if this verification fails, a new OPC UA Subscription is created inside the specified OPC UA Server using the OPC UA CreateSubscription Service. Moreover, OPC UA MonitoredItems are created inside the OPC UA Subscription, using the OPC UA CreateMonitoredItem Service; they are associated to the OPC UA Variables relevant to the Nodes passed by the Web User inside the ElementsToMonitor array. Furthermore, the SamplingInterval and DeadbandValue values are assigned to each MonitoredItem. It has been assumed to set the

PublishingInterval of the Subscription to the lowest value among the SamplingIntervals present in the ElementsToMonitor array.

If the Web User specifies a couple Topic/Broker already existing, the OPC UA Web Platform will retrieve information about the OPC UA Subscription associated to this couple. After having checked its belonging to the OPC UA Server relevant to the same data set specified by the Web User, OPC UA MonitoredItems are added to the OPC UA Subscription [26] for each Variable Node contained in the ElementsToMonitor array. Again, the OPC UA CreateMonitorItems Service is used to this aim. If the couple Topic/Broker refers to an OPC UA Subscription on a different OPC UA Server, an error condition is returned to the Web User.

In the case of successful creation of OPC UA MonitoredItems, POST response is sent to the Web User reporting the successful result.

In order to get NotificationMessages produced by the OPC UA Subscriptions created as just explained, the synchronous exchange of Publish requests and responses is realised between the OPC UA Web Platform and the OPC UA Server, as shown by Figure 5.12.
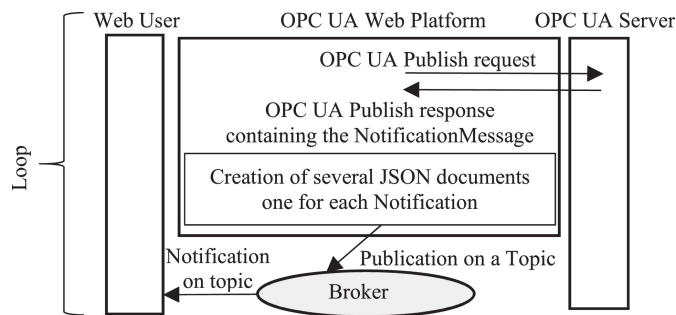


**Figure 5.12:** *Publishing Procedure through the Topic/Broker*

For each OPC UA Publish response received, the OPC UA Web Platform extracts the NotificationMessage and, in turns, the Notifications there contained. For each Notification, the OPC UA Web Platform creates a JSON document according to the JSON Schema shown by Listing 5.8.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "node-id": {"type": "string"},
    "Name": {"type": "string"},
    "Value": {},
    "ValueTypeSchema": {"type": "object"},
    "Status": {"enum": ["good", "uncertain", "bad"]}
  },
  "required": ["node-id", "Name", "Value", "ValueTypeSchema
      ", "Status"],
  "additionalProperties": false
}
```

***Listing 5.8:*** *JSON Schema for Variable value and/or status notification on the Topic/Broker*

Value property is relevant to the content of each Notification and ValueTypeSchema contains the description of the type relevant to the Value property; Value and ValueTypeSchema are filled as already seen in the previous sections.

Each JSON document so produced is published independently on the Topic of the Broker specified by the Web User.

Figure 5.12 points out the presence of a loop, as the entire sequence

of actions shown in the figure is cyclically repeated until the publication on the Topic/Broker is cancelled as explained in the Section 5.3.6.

#### 5.3.5.1 Case Study

Considering the simple Information Model shown by Figure 5.9, let us assume that Web User desires to activate the asynchronous transmissions of value and/or status of Controller1 Node using the SamplingInterval of 500 ms. Furthermore, let us assume that he wants to receive the notifications for changes in value and/or status on a MQTT Topic defined in a specific MQTT Broker. In order to do this, the Web User makes a POST request on the resource */data-sets/1/monitor"* setting the parameters as follow: ElementsToMonitor array contains a single element where node-id is set to "2-80", SamplingInterval is set to 500 ms, Deadband is set to "none", DeadbandType is not present (i.e., set to null), BrokerURL is set to a concatenation of the string "MQTT" and the address of the MQTT Broker specified by the Web User, and Topic is set to the string that identifies the Topic on the MQTT Broker which the Web User is subscribed.

According to Figure 5.11, the OPC UA Web Platform checks if an OPC UA Subscription exists for the Topic/Broker couple specified in the Monitor request and for the OPC UA Server relevant to the data set specified by the Web User. Assuming that this does not exist, a new OPC UA Subscription is created inside the OPC UA Server; a MonitoredItem is created inside it too. The OPC UA MonitoredItem is linked to the OPC UA Controller1 Node. The relevant SamplingInterval is set to 500 ms and the filter is set to DataChangeFilter with no deadband. PublishingInterval of the novel OPC UA Subscription

is set to 500 ms.

Once the OPC UA MonitoredItem has been created by the OPC
UA Web Platform, the publication of Notifications inside the Subscrip-
tion starts at a rate given by the inverse of the SamplingInterval. All
the Notifications produced at every PublishingInterval are collected
into a NotificationMessage. Therefore, at every PublishingInterval
only one Notification is produced and is inserted in the Notification-
Message. According to Figure 5.12, the OPC UA Web Platform will
start to trigger the transmission of each NotificationMessage by using
the OPC UA Publish Service.

For each Publish request sent to the OPC UA Server, the OPC UA
Web Platform receives a Publish response containing a Notification-
Message. The OPC UA Web Platform extracts the only Notification
contained in the NotificationMessage and create a JSON document
according to the JSON Schema shown by Listing 5.8.

Listing 5.9 shows the JSON document produced.

```
{
  "node -id": "2-80",
  "Name": "Controller1",
  "Value": filled with the JSON document of Listing 5.6,
  "ValueTypeSchema": filled with the JSON Schema of
      Listing 5.5,
  "Status": "good"
}
```

***Listing 5.9:*** *Example of JSON document created for the monitoring
of the Node Controller1 (ns=2;i=80)*

As it is possible to see, it has been assumed that the current Value

is given by the JSON document shown by Listing 5.6. The Value-TypeSchema is the same of that shown by Listing 5.5. The JSON document shown in Listing 5.9 is published over the MQTT Topic defined in the MQTT Broker specified by the Web User.

### 5.3.6   Stop Monitoring Variable Nodes

Web User who has previously activated a monitor service, may cancel it at any time. To accomplish this task, the controller resource *"/data-sets/{dataset-id}/stop-monitor"* has been defined for each data set. The Web User makes a POST request, providing the BrokerURL and Topic as defined in Section 5.3.5.

For each POST request, the OPC UA Web Platform looks for the couple Topic/Broker in its internal table. If it is present, the relevant OPC UA Subscription is deleted through the OPC UA DeleteSubscription Service [8, 26]. According to the OPC UA specifications, successful completion of this service causes all MonitoredItems that use the Subscription to be deleted. In this way, publication on the Topic/Broker is cancelled.

## 5.4   OPC UA Web Platform Implementation

The OPC UA Web Platform described in this chapter was implemented and the relevant source code was made available on GitHub [42].

Implementation was based on open source technologies able to cre-

ate cross platform web applications easily deployable on the main server operating systems. In particular, the implementation of the OPC UA Web Platform was achieved using Microsoft .NET Core and ASP.NET Core frameworks [50]. The former is an open source implementation of the .NET Standard Library, born and evolved in the last years with the aim to enable the development of cross-platform applications; its source code is available on GitHub [51]. The latter is a server-side web application framework. An ASP.NET Core Web Application is deployable on the majority of the modern platforms.

The implementation of the OPC UA Client was based on the Open Source OPC UA cross platform .NET Core stack available on GitHub [52], which was developed for .NET Core applications.

For the authentication and authorization, JSON Web Tokens (JWT) open standard was the solution adopted [53]. The JWT mechanisms was developed through the use of libraries provided by Microsoft in the .NET Core framework.

Two different implementations of the Publisher/Subscriber asynchronous communications shown by Figures 5.1 and 5.12 were realised. The relevant technologies adopted are Microsoft SignalR and MOSCA MQTT, respectively [44, 46].

# SIX

# INTEGRATION BETWEEN OPC UA AND OCF

As pointed out in the Introduction (Chapter 1), this chapter describes a proposal of integration between OPC UA and IoT/IIoT ecosystems. Among the current IoT/IIoT ecosystems, OCF has been chosen for the integration with OPC UA. In particular, the proposal aims to realise a mapping between OPC UA Information Model and OCF Resource Model.

Several papers and articles describe the mapping proposed. The mapping from OPC UA Information Model to OCF Resource Model is described in [54, 55, 56]. The mapping from OCF Resource Model to OPC UA Information Model is described in [57]. The complete bidirectional mapping is described in [58].

This chapter is organized as follows: the mapping from OPC UA Information Model to OCF Resource Model is described in Section 6.1; the mapping from OCF Resource Model to OPC UA Information Model is described in Section 6.2. Finally, the possible contribution

of the proposal will be discussed in Section 6.3.

# 6.1   Mapping from OPC UA Information Model to OCF Resource Model

This section presents the mapping from OPC UA Information Model to OCF Resource Model. Mapping may involve the entire OPC UA AddressSpace of a specific OPC UA Server or its subset.

## 6.1.1   Mapping idea

The proposal of mapping from OPC UA Information Model to OCF Resource Model is based on the use of an OCF Device, named *OPC UA Device*, belonging to the ad-hoc defined "x.opc.device" Device Type. Such Device is made up by several OCF Resources which may be related each other by OCF Links. Each OCF Resource belongs to one of three ad-hoc defined Resource Types: "x.opc.object", "x.opc.datavariable" and "x.opc.method". The Device Type and the Resource Types just introduced will be detailed in Section 6.1.2, 6.1.3, 6.1.4 and 6.1.5.

In order to better understand this section, an example of the proposed mapping is provided in Figure 6.1.

On the left of Figure 6.1, a simple subset of an OPC UA AddressSpace is shown. It is made up by an OPC UA Object (*MyObject1*) and a Folder (*MyFolder*). The Object *MyObject1* is made up by another Object (*MyNestedObject1*), by an OPC UA DataVariable (*MyDataVariable1*) with a Property specified by the OPC UA Property Node called
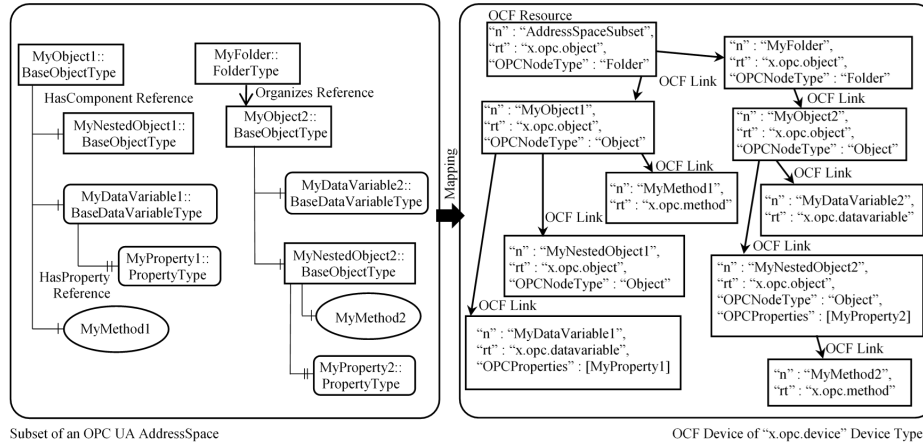
***Figure 6.1:*** *Example of the proposed mapping from OPC UA to OCF*

*MyProperty1*, and by an OPC UA Method (MyMethod1). The Folder *MyFolder* contains an OPC UA Object (*MyObject2*), which is in turn made up by an OPC UA DataVariable (*MyDataVariable2*) and by an OPC UA Object (*MyNestedObject2*). This last OPC UA Node features an OPC UA Property (*MyProperty2*) and is made up by an OPC UA Method (*MyMethod2*). OPC UA References are present to link these Nodes, as shown by the figure.

The subset of OPC UA AddressSpace shown on the left of Figure 6.1 is mapped into OCF by the OCF Device of "x.opc.device" Device Type present on the right side. Inside the OCF Device, the OCF Resources and OCF Links used for the mapping are shown. An OCF Resource is made up by several properties according to its type, which is represented by the value of the mandatory "rt" property. In the figure, properties of OCF Resources are shown using JSON formalism.

As the definition of an OCF Device Type must include mandatory Resources that must be implemented by all Devices of this type, the here proposed Device Type "x.opc.device" has a mandatory Resource, named "AddressSpaceSubset", of "x.opc.object" type. It is worth noting that this OCF Resource has not correspondence in the OPC UA Address Space and its aim is to aggregate the Resources mapping the OPC UA Nodes. Figure 6.1 shows this OCF Resource, featuring "rt" property with the value "x.opc.object" and the "n" property filled using the name "AddressSpaceSubset". More details will be provided in Section 6.1.2.

An OPC UA Node belonging to the Object NodeClass is mapped into OCF Resource of "x.opc.object" Resource Type; such a OCF Resource features the "rt" property set to the value "x.opc.object" and the "n" property filled using the DisplayName of the OPC UA Object Node represented. As an example, consider the OPC UA Object MyObject1; the relevant OCF Resource mapping this Node is the one holding the property "n" set to "MyObject1".

The OPC UA Node of OPC UA ObjectType NodeClass defining the type of an Object Node is mapped into a property of the OCF Resource mapping the OPC UA Object Node and defined by the "x.opc.object" ResourceType. This property is named "OPCNodeType" and is set to a string whose value may be "Folder" or "Object", according if OPC UA Object Node is defined by FolderType ObjectType (or its subtype) o by any other ObjectType, respectively. For instance, consider the OPC UA Object MyObject1 and the relevant OCF Resource named "MyObject1". Its "OPCNodeType" property is set to "Object" as the OPC UA Object MyObject1 is defined by the BaseObjectType ObjectType. On the other hand, the OPC

UA Node MyFolder is mapped into the OCF Resource with property
"n" set to "MyFolder" and "OPCNodeType" set to "Folder" as the
OPC UA Node is of FolderType ObjectType. Finally, considering the
OCF Resource named "AddressSpaceSubset", it has been assumed to
set the "OPCNodeType" property to "Folder", as its behaves like an
OPC UA Node of FolderType ObjectType organising several Nodes.
More details about "x.opc.object" Resource Type will be provided in
Section 6.1.3.

An OPC UA Node belonging to the Method NodeClass is mapped
into OCF Resource of "x.opc.method" Resource Type. In this case,
again property "n" is filled using the DisplayName of the OPC UA
Method Node, whilst property "rt" is set to "x.opc.method". In Fig-
ure 6.1, the OPC UA Nodes MyMethod1 and MyMethod2 are mapped
into the OCF Resources with property "n" set to "MyMethod1" and
"MyMethod2", respectively. More details about "x.opc.method" Re-
source Type will be provided in Section 6.1.5

Mapping of OPC UA Nodes belonging to the Variable NodeClass
depends on the kind of Variable.

An OPC UA DataVariable Node is mapped into OCF Resource of
"x.opc.datavariable" Resource Type, thus its "rt" property is set to
"x.opc.datavariable". As an example, consider the OPC UA DataVari-
able MyDataVariable1 and the relevant OCF Resource with property
"n" set to "MyDataVariable1" (i.e. the DisplayName of the OPC UA
Node). More details about "x.opc.datavariable" Resource Type will
be provided in Section 6.1.4.

An OPC UA Property Node is not mapped into an OCF Resource
but it is mapped as value of a property of the OCF Resource rep-
resenting the OPC UA Node to which the OPC UA Property be-

longs, named "OPCProperties". In other words, it has been assumed that the OCF Resource mapping an OPC UA Object, DataVariable or Method Node and connected to one or more OPC UA Property Nodes by HasProperty Reference, exposes a property "OPCProperties". This last has been realised as an array of JSON objects, each of which represents an OPC UA Property Node. For instance, considering the OCF Resources modelling the OPC UA MyDataVariable1 and MyNestedObject2, the "OPCProperties" array property of such resources contains an object called MyProperty1 and MyProperty2, respectively. Each object models an OPC UA Property Node, as said before. In both case the "OPCProperties" property is an array of just one element. More details about details about "OPCProperties" property and about the JSON objects there contained will be provided in Section 6.1.3.

About the mapping of OPC UA VariableType NodeClass, the relevant information carried out is used to distinguish whether a Variable Node is an OPC UA DataVariable or a Property Node, so that it could be mapped in the proper way, as described before.

As described in Section 2.1.5, the type of the Value attribute for OPC UA Variable Node (DataVariable or Property) is defined by a DataType Node belonging to the DataType NodeClass (e.g., MyType Node shown by Figure 2.3). It has been assumed to map the Value attribute of a Variable Node and the relevant DataType Node using an ad-hoc defined property inside the OCF representation of the OPC UA Variable Node. This property holds the representation of the Value attribute according to the relevant OPC UA DataType. For OPC UA DataVariable Node, the property is named "OPCValue" (not shown in Figure 6.1) and belongs to the OCF Resource representing

the OPC UA DataVariable Node. In the case of an OPC UA Property Node, the relevant JSON object contained in the "OPCProperties" array features a property called "opc-property-value" (not shown in Figure 6.1). Section 6.1.6 will give technical details about "OPCValue" and "opc-property-value" properties and the representation of the Value attribute according to the relevant OPC UA DataType.

An OPC UA Reference is defined by an OPC UA Node belonging to the ReferenceType NodeClass. OPC UA References are mapped in OCF according to their ReferenceType.

An OPC UA Reference belonging to Organizes or HasComponent ReferenceTypes is mapped as an OCF Links. Considering Figure 6.1, Organizes and HasComponent References are directly mapped into OCF Links owned by OCF Resources modelling OPC UA Folder, Object or DataVariable. It is important to point out that the meaning of the OPC UA Organizes and HasComponent References is not lost in the mapping; in fact, the kind of relationship realised by an OCF Link between two OCF Resources may be distinguished by the "OPCNodeType" property of the OCF Resource from which this OCF Link starts.

An OPC UA Reference belonging to the HasTypeDefinition ReferenceType is used to distinguish between OPC UA Object Node of FolderType ObjectType or of any other ObjectType, and is mapped by "OPCNodeType" property.

An OPC UA Reference belonging to the HasProperty ReferenceType is involved in the mapping of OPC UA Properties, to fill the "OPCProperties" array, as discussed previously.

As said in Sections 2.1.2 and 2.1.3, OPC UA defines Events and uses References belonging to the HasEventSource ReferenceType to

identify OPC UA Nodes acting as Event source. It has been assumed that, if an OPC UA Node reached by an OPC UA HasEventSource Reference is represented as an OCF Resource, all the OCF Links having such a Resource as target will be marked with the Observable flag in the "p" parameter [37]. Figure 6.2 shows on the left a subset of an OPC UA AddressSpace containing an OPC UA Object and a DataVariable Node; the latter is at the same time a component of the OPC UA Object and an Event source. On the right the mapping into OCF is shown; the OCF Link models the HasComponent Reference, as said before. The HasEventSource Reference is modelled setting the Observable bit in the "p" parameter for this OCF Link.
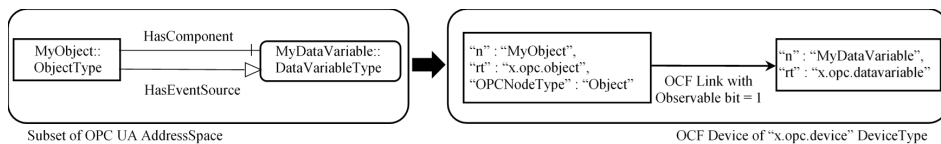


***Figure 6.2:*** *Mapping of OPC UA HasEventSource Reference*

On the basis of what said about OPC UA References, it is clear that some of them are not mapped into elements of the OCF Resource Model; in the following, it will be pointed out that missing mappings occur because they are not necessary and no information is lost in these cases.

OPC UA References belonging to the HasNotifier ReferenceType are not mapped as Events are not visible in the OPC UA AddressSpace, as pointed out in [8].

OPC UA References belonging to HasModellingRule, HasParent-Model and GeneratesEvent ReferenceTypes are only used when new instances of OPC UA Nodes must be created inside the OPC UA

AddressSpace. For this reason, their mapping to OCF is meaningless, because the aim of the mapping here proposed is obtaining the representation of each information already existing in OPC UA AddressSpace into OCF ecosystem.

As said before, in the case of OPC UA Variable Node the Value attribute depends on the relevant OPC UA DataType, whose description can be reached in the AddressSpace using HasEncoding and HasDescription References, as shown by Figure 2.3. Description of OPC UA DataType is included inside each OCF Resource modelling an OPC UA DataVariable Node or inside the JSON object contained in the "OPCProperties" array in the case of OPC UA Property Node, as briefly explained before and described in great details in Section 6.1.6. On the basis of this choice, the mapping of HasEncoding and HasDescription References is not needed at all.

## 6.1.2   "x.opc.device" Device Type

The Device Type "x.opc.device" specifies that a Device of this type shall expose a mandatory OCF Resource of the "x.opc.object" Resource Type named "AddressSpaceSubset". This Resource does not map any actual OPC UA Node but its aim is to aggregate OCF Resources mapping OPC UA Nodes. The presence of this Resource is due to the constraint that a Device Type definition must include mandatory Resources for its Device instance [37]. Table 6.1 gives a description of this Device Type. As seen, Figure 6.1 shows an example of a Device belonging to this type; as it is possible to see in the right part of the figure, the Device exposes the mandatory OCF Resource named "AddressSpaceSubset" linking all the OCF Resources used to

represent the OPC UA Nodes shown on the left of Figure 6.1.

**Table 6.1:** *"x.opc.device" Device Type*

| Device Name | Device Type | Required Resource Name | Required Resource Type |
|---|---|---|---|
| OPC UA Device | "x.opc.device" | AddressSpaceSubset | "x.opc.object" |

According to the overview about OCF Device Type done in Section 4.1, a Device of "x.opc.device" type will include the three mandatory OCF Resources described in Section 4.1, i.e. addressed by "oic/p", "oic/d" and "oic/res" URI. Among them, the OCF Resource identified by the "/oic/d" URI, used to represent the OCF Device, features the "rt" property containing the "x.opc.device" Device Type defined. The other properties of this Resource are strictly related to the OCF specification; hence, their values are assigned according to OCF and there is no need to use information from OPC UA to fill them. The only exception is represented by the optional "n" property, which may be filled by a string aimed to give some information about the mapping between OPC UA and OCF: for instance, it may assume a string value able to give general information about the content of the subset of OPC UA AddressSpace mapped.

### 6.1.3   "x.opc.object" Resource Type

The Resource Type "x.opc.object" is aimed to map OPC UA Object Nodes of any ObjectType. As explained in Section 4.1, every OCF Resource shall implement the common properties defined by the "oic.core" Resource Type and summarised by Table 4.1. In the case

112

of "x.opc.object" Resource Type, the property "n" is filled using the
OPC UA attribute DisplayName; the Resource Type property ("rt")
is filled with the name of the Resource Type used to represent the
Resource (i.e. "x.opc.object"); the identifier of the Resource ("id")
is filled using a string representation of the OPC UA NodeId of the
mapped OPC UA Object Node. Consider a NodeId with integer in-
dex: the string representation consist of a concatenation of the value
$ns$ (NamespaceIndex) and value $i$ (Identifier) separated by a dash;
for example, the NodeId made up by ns=2 and i=12 becomes "2-12".
The interface property ("if") is strictly related to the OCF RESTful
services: for this reason its value depends on the implementation of
the Device and has nothing to do with the mapping here treated.

In addition to the OCF common properties explained above, a Re-
source belonging to the Resource Type "x.opc.object" exposes several
properties, as shown by Table 6.2.

**Table 6.2:** *OCF Properties defined by "x.opc.object" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| Common Properties | See "oic.core" Resource Type in Table 4.1 | | |
| OPCNodeType | string | Yes | It specifies if the Resource represents an OPC UA Node of FolderType or any other ObjectType |

### *Table 6.2 Continued*

| Property Name | OCF data type | Mandatory | Description |
| --- | --- | --- | --- |
| OPCDescription | string | No | The textual description of the mapped OPC UA Object Node |
| OPCProperties | array of objects | No | An array of JSON objects representing OPC UA Property Nodes |
| links | array of OCF Links | No | An array of OCF Links mapping OPC UA References |

An instance of the "x.opc.object" is a Collection Resource, since it can be linked to other Resources through OCF Links; this explains the presence of the property "links" in Table 6.2. It is an array containing the OCF Links mapping OPC UA Organizes and HasComponent References starting from the OPC UA Object Node represented by the current OCF Resource. For each Link, the "href" property is filled with the target URI of the Resource corresponding to the OPC UA Node pointed by the OPC UA Reference modelled by the Link. Furthermore, the parameter "p" of the Link will have the Observable flag marked, if the OCF Resource reached by the Link represents an OPC UA Node targeted by an HasEventSource Reference.

A property named "OPCNodeType" is defined as a string in order

to differentiate whether the OCF Resource represents an OPC UA Object Node whose type may belong to the FolderType ObjectType or to any other ObjectType. HasTypeDefinition Reference starting from the OPC UA Object Node is used in the mapping to obtain this information. According to these two cases, this property can assume the values "Folder" and "Object", respectively.

A string property named "OPCDescription" is defined in order to map the Description attribute of the OPC UA Object Node represented.

A property named "OPCProperties" is defined in order to represent the OPC UA Property Nodes connected to the represented OPC UA Node by HasProperty References; it is defined as an array of JSON objects. For each HasProperty Reference, a JSON object mapping the target OPC UA Property Node is created and inserted in the JSON array. As seen in Section 6.1.1, Figure 6.1 shows a very simple example where each of two OPC UA Nodes MyDataVariable1 and MyNestedObject2 features a single HasProperty Reference starting from it. For this reason, each OCF Resource representing one of these Nodes contains the array property named "OPCProperties". In both cases, this property has only a JSON object representing the relevant OPC UA Property of the Node (i.e. the OPC UA Nodes MyProperty1 and MyProperty2). Table 6.3 shows the properties of the JSON object contained in the OPCProperties array.

***Table 6.3:*** *JSON object of the OPCProperties array*

| Property Name | OCF data type | Mandatory | Description |
| --- | --- | --- | --- |

## *Table 6.3 Continued*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| opc-property-name | string | Yes | It contains the BrowseName attribute of the OPC UA Property Node |
| value-type | string | Yes | It specifies the OCF data type used for the representation of the Value attribute of OPC UA Property Node. |
| opc-property-value | <type> | Yes | The Representation of the Value attribute of OPC UA Property Node. Its OCF data type <type> is specified by "value-type" property. |
| enum-values | array of objects | No | It contains the enumeration values if the OPC UA Value attribute mapped is an enumeration. |
| num-dimensions | array of numbers | No | If the OPC UA Value attribute mapped is an array, this property specifies the relevant dimensions |

*Table 6.3 Continued*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| innermost-type | string | No | If the OPC UA Value attribute mapped is an array, this property specifies the OCF data type mapping the OPC UA DataType of the elements of the array. |

Table 6.3 points out that each JSON object representing an OPC UA Property Node is made up at least by three mandatory properties.

The first mandatory property is named "opc-property-name" and is filled using the BrowseName attribute of the OPC UA Property Node, since this attribute is very suitable to define the semantic of the property. The OCF data type used to represent this property is string [37].

The other two mandatory properties are "value-type" and "opc-property-value". The latter is used to represent the OPC UA Value attribute of the OPC UA Property using a OCF data type (i.e. <type>) specified by the former. The choice of the OCF data type used depends on the OPC UA DataType of the OPC UA Property: Section 6.1.6 will point out how this mapping has been realised.

If the Value attribute of the OPC UA Property Node is an array, the "value-type" property is filled by the string "array" (i.e. <type> is OCF data type array). In this case the two optional properties "num-dimensions" and "innermost-type" are present specifying the dimensions and the OCF data type of each element of the array, re-

spectively. So that, the "opc-property-value" will contain the representation of the original OPC UA array. Again, representation of each value of the array will be described in Section 6.1.6.

If the Value attribute of the OPC UA Property Node is an enumeration, the "value-type" property is filled by the string "number"(i.e. <type> is OCF data type number) and "opc-property-value" contains the integer representation of the enumeration (as the OPC UA Property Value attribute is of the Int32 Built-in DataType). The enumeration values are inserted in the optional property "enum-values": this property is an array of JSON objects where each object contains one of the possible enumeration value. Representation of each value of the array will be described in Section 6.1.6.

It is worth noting that this Resource Type is used also to define a mandatory Resource for the "x.opc.device" Device Type, as seen in the previous subsection. As widely discussed, this mandatory Resource do not map any OPC UA Node. It has been assumed to fill its properties as specified in the following. The property "n" is set with the string "AddressSpaceSubset" (because "x.opc.device" define its Required Resource using this name) and the identifier of the Resource ("id") is not used as it is not mandatory. The properties "rt" is filled with "x.opc.object", obviously. Finally, "if" is filled as mandated by the Device implementation. The property "OPCNodeType" is filled with "Folder" because this mandatory Resource acts as aggregator for the OCF Resources representing OPC UA Nodes.

### 6.1.4 "x.opc.datavariable" Resource Type

OPC UA DataVariable Node is mapped as an instance of the OCF Resource Type "x.opc.datavariable", which is defined in this proposal and shown by Table 6.4.

***Table 6.4:*** *OCF Properties defined by "x.opc.datavariable" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| Common Properties | See "oic.core" Resource Type in Table 4.1 | | |
| value-type | string | Yes | It specifies the OCF data type used for the representation of the Value attribute of the OPC UA DataVariable Node. |
| OPCValue | \<type\> | Yes | The Representation of the Value attribute of OPC UA DataVariable Node. Its OCF data type \<type\> is specified by "value-type" property. |
| enum-values | array of objects | No | It contains the enumeration values if the OPC UA Value attribute mapped is an enumeration. |
| num-dimensions | array of numbers | No | If the OPC UA Value attribute mapped is an array, this property specifies the relevant dimensions |

*Table 6.4 Continued*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| innermost-type | string | No | If the OPC UA Value attribute mapped is an array, this property specifies the OCF data type mapping the OPC UA DataType of the elements of the array. |
| OPCProperties | array of objects | No | An array of JSON objects representing OPC UA Property Nodes |
| links | array of OCF Links | No | An array of OCF Links mapping OPC UA References |

Common properties of the "oic.core" Resource Type are filled as explained for "x.opc.object". The property "links" is present to contain the OCF Links mapping OPC UA HasComponent References starting from the modelled OPC UA DataVariable Node. The considerations done in the previous subsection for the properties "href" and for the Observable flag of the parameter "p" are valid in this case too.

A mandatory property named "OPCValue" will contain the representation of the Value attribute of OPC UA DataVariable Node according to the OCF data type mapping the OPC UA DataType of the Value attribute. Section 6.1.6 will point out how this mapping of data types is realised and how the representation of the Value attribute is realised. The OCF data type used for the representation of OPC UA DataVariable Node Value attribute is declared in the "value-type"

property. If the Value attribute is an array, the "value-type" property contains the string "array": in this case, the OCF data type of each element is specified by the "innermost-type" property. Furthermore, always in this case, the "OPCValue" property contains the representation of the original OPC UA array according to the OCF data type specified by the "innermost-type" property.

The property "num-dimension" is present only if "value-type" is array and it specifies the relevant dimensions.

The property "enum-values" is present only if "value-type" is number, hence the "OPCValue" is the index of the enumeration; if present, the property is realised as a JSON array containing the enumeration values coded as objects.

As for "x.opc.object", a property named "OPCProperties" is defined in order to represent each OPC UA Property Node connected to the relevant OPC UA DataVariable Node through the OPC UA HasProperty Reference. The same considerations made for the "x.opc.object" are valid also for "x.opc.datavariable" Resource Type.

### 6.1.5   "x.opc.method" Resource Type

Before description of the mapping of OPC UA Method Nodes, it is important to clarify what may be mapped and what is lost in the mapping. An OPC UA Method is an entity that is always associated with an Object whose presence is similar to a declaration of a function prototype in a high-level language. In order to invoke the method, it is necessary to use a specific OPC UA Service, named *Call*, where actual parameters are passed [26]. The concept of method does not exist in OCF. In OCF everything is intended as an interaction with

the state of an OCF Resource, performing both reading and/or writing operations. For this reason, the proposed mapping must be limited to map information about the existence of an OPC UA Method and its relevant properties (among which input/output parameters).

For the mapping of OPC UA Method Nodes, "x.opc.method" Resource Type has been defined in order to represent this kind of OPC UA Nodes as OCF Collection Resources. The properties of the Resource Type "x.opc.method" are summarised in Table 6.5.

**Table 6.5:** *OCF Properties defined by "x.opc.method" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| Common Properties | See "oic.core" Resource Type in Table 4.1 | | |
| OPCInputArg | array of objects | No | An array of JSON objects representing the input arguments of the Method. |
| OPCOutputArg | array of objects | No | An array of JSON objects representing the output arguments of the Method. |
| executable | boolean | Yes | It indicates if the Method is callable inside OPC UA environment. |
| OPCProperties | array of objects | No | An array of JSON objects representing OPC UA Property Nodes |

### Table 6.5 Continued

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| links | array of OCF Links | No | An array of OCF Links mapping OPC UA References |

Common properties are filled as explained for "x.opc.object". As done for the other Resource Types, the "links" property includes the list of OCF Links mapping OPC UA References starting from the OPC UA Method Node. The same setting descriptions about "href" and "p" properties done for the other Resource Types are applied also for this one.

Properties named "OPCInputArg" and "OPCOutputArg" are defined in order to represent the *InputArgument* and *OutputArgument* Properties of the OPC UA Method Node, respectively.

A boolean property named "Executable" is used to specify whether the OPC UA Method can be invoked by an OPC UA Client, based on the current value of the *Executable* and *UserExecutable* attributes of the OPC UA Method Node. As said at the beginning of this subsection, this property does not mean that the method is executable in the OCF ecosystem; it allows only to highlight if the Method is currently callable inside OPC UA environment.

As for "x.opc.object" and "x.opc.datavariable", a property named "OPCProperties" is defined in order to represent the OPC UA Property Nodes linked to the OPC UA Method Node by HasProperty References. As said, it is made up by a list of JSON objects, each representing an OPC UA Property Node. Description of each JSON

123

object is the same done for the other Resource Types seen before.

## 6.1.6 Mapping OPC UA DataType and OPC UA Variable Node Value attribute

Two important issues were left unsolved in the previous subsections. The first one is about how the OPC UA DataType relevant to the Value attribute of an OPC UA Variable Node is mapped into an OCF data type. The second issue is the representation of the Value attribute according to the OCF data type just found.

As said in Section 2.1.5, OPC UA DataTypes may be Built-in, Enumeration and Structured; array of elements belonging to these DataTypes may be defined too. It has been assumed that mapping of Built-in, Enumeration, Structured DataTypes and array into OCF data types is done by a two-steps process.

At the first step, the mapping is done from OPC UA DataType to JSON base type, according to the JSON DataEncoding defined by [27]. The two leftmost columns of Table 6.6 summarise this mapping; for each OPC UA DataType and array the relevant the JSON base type is given. For the definition of OPC UA DataTypes found in the Table please refers to [24].

***Table 6.6:*** *Mapping OPC UA DataTypes and array into JSON base types and OCF data types*

| OPC UA DataTypes and array | JSON base types | OCF data types |
|---|---|---|
| **Built-in**: Integer, Float, Double, StatusCode | Number | Number |

*Table 6.6 Continued*

| OPC UA DataTypes and array | JSON base types | OCF data type |
|---|---|---|
| **Enumeration** | Number | Number |
| **Built-in**: String, DateTime, Guid, ByteString, XmlElement | string | string |
| **Built-in**: Boolean | literal names *true* and *false* | boolean |
| **array** | array | array |

At the second step, the JSON base types shown in Table 6.6 are mapped to the relevant OCF data types according to [37]. As said in Section 4.1, OCF adopts all the JSON base types shown in the Table 6.6, with the exception of the JSON base type literal names *true* and *false* which are mapped into OCF data type boolean. Table 6.6 shows the final mappings into OCF data types in the rightmost column.

In the following, the overview about the representation of the Value attribute of the OPC UA Variable Node according to the OCF data type achieved as explained before, will be given for each kind of OPC UA DataType.

### 6.1.6.1   Built-in DataType

The current value of the Value attribute of OPC UA Variable Node (both Property and DataVariable) is encoded according to the OCF data type corresponding to the OPC UA DataType of the Value attribute, found on the basis of the content of Table 6.6. The value so obtained is assigned to the property "opc-property-value" in the case of OPC UA Property Node or to the property "OPCValue" in the case of OPC UA DataVariable Node. In both cases, the "value-type" property is set to the OCF data type found as said before.

### 6.1.6.2   Enumeration DataType

According to Table 6.6, the "value-type" property in Tables 6.3 and 6.4 is set to "number".

OPC UA Enumeration Value is mapped as an OCF data type number; the value so obtained is assigned to the property "opc-property-value" in the case of OPC UA Property Node, or to the property "OPCValue" in the case of OPC UA DataVariable Node. This value is the integer representation of the enumeration.

The property "enum-values" in Tables 6.3 and 6.4 is filled with an array of JSON objects containing the enumeration values. It has been assumed to define the JSON object shown by Table 6.7 to this aim.

***Table 6.7:*** *JSON object representing a single enumeration value*

| Property Name | OCF data type | Description |
|---|---|---|
| enumeration-index | number | The integer representation of an Enumeration |

*Table 6.7 Continued*

| Property Name | OCF data type | Description |
| --- | --- | --- |
| enumeration-value | string | A human-readable representation of the Value of the Enumeration |

As shown, this object is made up by two properties named "enumeration-index" and "enumeration-value"; the first contains the integer representation of the enumeration and the second is the relevant value of the enumeration. The values of these two properties are obtained from the mutual exclusive *EnumStrings* or *EnumValues* Properties of the OPC UA Enumeration DataType. For example, let us assume that the OPC UA Enumeration DataType features the EnumValues Property; as said in Section 2.1.5, it is an array of elements each of which has a structure given by Table 2.1. This elements are used to fill the JSON object shown in Table 6.7. In particular, the value of the property "enumeration-index" is obtained from the content of Value encoded as OCF number data type; the value of the "enumeration-value" property is obtained by the content of Display-Name encoded as OCF string data type. It has been assumed to not take into account the Description element shown by Table 2.1, as it may be empty.

### 6.1.6.3 Structured DataType

According to Table 6.6, the OCF data object is used to map the OPC UA Structured DataType. For this reason, the content of "value-type"

property in Tables 6.3 and 6.4 is a string containing the value "object".

As seen in Section 2.1.5, an OPC UA Structured DataType is made up by several Fields each featuring the *FieldName* and *TypeName* properties, specifying the name and the type of an OPC UA Field, respectively(see Figure 2.3). For this reason, the content of the properties "opc-property-value" and "OPCValue" in Tables 6.3 and 6.4 is a JSON object made up by a single property named "fields" which represents the Value attribute of the Variable Node. This property is an array of JSON objects, each of which is made up as shown by Table 6.8. Each JSON object models a single OPC UA Field of an OPC UA Structured DataType.

***Table 6.8:*** *JSON object representing a field of a Structured DataType*

| Property Name | OCF data type | Mandatory | Description |
| --- | --- | --- | --- |
| field-name | string | Yes | It contains the Field Name value of the OPC UA Field. |
| value-type | string | Yes | It specifies the OCF data type used for the representation of the OPC UA Field value. |
| field-value | <type> | Yes | The Representation of the OPC UA Field value. Its OCF data type <type> is specified by "value-type" property. |
| enum-values | array of objects | No | It contains the enumeration values if the OPC UA Field value mapped is an enumeration. |

### Table 6.8 Continued

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| num-dimensions | array of numbers | No | If the OPC UA Field value mapped is an array, this property specifies the relevant dimensions |
| innermost-type | string | No | If the OPC UA Field value attribute mapped is an array, this property specifies the OCF data type mapping the TypeName of the elements of the array. |

Figure 6.3 shows an example of the representation of a Value belonging to the MyType Structured DataType (shown by Figure 2.3). In the example the current values of the fields "var1", "var3" and "var4" are 10, 12 and "hello", respectively.



***Figure 6.3:*** *Example of Value of MyType DataType*

Figure 6.3 assumes that the current values of the fields "var1",

"var3" and "var4" are 10, 12 and "hello", respectively.  The figure points out that the representation of the Value attribute is contained into a property named "fields" shown on the left of the figure.  It is an array containing two elements, one for each of the two fields of the MyData Structured DataType (i.e. "var1" and "var2").  In particular, each field is represented as a JSON object containing the properties defined in Table 6.8.

According to the example considered, the field "var1" is mapped filling the "field-name" with the name of the field (i.e. "var1") and the "field-value" with its current value (i.e. 10).  Furthermore, the OCF data type used for the representation of current value of the OPC UA field is used to fill the third property named "value-type".  For this reason, in the case of the field "var1", the "value-type" contains "number".

Considering the field "var2", its representation is a little bit more complex as "var2" belongs to a Structured DataType (i.e. *VarType*, as shown by Figure 2.3).  For this reason, the JSON object representing the field "var2" contains the "field-name" property filled by the name of the field (i.e. "var2") and the "value-type" property set to "object".  Furthermore, the value contained in the "field-value" will be a JSON object made up by only one property, named "fields".  As seen before, this property is an array of JSON object, each of which represents a field of the Structured DataType defining the value.  Table 6.8 shows the properties of the two JSON objects relevant to the fields "var3" and "var4".

#### 6.1.6.4 Array

OPC UA Property and DataVariable Nodes can feature an array Value attribute. In this case, the "value-type" property specified in Tables 6.3 and 6.4 assumes the value "array".

The OCF data type of each element is obtained as explained in the previous subsections, and it is used to fill the content of the "innermost-type" property.

The "num-dimensions" property is present and it is filled with the array dimensions obtained by the *ValueRank* and *ArrayDimensions* attributes of the OPC UA Variable Node. This property is an array of integers where the length of the array maps the number of dimensions of the OPC UA Value attribute (i.e. ValueRank), and each element specify the length of the relevant dimension (i.e. ArrayDimensions).

Finally, the "opc-property-value" (in case of OPC UA Property Node) or the "OPCValue" (in case of OPC UA DataVariable Node) will contain the representation of the original OPC UA array; each element of this array is encoded according to the OCF data type found for the "innermost-type" property.

### 6.1.7 Case Study

This section provide an example of the mapping from OPC UA to OCF in order to improve the understanding of the proposal.

Figure 6.4 shows a subset of an OPC UA AddressSpace. It is made up by an OPC UA Object named *OrdersFolder* belonging to the FolderType ObjectType. This Node organises the DataVariable *Order1* (through an Organizes Reference) and features the OPC UA Property *MaxOrderNumber* (linked through a HasProperty Reference). For

each OPC UA Node, Figure 6.4 shows the attributes and the relevant values involved in the mapping example. It is possible to see that Order1 DataVariable Node features a Value belonging to the *OrderType* DataType, for which the NodeId is specified (i.e. ns=2 and i=17).

Figure 6.5 shows the description of the OrderType DataType assumed to be present in the AddressSpace; such description has been named *OrderDescription* (as shown by the Variable Node of DataTypeDescriptionType type in Figure 6.5). Analysing the content of the *MyDictionary* Variable in the figure, it is clear that this DataType is a Structured DataType made up by two fields named "OrderID" and "Client". The former is an integer, the latter a nested structure described by the Structured DataType *PersonType*.

Description of PersonType DataType is contained in the MyDictionary Variable. It is made up by two string fields named "Name" and "Surname". The structure of the OrderType DataType described so far clarifies the content of the Value attribute of the Order1 DataVariable shown by Figure 6.4. This attribute contains the two current value of the fields "OrderID" and "Client", i.e. 1234 for the first field and the couple of strings "John" and "Smith" for the other field.

The subset of OPC UA AddressSpace shown in Figure 6.4 is mapped into OCF ecosystem by an OCF Device of "x.opc.device" Device Type. As said, this OCF Device must include the mandatory Resource named "AddressSpaceSubset" of "x.opc.object" type, used to aggregate the OCF Resources mapping the OPC UA Nodes shown by Figure 6.4.

The mapping of the OrdersFolder Node is shown in Listing 6.1.

```
{
```

OPC UA Object Node

OrdersFolder::FolderType

Attributes
NodeId = (ns=2; i=12)
DisplayName = "OrdersFolder"
BrowseName = "OrdersFolder"
NodeClass = "Object"
Description = "A folder containing orders"

Organizes

OPC UA DataVariable Node

Order1::
DataVariableType

Attributes
NodeId = (ns=2; i=20)
DisplayName = "Order1"
BrowseName = "Order1"
NodeClass = "Variable"
Value = {
        OrderID: 1234,
        Client: {
                Name: "John",
                Surname: "Smith"
        }
}
DataType = OrderType (ns=2; i=17)

HasProperty

OPC UA Property Node

MaxOrderNumber::PropertyType

Attributes
NodeId = (ns=2; i=45)
DisplayName = "MaxOrderNumber"
BrowseName = "MaxOrderNumber"
NodeClass = "Variable"
Value = 10
DataType= Int32

**Figure 6.4:** *Subset of OPC UA AddressSpace to be mapped to OCF*

OrderType
(ns = 2; i = 17)

HasEncoding

DefaultBinary::
DataTypeEncodingType

HasDescription

OrderDescription::
DataTypeDescriptionType

Attribute:
DisplayName = "OrderDescription"
Value = {"OrderType"}
DataType = ByteString

HasComponent

MyDictionary::
DataTypeDictionaryType

Attribute:
DisplayName = "MyDictionary"
Value = {"
                .......
<opc:StructuredType Name="OrderType">
        <opc:Field Name="OrderID" TypeName="opc:Int32"/>
        <opc:Field Name="Client" TypeName="tns:PersonType"/>
</opc:StructuredType>
                .......

<opc:StructuredType Name="PersonType">
        <opc:Field Name="Name" TypeName="opc:String"/>
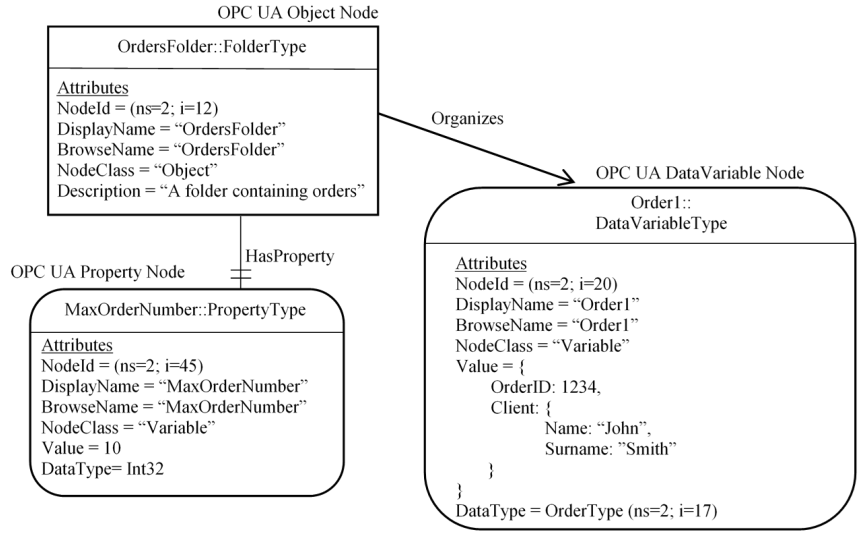        <opc:Field Name="Surname" TypeName="opc:String"/>
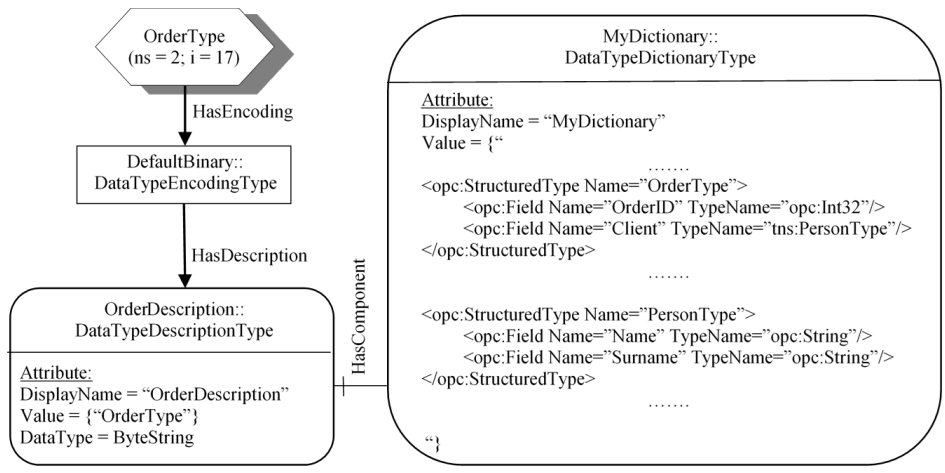</opc:StructuredType>
                .......

"}

**Figure 6.5:** *Description of OrderType DataType*

133

```
  "rt": "x.opc.object",
  "if": [...],
  "id": "2-12",
  "n": "OrdersFolder",
  "OPCNodeType": "folder",
  "OPCDescription": "A folder containing orders",
  "OPCProperties": [
    {
      "opc-property-name": "MaxOrderNumber",
      "opc-property-value": 10,
      "value-type": "number"
    }
  ],
  "links": [
    {
      "href": "/Order1",
      "rt": "x.opc.datavariable",
      "if": [...]
    }
  ]
}
```

**_Listing 6.1:_**    _JSON document representing the OPC UA OrdersFolder Node_

The mapping of OrdersFolder Node is realised using an OCF Resource belonging to the "x.opc.object" Resource Type.

Concerning the common OCF properties of the JSON document shown by Listing 6.1, the "rt" property is filled by the string "x.opc.object", the "id" property contains the string representation of the NodeId identifying the OPC UA OrdersFolder Node (i.e. the couple ns=2 and i=12 which becomes the string "2-12") and the "n"

134

property is set to the DisplayName of the same Node (i.e. "Orders-Folder"). It is worth noting that the "if" property is not filled because strictly related to OCF Services, as explained in Section 6.1.3.

Regarding the additional properties defined by the "x.opc.object" Resource Type and shown by Table 6.2, "OPCNodeType" is set to "folder" (as the OPC UA Node represented is of the FolderType ObjectType). The "OPCDescription" property is filled with the value contained in the OPC UA Node attribute Description (in this case it was assumed that this attribute contains the value "A folder containing orders", as shown by Figure 6.4).

The "OPCProperties" property in Listing 6.1 is an array containing JSON objects representing the OPC UA Properties relevant to the Node represented. Figure 6.4 shows that OrdersFolder Node features a single OPC UA Property named MaxOrderNumber. Hence, "OPCProperties" is an array containing a single JSON object representing the OPC UA MaxOrderNumber Property Node. This object is made up by the properties "opc-property-name", "opc-property-value" and "value-type" as defined in Table 6.3. The "opc-property-name" is filled with the value of the BrowseName attribute (i.e. "MaxOrderNumber"). The "opc-property-value" is filled by the representation of the Value attribute of the OPC UA Property Node according to the OCF data type relevant to the OPC UA DataType of the Value attribute. Figure 6.4 shows that this attribute is of Int32 Built-in DataType. Applying the conversion specified in Table 6.6, the relevant OCF data type is number. On account of what just said, the Int32 Value attribute of the OPC UA Property Node will be represented as a number. As consequence, "value-type" and "opc-property-value" properties are filled by "number" and 10, respectively.

Figure 6.4 shows that OrdersFolder is the source of an Organizes Reference targeting the OPC UA Order1 DataVariable. As explained in Section 6.1.3, each Organizes Reference of an OPC UA Folder Node is mapped as an OCF Link. For this reason, the "links" property of Listing 6.1 is an array with a single OCF Link representing the Organizes Reference shown in Figure 6.4. It specifies the URI of the pointed Resource (i.e. "/Order1") and its Resource Type (i.e. "x.opc.datavariable") through the "href" and "rt" properties, respectively.

The state of the OCF Resource representing the OPC UA Order1 DataVariable is shown by Listing 6.2.

```
{
  "rt": "x.opc.datavariable",
  "if": [...],
  "id": "2-20",
  "n": "Order1",
  "value-type": "object",
  "OPCValue": {
    "fields": [
      {
        "field-name": "OrderID",
        "field-value": 1234,
        "value-type": "number"
      },
      {
        "field-name": "Client",
        "field-value": {
          "fields": [
            {
              "field-name": "Name",
```

```
              "field-value": "John",
              "value-type": "string"
            },
            {
              "field-name": "Surname",
              "field-value": "Smith",
              "value-type": "string"
            }
          ]
        },
        "value-type": "object"
      }
    ]
  }
}
```

***Listing 6.2:*** *JSON document representing the OPC UA Order1 Node*

Concerning the common OCF properties of the JSON document
shown by Listing 6.2, "rt" is filled by the string "x.opc.datavariable",
"id" contains the string representation of the NodeId identifying the
OPC UA Order1 Node (i.e. "2-20"), and "n" contains the Display-
Name of the Node represented (i.e. "Order1").

Regarding     the     additional     properties     defined     by     the
"x.opc.datavariable" Resource Type and specified by Table 6.4,
the "value-type" specifies the OCF data type used to represent the
Value attribute of the OPC UA Order1 Node and "OPCValue" is
filled with this representation. In this example, the DataType of the
Value attribute is the Structured DataType named OrderType and
shown in Figure 6.5. Applying the conversion specified in Table 6.6,
the "value-type" property is set to "object".

Applying the mapping described in Section 6.1.6, "OPCValue" property is filled by a JSON object with a single property named "fields". This property is an array containing an object for each field present in the Structured DataType. For this reason, the property "fields" in Listing 6.2 is an array containing two objects representing the fields OrderID and Client shown by Figure 6.5. Each object is made up by the properties defined by Table 6.8.

Considering the object representing the field OrderID, the "field-name" is set to field name value, i.e. "OrderID". Applying the conversion rules defined in Table 6.6, the "value-type" property is filled with "number". The "field-value" contains the representation of the current value of the OPC UA Field according to the type specified by the "value-type", i.e. "number"; according to the current value shown by Figure 6.4, the "field-value" is set to 1234.

Considering the object representing the field Client (which is specified by another Structured DataType named PersonType), the "field-name" is set to "Client". The "value-type" property is filled with "object", as its "field-value" property contains an object created applying the rules just explained for "OPCValue". This object features the property "fields" which is an array containing two objects relevant to the fields "Name" and "Surname". Both objects feature the "value-type" property set to "string". Listing 6.2 shows that for the the JSON objects represeting the fields Name and Surname, the values of "value-type" property are set to "John" and "Smith", respectively.

# 6.2 Mapping from OCF Resource Model to OPC UA Information Model

This section presents the mapping from OCF Resource Model to OPC UA Information Model. The mapping proposed is able to represent the element of the OCF Resource Model in the OPC UA ecosystem.

## 6.2.1 Mapping idea

The mapping from OCF Resource Model to OPC UA Information Model has the aim to enable the representation of fundamental OCF elements in OPC UA ecosystems.

As said in Chapter 4, the OCF Resource Model is based on the concepts of OCF Device and OCF Resource. An OCF Device is used to represent real physical devices and their components which are represented as OCF Resources inside the OCF Device. In order to model these concepts and the relevant information, the OCF Resource Model defines several elements (i.e. OCF Device Type, OCF Resource Type, ect.).

Integration of physical entities like devices is present also in OPC UA. In fact, OPC foundation has defined an information model to enable the representation of devices and their integration in the OPC UA ecosystem. As said in Chapter 2, this information model is described in the specification *OPC UA for Devices - Companion Specification* [30] and is called *OPC UA Device Model*.

On the basis of what said and taking into account the goal of the mapping proposed, the OPC UA Device Model has been considered the starting point for the definition of a novel information model,

139

called *OCF OPC UA Information Model*. It is built on top of standard OPC UA Information Model and OPC UA Device Model as graphically represented in Figure 6.6.
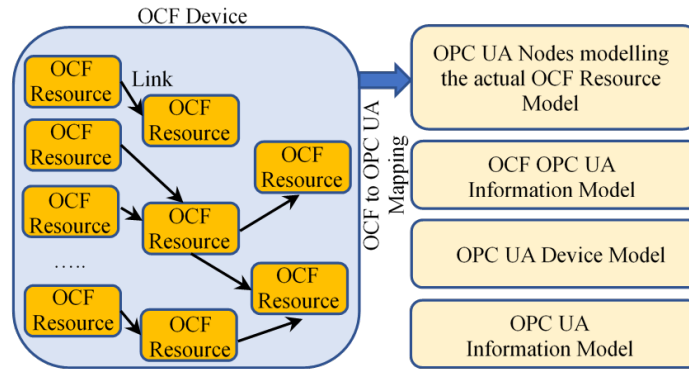


***Figure 6.6:*** *OCF OPC UA Information Model*

The proposed OCF OPC UA Information Model mainly offers several novel OPC UA ObjectTypes. Their definition was done in order to allow the mapping of fundamental elements of OCF Resource Model, i.e. Device Type, Resource Type, Resource, Collection. Most of the novel ObjectTypes inherit from the types of the OPC UA Device Model (i.e. TopologyElementType and DeviceType ObjectTypes) which in turn extends the types defined in the OPC UA Information Model.

The highest level represented in Figure 6.6, represents the set of OPC UA Nodes, instances of the novel ObjectTypes, used to map the actual OCF elements of the OCF Resource Model, e.g. the OCF Device and the OCF Resources there contained.

The novel ObjectTypes defined in the research work here presented, are: *OCFResourceType*, *OCFResourceInstanceType* and

*OCFDeviceType.* Before deepening the description of these Object-
Types, it is important to point out the main assumptions assumed in
this research.

First of all, it is worth noting that OCF Resource Model allows
multiple inheritance. In fact, an OCF Resource can be instance of one
or more OCF Resource Types. Each OCF Resource Type defines a
set of properties owned by the OCF Resource. Mapping of each of the
OCF Resource Types has been realised defining the abstract Object-
Type named *OCFResourceType*. A subtype of *OCFResourceType* is
used to define a new OPC UA ObjectType mapping an OCF Resource
Type. Each subtype of *OCFResourceType* features components made
up by OPC UA DataVariables, mapping all the properties of relevant
OCF Resource Type.

The obvious mapping of an OCF Resource in OPC UA ecosys-
tem should consist of defining an OPC UA ObjectType subtype of
*OCFResourceType* for each of its OCF Resource Type and create an
OPC UA Object instance of all these ObjectTypes. Although a so-
lution of this kind, graphically represented in Figure 6.7, seems to be
perfect to realise the mapping of an OCF Resource, in the OPC UA
ecosystem multiple inheritance is forbidden and a different approach
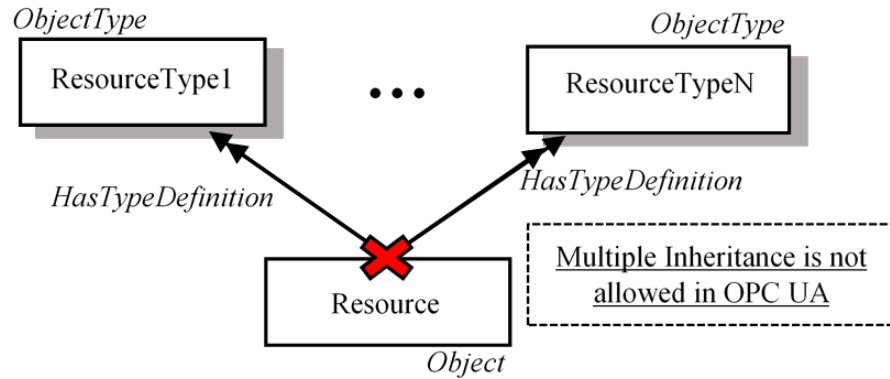must be researched.

***Figure 6.7:*** *Prohibition of multiple inheritance in OPC UA*

To overcome this problem, the solution adopted in this proposal is the definition of a particular OPC UA ObjectType, called *OCFResourceInstanceType*. For each OCF Resource, an instance of *OCFResourceInstanceType* is created; the goal is to use this instance to aggregate instances of the *OCFResourceType* subtypes modelling OCF Resource Types relevant to the OCF Resource. For this reason, it has been assumed that these instances contain the actual values of the properties relevant to the OCF Resource represented. In this way, the solution adopted allows to distribute the state of a Resource among these instances.

This aggregation is realised using the Configurable Component pattern defined in [30], as explained in the following. The instance of *OCFResourceInstanceType* created for each OCF Resource contains an OPC UA Object of ConfigurableObjectType ObjectType, named in this paper *Aspects*. In turn, Aspects contains an instance of each *OCFResourceType* subtypes modelling the OCF Resource Types rel-

142

evant to the OCF Resource. Finally, due to the features of the ConfigurableObjectType, Aspects owns a folder named SupportedTypes; it is used to organise the subtypes of *OCFResourceTypes*. In this way, the aggregation is able to overcome the lack of multiple inheritance.

In the following, a simple example will be presented to better understand the main assumptions just explained.

Consider an OCF Resource representing a real bulb. The state of this OCF Resource is shown by Listing 6.3.

```
/a/bulb
{
    "n": "bulb",
    "rt": ["oic.r.switch.binary","oic.r.light.brightness"],
    "if": [...],
    "value": true,
    "brightness": 80
}
```

***Listing 6.3:*** *OCF Resource representing a bulb*

As specified by the "rt" property, the OCF Resource is instance of two OCF Resource Types called "oic.r.switch.binary" and "oic.r.light.brightness". The former, described by Table 6.9, defines a boolean property named "value" indicating if the bulb is on or off. The latter, described by Table 6.10, defines an integer property named "brightness" indicating the brightness level. Both "value" and "brightness" property are part of the state of the OCF Resource shown by Listing 6.3.

**Table 6.9:** *OCF Properties defined by "oic.r.switch.binary" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| Common Properties | See "oic.core" Resource Type in Table 4.1 | | |
| value | boolean | Yes | Status of the switch. |

**Table 6.10:** *OCF Properties defined by "oic.r.light.brightness" Resource Type*

| Property Name | OCF data type | Mandatory | Description |
|---|---|---|---|
| Common Properties | See "oic.core" Resource Type in Table 4.1 | | |
| brightness | integer | Yes | Quantized representation in the range 0-100 of the current sensed or set value for Brightness. |

Based on what said above, Figure 6.8 shows the mapping of the OCF Resource of Listing 6.3.

As shown in Figure 6.8, the instance of *OCFResourceInstance-Type* created for the OCF Resource representing the bulb contains an OPC UA Object of ConfigurableObjectType ObjectType, named Aspects. In turn, Aspects contains an instance of each *OCFResourceType* modelling the OCF Resource Types relevant to the OCF
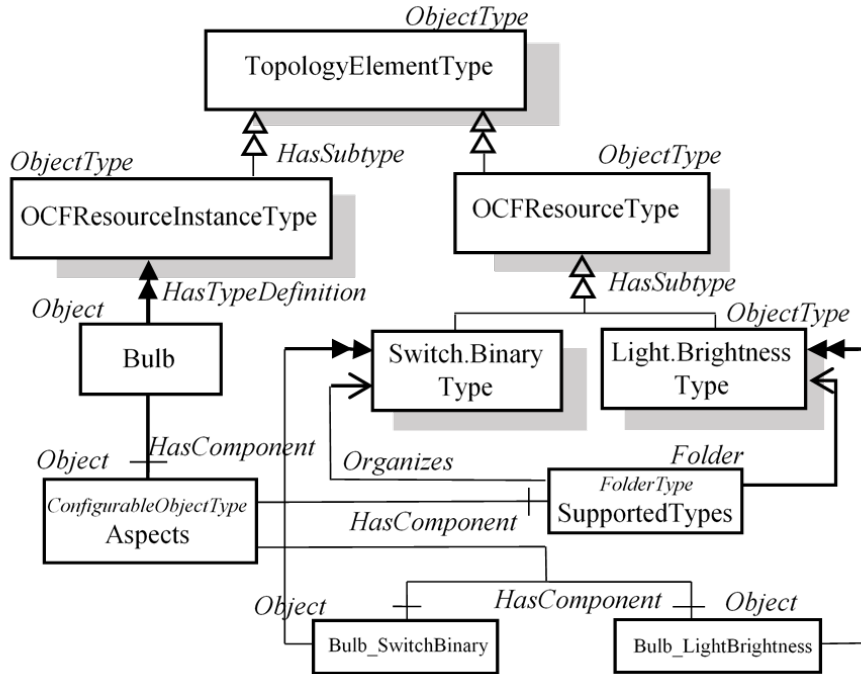
144

**Figure 6.8:** *Example of OCFResourceInstanceType and OCFResourceType*

Resource, i.e.  Switch.BinaryType and Light.BrightnessType representing "oic.r.switch.binary" and "oic.r.light.brightness", respectively. Furthermore, both Switch.BinaryType and Light.BrightnessType ObjectType are target of an Organizes Reference starting from SupportedTypes Node.

Figure 6.8 shows that both *OCFResourceInstanceType* and *OCFResourceType* has been defined as subtypes of the TopologyElementType ObjectType.  This choice will be better explained in the following, when the novel ObjectTypes defined in the proposal will be

145

great detailed.

It is worth noting that the solution just presented for the multiple inheritance problem is not the only one possible but it has been chosen as it is able to avoid loss of information in the mapping. In fact, through the OPC UA Folder SupportedTypes it is possible to expose the information about each OCF Resource Type defining the OCF Resource represented, included the properties and the relevant actual values. Furthermore, it is possible to easily retrieve information about the OCF Resource Types defining an OCF Resource: all the *OCFResourceTypes* supported by the *OCFResourceInstanceType* instance are reachable just following the Organizes References starting from the SupportedTypes Node instead of retrieving this information following the HasTypeDefinition starting from each instance of *OCFResourceType*.

The last novel ObjectType defined in this mapping is *OCFDeviceType*. It is a subtype of the OPC UA DeviceType ObjectType and its instances will be used to represent OCF Devices. As an OCF Device contains OCF Resources, the instances of *OCFResourceInstanceType* will be component of an *OCFDeviceType* instance.

In the following subsections, all the novel ObjectType just defined will be described in depth.

### 6.2.2   OCFResourceType ObjectType

This ObjectType has the aim to represent OCF Resource Types in OPC UA. It is abstract: this means that an ObjectType extending *OCFResourceType* shall be created for each OCF Resource Type (e.g., Switch.BinaryType and Light.BrightnessType in Figure 6.8).

*OCFResourceType* is a subtype of TopologyElementType Object-Type and inherits each component of this last type, among which *ParameterSet* Object. All the properties defined by an OCF Resource Type are mapped as OPC UA *Parameters* and grouped by the *ParameterSet* Object.

As for example, consider the OCF Resource Type "oic.r.light.brightness" defined in Table 6.10; it defines an integer property named "brightness". As shown by Figure 6.8, this OCF Resource Type is mapped in OPC UA using a subtype of the *OCFResourceType* ObjectType called Light.BrightenessType. Figure 6.9 shows in more details this ObjectType.
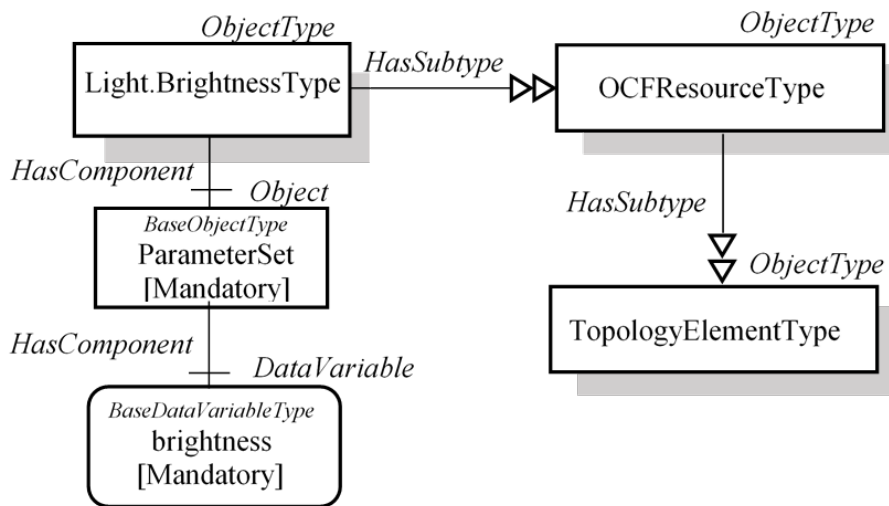


***Figure 6.9:*** *Light.BrightnessType ObjectType details*

As said before, Light.BrightnessType ObjectType is subtype of *OCFResourceType* ObjectType which in turn is subtype of TopologyElementType ObjectType: for this reason, it inherits the compo-

nent of this last one ObjectType. Among these component there is the Object named ParameterSet, shown by Figure 6.9. This Object is an InstanceDeclaration as it features a Mandatory ModellingRule Object (the notation [Mandatory] specifies that the Node is the source of a HasModellingRule Reference targeting a Mandatory ModellingRule Object as described in Section 2.1.6). This means that an instance of the Light.BrightnessType ObjectType must contains an HasComponent Reference targeting a ParameterSet Object. Furthermore, Figure 6.9 shows that the ParameterSet Object contains an OPC UA Parameter realised as an OPC UA DataVariable. The aim of this Parameter is to represent the "brightness" property defined by the "oic.r.light.brightness" OCF Resource Type. It is worth noting that this DataVariable is a InstanceDeclaration too (it features the ModellingRule Mandatory) and for this reason the ParameterSet Object contained in an instance of Light.BrightnessType ObjectType must contain this DataVariable Node. As said before, the Value attribute of this Node will contain the actual value of the "brightness" property: assuming the mapping of the OCF Resource representing a bulb shown by Listing 6.3, the Value attribute of this DataVariable Node will contain the value 80.

As said in Chapter 4, OCF specifications foresee several basic OCF Resource Types; some of them are: "oic.wk.d", "oic.wk.p" and "oic.wk.col". Three OPC UA ObjectTypes extending the *OCFResourceType* have been defined to represent them. They has been called DType, PType and ColType and will be used to map "oic.wk.d", "oic.wk.p" and "oic.wk.col" OCF Resource Types, respectively, as described in the following subsections.

### 6.2.3   OCFResourceInstanceType ObjectType

Instances of *OCFResourceInstanceType* ObjectType are used to map OCF Resources. It is a concrete ObjectType, subtype of TopologyElementType ObjectType and defined as shown by Figure 6.10.
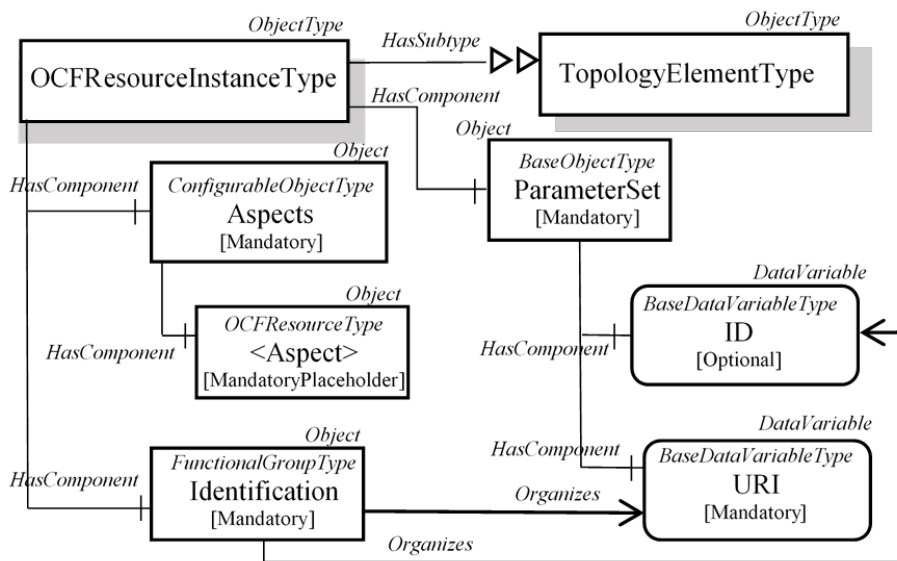


**Figure 6.10:** *OCFResourceInstanceType ObjectType*

An instance of this ObjectType is made up by several components. One of them is the *Aspects* Object. It is a ConfigurableObject aggregating instances of *OCFResourceType* subtypes, one for each OCF Resource Type defining the OCF Resource represented. As said in the previous subsection, each of these instances will contain *Parameters* representing the properties of the OCF Resource.

Another component is ParameterSet Object, inherited from TopologyElementType. All the Parameters of every component of Aspects

will be grouped by the ParameterSet of the instance of *OCFResource-InstanceType*. This grouping, not shown in Figure 6.10 due to the lack of space, allows to provide all the OCF properties composing the OCF Resource as Parameter of the Node representing the OCF Resource itself, resolving the problem of the prohibition of multiple inheritance in OPC UA.

It is worth noting that with the solution proposed is possible to obtain entire set of Parameters (i.e. reaching the Parameter through the ParameterSet Object of the instance of *OCFResourceInstanceType*) or to filter the Parameter by the *OCFResourceType* subtype (i.e. reaching the Parameter through the Aspects Object).

ParameterSet groups also other Parameters, among which Figure 6.10 shows URI (that is mandatory and it is used to map the URI of the OCF Resource represented) and ID (that is optional and it is used to map the "id" common property of the OCF Resource state). Since URI and ID identify the OCF Resource, they shall be grouped by the FunctionalGroup called *Identification* as explained in Section 2.4.1.1.

### 6.2.4 OCFDeviceType ObjectType

*OCFDeviceType* ObjectType is an abstract ObjectType subtype of OPC UA *DeviceType*. A subtype of *OCFDeviceType* ObjectType shall be created for each OCF Device Type; an instance of such subtype maps an OCF Device and the information it gathers. *OCFDeviceType* is graphically described in Figure 6.11.

As explained in Chapter 4, an OCF Device must expose OCF Resources and, eventually, subdevices. It has been assumed that map-
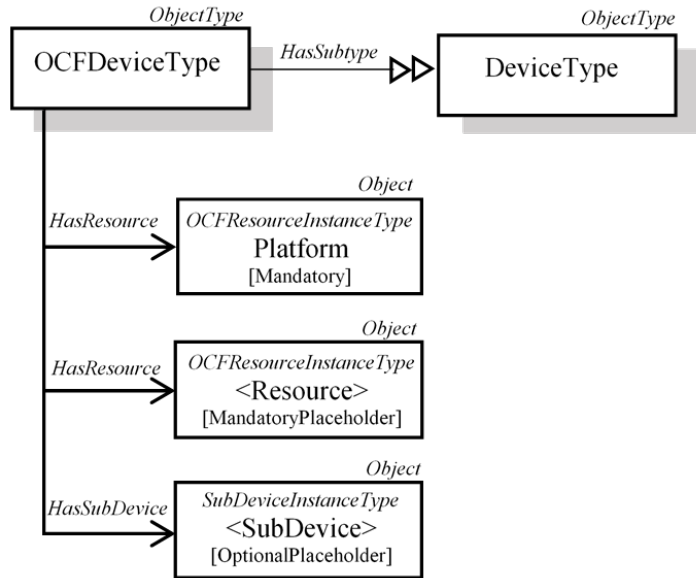
**Figure 6.11:** *OCFDeviceType ObjectType*

ping of the relationships between an OCF Device and its Resources
is achieved through the use of an ad-hoc defined OPC UA Reference-
Type, named *HasResource*. The other relationship with the subdevices
is modelled by another ad-hoc defined reference called *HasSubDevices*.
HasSubDevice is subtype of HasResource which in turn is subtype of
HasComponent ReferenceType. The definition of these Reference-
Types is shown in Figure 6.12.

HasResource and HasSubDevice References requires an instance
of *OCFDeviceType* subtype as source, modelling the OCF Device as
said. An HasResource Reference targets an instance of *OCFResource-
InstanceType* ObjectType modelling an OCF Resource owned by the
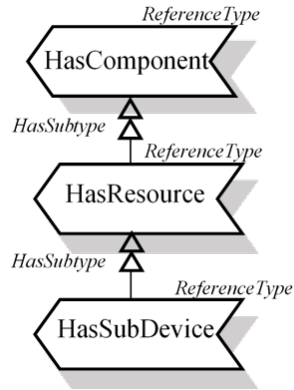OCF Device whilst HasSubDevice targets the instances of an ad-hoc

151

**_Figure 6.12:_** _HasResource and HasSubdevice ReferenceTypes_

subtype of _OCFResourceInstanceType_ ObjectType named SubDevice-InstanceType and modelling a subdevices of the OCF Device.

Among the Resources exposed by an OCF Device, the three ones addressed by the URIs "/oic/p", "/oic/res" and "/oic/d" are mandatory.

The OCF Resource addressed by "/oic/p" is mapped as an instance of _OCFResourceInstanceType_, named _Platform_ in Figure 6.11. For this reason, Platform Node features an Aspects Object as a component. In order to map the properties of the OCF Resource addressed by "/oic/p" (which are defined by "oic.wk.p" Resource Type) an instance of the _PType_ ObjectType is used. It is defined as a component of Aspects and exposes a Parameter for each property of the OCF Resource (i.e. each property defined by the ResourceType "oic.wk.p").

The OCF Resource addressed by "/oic/res" provides the list of OCF Links pointing the OCF Resources exposed by an OCF Device. It has been assumed to avoid the use of an OPC UA Node to repre-

sent the OCF Resource addressed by "/oic/res" and to map only the OCF Resources linked. These Resources are represented as instances of *OCFResourceInstanceType* and are targeted by HasResource References starting from the instance of the relevant *OCFDeviceType* subtype. In Figure 6.11, they are represented by the InstanceDeclaration named <Resource>.

As said, the OCF Resource addressed by the URI "/oic/d" is used to provide information about the relevant OCF Device through its properties (defined by "oic.wk.d" Resource Type). It has been assumed to avoid the mapping of the Resource by means of an OPC UA Node. Instead, the properties of this OCF Resource are mapped by OPC UA Properties (inherited by OPC UA DeviceType) and OPC UA Node Attributes of the instance of an *OCFDeviceType* subtype as specified by Table 6.11.

**Table 6.11:** *Mapping rules for OCF Resource addressed by "/oic/d"*

| "/oic/d" properties defined by "oic.wk.d" | OPC UA Property of OCFDeviceType | OPC UA Attribute of OCFDeviceType |
|---|---|---|
| Name (n) | - | DisplayName |
| Localized Description (ld) | - | Description |
| Software Version (sv) | SoftwareRevision | - |
| Manufacturer Name (dmn) | Manufacturer | - |
| Model Number (dmno) | Model | - |

As said in Chapter 4, subdevices must be represented as OCF Resources of the relevant Device. The properties of the Resource representing a subdevice are defined by the "oic.wk.d" Resource Type. Alongside this Resource Type, a Device Type is defined. Furthermore, if the Resource representing a subdevice is also a Collection (i.e. it is also defined by the "oic.wk.col" Resource Type), it shall link mandatory Resource specified by the Device Type.

In order to map an OCF Resource representing a subdevice, a subtype of *OCFResourceInstanceType*, named *SubDeviceInstanceType*, has been defined and it is graphically represented in Figure 6.13.
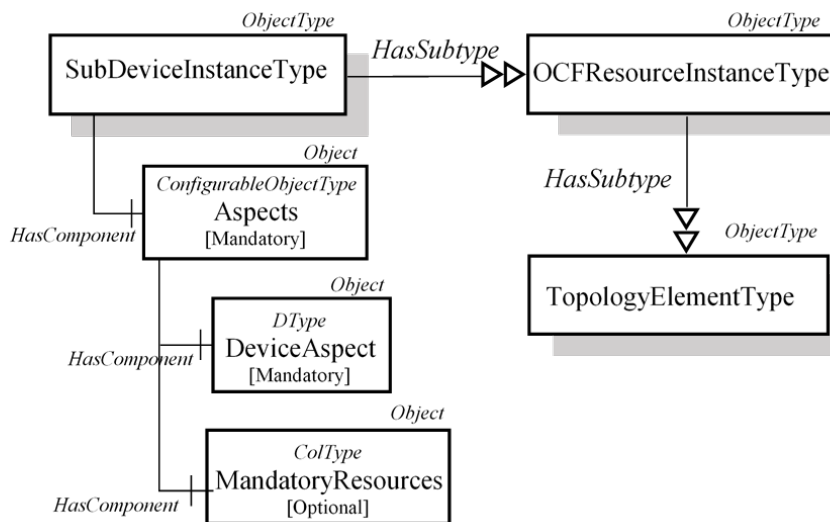


**Figure 6.13:** *SubdeviceInstanceType ObjectType*

Since *SubDeviceInstanceType* extends *OCFResourceInstanceType*, its instances expose an Aspects Object. In order to map the properties of the subdevice (which are defined by "oic.wk.d" Resource Type) an

instance of *DType* ObjectType is used. It is defined as a component of Aspects, named *DeviceAspect*, and exposes a Parameter for each property defined by "oic.wk.d".

If the subdevice is a Collection (i.e., "rt" property contains "oic.wk.col"), the instance of SubDeviceInstanceType will expose an instance of *ColType* ObjectType named *MandatoryResources* as component of Aspects. This Object is used to group the Resources of the Collection.

ColType is shown in Figure 6.14. It is a subtype of *OCFResource-Type* and has been defined to represent "oic.wk.col" Resource Type.
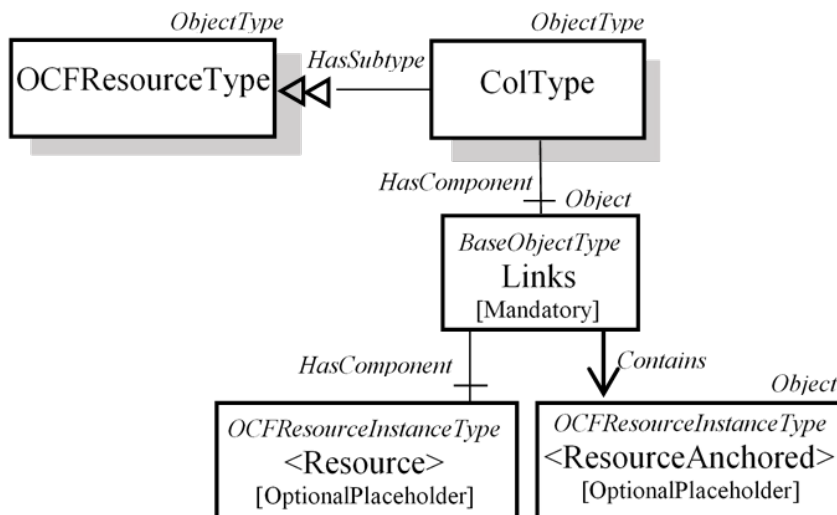


*Figure 6.14:* *ColType ObjectType*

Each instance of this type contains an Object named Links aggregating all the *OCFResourceInstanceType* instances representing OCF Resources belonging linked by the Collection. The aggregation is re-

alised through OPC UA References. Two kind of Resources can be aggregated.

The first kind are the Resources target of a Link without the "anchor" parameter set. In this case, the Link indicates that the OCF Collection owns directly the OCF Resource linked: for this reason, it has been assumed to use the HasComponent Reference which in OPC UA express a relationship of owning. In Figure 6.14 these Resources are specified by the InstanceDeclaration <Resource>.

The second kind are the Resources target of a Link with the "anchor" parameter set. In this case, the Link indicates that the OCF Resource is a part of the OCF Collection but without the relationship of owning: for this reason, it has been assumed to use the Aggregates Reference which in OPC UA express a similar semantics. As Aggregates is an abstract ReferenceType, Contains ReferenceType has been defined in order to use it for the reason just explained. In Figure 6.14 these Resources are specified by the InstanceDeclaration <ResourceAnchored>. Figure 6.15 shows the definition of Contains ReferenceType.
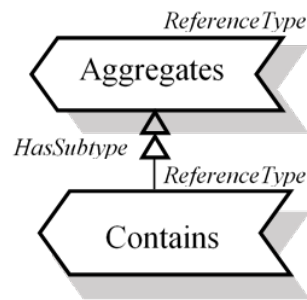


**Figure 6.15:** *Contains ReferenceType*

As an example consider a Resource representing a floor containing OCF Links pointing to the Resources representing the rooms in the floor. If each room contain lights, these may be defined in the Resource representing the floor as OCF Links having the "anchor" parameter set to the URI of the Resources representing the rooms containing the lights. In this case, the instance of *OCFResourceInstanceType* mapping the OCF Resource representing the floor contains a component of Aspects relevant to the ColType ObjectType; the OPC UA Nodes mapping the Resources representing the rooms will be target of Has-Component References starting from OPC UA Links Object whilst the OPC UA Nodes mapping the Resources representing the light will be target of Contains References starting from the OPC UA Links Object too.

## 6.2.5   CaseStudy

The aim of this Section is to provide an example of the mapping from OCF to OPC UA. The example is based on the mapping of an OCF Device belonging to the OCF Device Type "x.customdevice" described by Table 6.12.

***Table 6.12:*** *OCF CustomDevice Device Type*

| Device Name | Device Type | Required Resource Name | Required Resource Type |
|---|---|---|---|
| CustomDevice | "x.customdevice" | Bulb | ["oic.r.switch.binary", "oic.r.light.brightness"] |

Table 6.12 points out that each OCF Device instance of the

CustomDevice Device Type must expose an OCF Resource named Bulb belonging to the Resource Types "oic.r.switch.binary" and "oic.r.light.brightness", defined by Tables 6.9 and 6.10, respectively. The OCF Device is graphically shown by Figure 6.16.
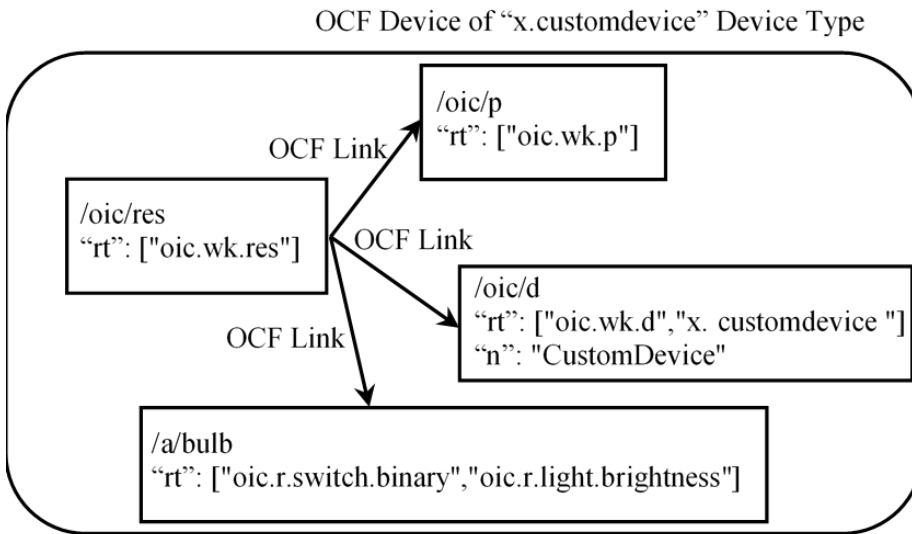


**Figure 6.16:** *OCF Device belonging to the CustomDevice Device Type*

As shown by Figure 6.16., the OCF Device contains the OCF Core Resources addressed by "/oic/d", "/oic/res" and "oic/p". Furthermore, the OCF Device contains the OCF Resource mandated by the OCF Device Type: it has been assumed that this Resource is the one specified by Listing 6.3.

Figure 6.17 shows the OPC UA representation of the OCF Device of Figure 6.16.

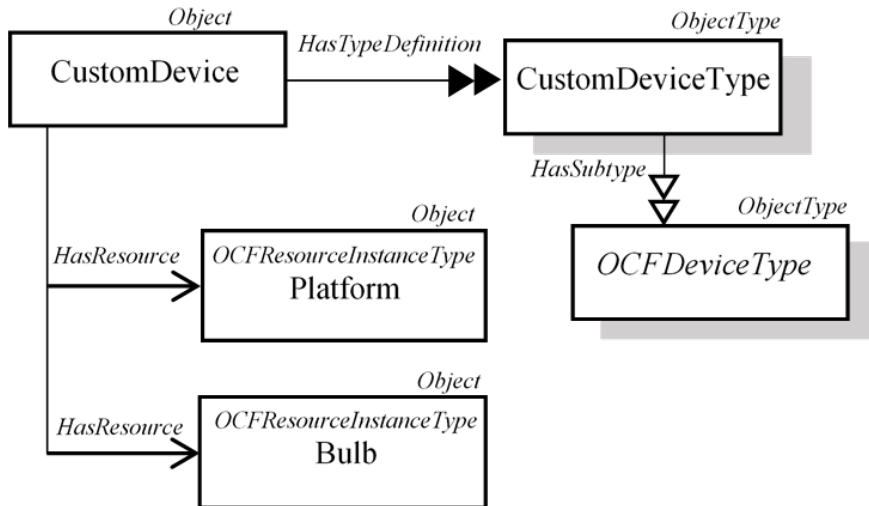In order to map the CustomDevice DeviceType, a new subtype of

**Figure 6.17:** *Mapping of the OCF Device of Figure 6.16*

*OCFDeviceType* ObjectType, named CustomDeviceType, has been defined. The OCF Device of Figure 6.16 is mapped as an instance of CustomDeviceType ObjectType named CustomDevice. As shown by the figure, this Object feature a DisplayName set to the name property of the OCF Device represented. It has been assumed that the optional properties localized description, software version, manufacturer name and model number of the OCF Resource addressed by "/oic/d" are not present: for this reason their mapping is not present in the Figure.

The CustomDevice Object features two HasResource References: the first points to the Platform Object, representing the OCF Resource addressed by "/oic/p". The second HasResource points to the representation of the OCF Resource of Listing 6.3. For space reason mapping of the instances of *OCFResourceInstanceType* owned by the

CustomDevice Object is not provide.

However, the mapping relevant to the OCF Resource of Listing 6.3 will be deepened: it is shown by Figure 6.18.
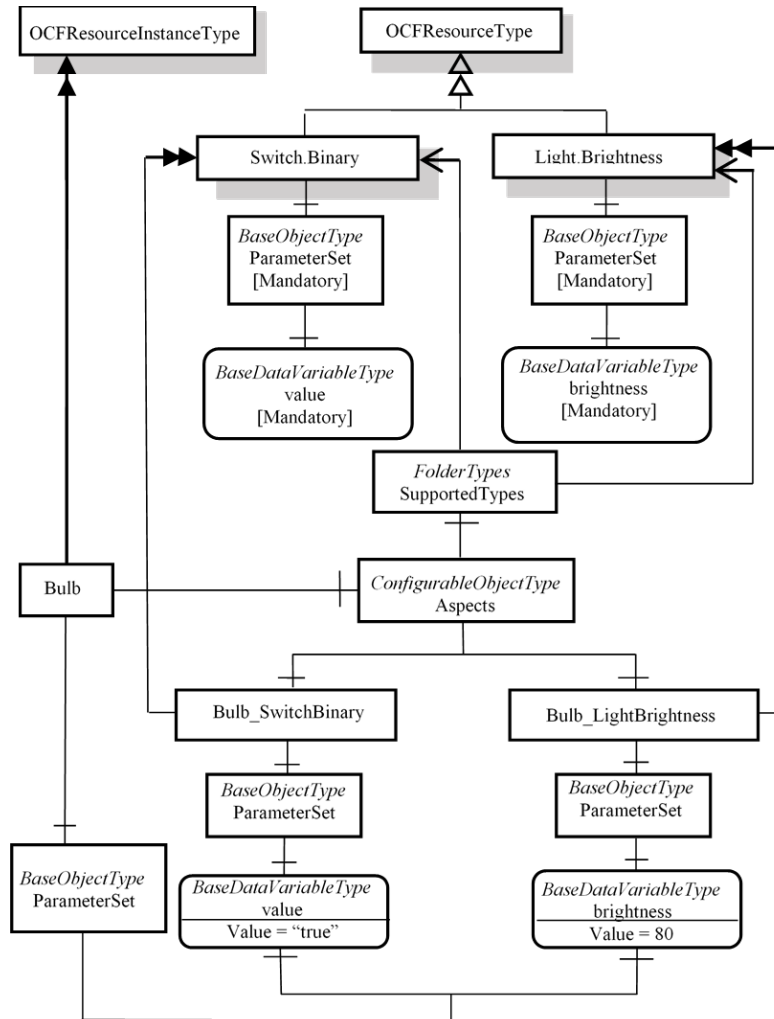


**Figure 6.18:** *Mapping of the OCF Resource of Listing 6.3*

The OPC UA ObjectTypes (Switch.BinaryType and Light.BrightnessType) model the two OCF Resource Type mentioned above as subtypes of *OCFResourceType* ObjectType. For each of the two subtypes, the figure shows the ParameterSet components and the relevant properties (value and brightness, respectively) defined as InstanceDeclarations.

The OCF Resource is mapped by an OPC UA Object of *OCFResourceInstanceType*. This Object has two components: Aspects and ParameterSets. The former is used to collect both the subtypes of *OCFResourceTypes* through the SupportedTypes Folder and the instances of these two subtypes. These instances are named Bulb_SwitchBinary and Bulb_LightBrightness of types Switch.BinaryType and Light.BrightnessType, respectively. For each instance, the actual values of the properties are shown (i.e. value and brightness). These values are directly accessible from Bulb Object through the ParameterSet component of the same Object or from each component of Aspects.

## 6.3   Contribution of the proposal inside the OCF standardisation activity

OCF specifies a particular Device, named *OCF Bridge*, with the aim of representing devices that exist on the network but that communicate using bridged protocol rather than OCF Protocol [59]. The overall goals of the OCF Bridge are to make Bridged Servers appear to OCF Clients as if they were native OCF servers, and to make OCF Servers appear to Bridged Clients as if they were native non-OCF servers.

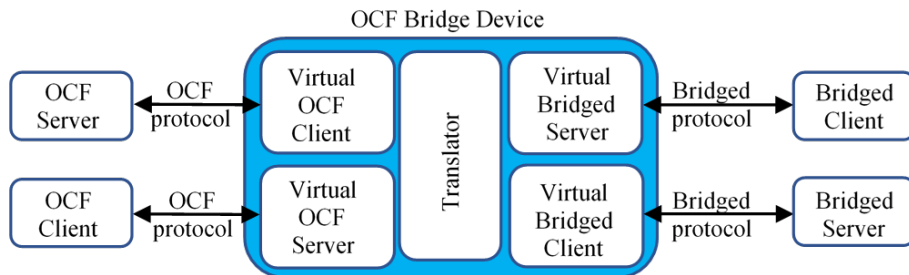Figure 6.19 shows the main components foreseen for the OCF Bridge Device [59].



**Figure 6.19:** *OCF Bridge Device Component*

*Translator* is responsible for translating to or from a specific Bridged Protocol; more than one translator can exist on the same OCF Bridge Device, for different Bridged Protocols.

*Virtual Bridged Client* is the logical entity that accesses data via a Bridged Protocol, representing the OCF Client on the other side of the OCF Bridge. Likewise, *Virtual Bridget Server* is the logical representation of an OCF Server, which an OCF Bridge Device exposes to Bridged Clients.

*Virtual OCF Client* and *Virtual OCF Server* are the logical representations of a Bridged Client (which an OCF Bridge Device exposes to OCF Servers) and a Bridged Server (which an OCF Bridge Device exposes to OCF Clients), respectively.

When translating between a Bridged and OCF protocols, [59] foresees for two possible types of translation named *deep translation* and *on-the-fly translation*.

In *deep translation*, information models used with the Bridged Pro-

tocol are mapped to the equivalent OCF Resource Types and vice-versa, in such a way that a compliant OCF Client or Bridged Client would be able to interact with the service without realising that a translation was made. Realisation of such translation requires dedicated effort and study of the data schema used on both sides to store information. The burden of this translation is on the Translator which is expected to dedicate most of its logic to realise the deep translation.

On-the-fly translation does not require any prior knowledge of the specific schema on the part of the translator. Instead, the burden is on one of the other participants in the communication, usually the client application.

It is very important to point out that, at the moment, OCF does not provide for an OCF Bridge Device realising translation between OPC UA and OCF for both deep and on-the-fly translations.

According to what said until now about the Bridge Device, the proposed mapping may be used for the Translator, in order to realise the deep translation between OPC UA and OCF.

# SEVEN

# CONCLUSION

The objective of this thesis is to investigate about the enhancement of interoperability in Industry 4.0, IoT and IIoT and to propose some improvements based on OPC UA. In particular two proposal are described.

The first proposal, called *Integration between OPC UA and the Web* and described in Chapter 5, is a proposal of interoperability opening OPC UA to the Web. The proposal is based on the definition of a novel data model mapping the OPC UA Information Model and based on common web data-formats (e.g. JSON). This data model has been used in the implementation of a RESTful web platform called *OPC UA Web Platform*, able to offer access to OPC UA Servers through the Web in a resource-oriented manner. The main feature of the proposed platform is its capability to limit the number of messages exchanged between user and platform and to simplify the view of the OPC UA information model from the user's perspective, requiring the user to have

a very limited knowledge of basic concepts detailed in the thesis. Platform may be accessed by a common application which is constrained neither to be compliant to OPC UA specification nor to implement OPC UA communication stack. The entire set of information maintained by an OPC UA Server continues to be accessible in the web by a common user; but it can be retrieved in a more user-friendly way limiting the data exchange and the number of accesses to the information model. Furthermore, it is able to update values of OPC UA Variables without being compliant to the OPC UA protocol. This is the key point that make the difference between the proposal here presented and the other REST-based solutions available in literature, as pointed out in Chapter 5.

The second proposal, called *Integration between OPC UA and OCF* and described in Chapter 6, is a proposal of interoperability based on the integration of OPC UA and IoT/IIoT World through a bidirectional mapping between OPC UA Information Model and OCF Resource Model. The mapping from OPC UA to OCF enables the mapping of each OPC UA Node and Reference of the AddressSpace into an element belonging to the OCF Resource Model. The mapping from OCF to OPC UA enables the mapping of each OCF fundamental element (i.e. OCF Device, OCF Resource, ect.) in OPC UA using an ad-hoc defined information model called OCF OPC UA Device Model. The main advantage of the proposal is the capability to enhance the interoperability of OPC UA in the IoT domain. In fact, each information maintained by an OPC UA Server may be used to populate an OCF device acting as server, and each information provided by an OCF Device may be used to populate the AddressSpace of an OPC UA Server. In this way is possible to enable publication of informa-

166

tion between OPC UA and OCF-compliant devices. The proposal is original as the issue has not been treated until now, due to the very recent definition of the OCF specifications. Furthermore, OCF specifications define a particular Device, named Bridge, with the aim of representing devices that exist on the network but that communicate using bridged protocol rather than OCF Protocol. According to the Bridging specification, a main component of the Bridge Device is the Translator. Translator is expected to dedicate most of its logic to a so-called "deep translation" types of communication, in which information models used with the Bridged Protocol are mapped to the equivalent OCF Resource Types and vice versa, in such a way that a compliant OCF Client or Bridged Client would not realise that a translation was made. At this moment, deep translation between OPC UA and OCF does not exist. The proposed mapping may be used as a component of the Translator to realise a part of the deep translation, relevant to the mapping of the information models between OPC UA and OCF.

# BIBLIOGRAPHY

[1] T. Guarda, M. Leon, M. F. Augusto, L. Haz, M. de la Cruz, W. Orozco, and J. Alvarez, "Internet of Things challenges," in *12th Iberian Conference on Information Systems and Technologies (CISTI)*, IEEE, 2017.

[2] Y. Liao, F. Deschamps, E. de Freitas Rocha Loures, and L. F. P. Ramos, "Past, present and future of Industry 4.0 - a systematic literature review and research agenda proposal," *International Journal of Production Research*, vol. 55, no. 12, pp. 3609–3629, 2017.

[3] L. D. Xu, E. L. Xu, and L. Li, "Industry 4.0: state of the art and future trends," *International Journal of Production Research*, vol. 56, no. 8, pp. 2941–2962, 2018.

[4] S. Jeschke, C. Brecher, H. Song, and D. B. Rawat, *Industrial Internet of Things.* Springer, 2017.

[5] S. Weyer, M. Schmitt, M. Ohmer, and D. Gorecky, "Towards industry 4.0 - standardization as the crucial challenge

for highly modular, multivendor production systems," *IFAC-PapersOnLine*, vol. 48, no. 2, pp. 579–584, 2015.

[6] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, p. 1234–1249, 2013.

[7] OPC Foundation, *OPC Specification - Part 1: Overview and Concepts*, 2017. Version 1.04.

[8] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Springer Verlag, 2009.

[9] P. Adolphs, H. Bedenbender, and D. Dirzus, "Reference Architecture Model Industrie 4.0 (RAMI4.0)," tech. rep., VDI – The Association of German Engineers and ZVEI – German Electrical and Electronic Manufacturers' Association, 2015.

[10] S.-W. Lin, B. Miller, J. Durand, G. Bleakley, A. Chigani, R. Martin, B. Murphy, and M. Crawford, "The Industrial Internet of Things Volume G1: Reference Architecture (IIRA v1.8)," tech. rep., Industrial Internet Consortium (IIC), 2017.

[11] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, Dept. Information and Computer Science, University of California, USA, 2000.

[12] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, 2007.

[13] S. Grüner, J. Pfrommer, and F. Palm, "RESTful Industrial Communication With OPC UA," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1832–1841, 2016.

[14] "Prosys OPC UA Web Client." `https://www.prosysopc.com/blog/prosys-opc-ua-web-client-released`.

[15] T. Paronen, "A web-based monitoring system for the Industrial Internet," Master's thesis, School of Science, Aalto University, Finland, 2015.

[16] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST—A protocol for integrating sens. networks into the Internet," in *Workshop Real-World Wireless Sens. Netw. (REALWSN)*, 2005.

[17] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *2010 Internet of Things (IOT)*, 2010.

[18] "Open Connectivity Foundation (OCF)." `https://openconnectivity.org`.

[19] ETSI, "Cross Fertilisation through Alignment, Synchronisation and Exchanges for IoT. Strategy and coordination plan for IoT interoperability and standard approaches." `https://european-iot-pilots.eu/wp-content/uploads/2017/10/D06_01_WP06_H2020_CREATE-IoT_Final.pdf`, 2017.

[20] M. J. A. G. Izaguirre, A. Lobov, and J. L. M. Lastra, "OPC-UA and DPWS interoperability for factory floor monitoring using

complex event processing," in *9th IEEE International Conference on Industrial Informatics*, 2011.

[21] Object Management Group (OMG), "OPC UA/DDS Gateway." `https://www.omg.org/spec/DDS-OPCUA/`.

[22] OPC Foundation, *OPC Specification - Part 14: PubSub*, 2018. Version 1.04.

[23] R.Baldoni, M. Contenti, and A. Virgillito, *Future Directions of Distributed Computing*, vol. 2584, ch. The Evolution of Publish/-Subscribe Communication Systems, pp. 137–141. Springer, 2003.

[24] OPC Foundation, *OPC Specification - Part 3: Address Space Model*, 2017. Version 1.04.

[25] OPC Foundation, *OPC Specification - Part 5: Information Model*, 2017. Version 1.04.

[26] OPC Foundation, *OPC Specification - Part 4: Services*, 2017. Version 1.04.

[27] OPC Foundation, *OPC Specification - Part 6: Mappings*, 2017. Version 1.04.

[28] OPC Foundation, *OPC Specification - Part 6: Data Access*, 2017. Version 1.04.

[29] OPC Foundation, *OPC Specification - Part 2: Security Model*, 2015. Version 1.03.

[30] OPC Foundation, *OPC Unified Architecture for Devices Companion Specification*, 2013. Version 1.01.

[31] IETF, "The JavaScript Object Notation (JSON) Data Interchange Format." `https://tools.ietf.org/html/rfc8259`, 2017.

[32] L. Bassett, *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. O'Reilly Media, 2015.

[33] IETF, "JSON Schema: A Media Type for Describing JSON Documents." `https://tools.ietf.org/html/draft-handrews-json-schema-01`, 2018.

[34] "JSON Schema." `http://json-schema.org`.

[35] M. Droettboom, "Understanding JSON Schema." `http://json-schema.org/understanding-json-schema/index.html`, 2018.

[36] "Open Connectivity Foundation Specification." `https://openconnectivity.org/developer/specifications`.

[37] Open Connectivity Foundation, *OCF Core Specification*, 2018. Version 1.3.1.

[38] S. Cavalieri, D. D. Stefano, M. G. Salafia, and M. S. Scroppo, "Integration of OPC UA into a Web-based Platform to enhance interoperability," in *26th IEEE International Symposium on Industrial Electronics (ISIE 2017)*, 2017.

[39] S. Cavalieri, D. D. Stefano, M. G. Salafia, and M. S. Scroppo, "A Web-based Platform for OPC UA integration in IIoT environment," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2017)*, 2017.

[40] S. Cavalieri, D. D. Stefano, M. G. Salafia, and M. S. Scroppo, "OPC UA integration into the Web," in *43rd Annual Conference of the IEEE Industrial Electronics Society (IECON 2017)*, 2017.

[41] S. Cavalieri, M. G. Salafia, and M. S. Scroppo, "Integrating OPC UA with web technologies to enhance interoperability," *Computer Standards & Interfaces*, 2018.

[42] "OPC UA Web Platform." `https://github.com/OPCUAUniCT/OPCUAWebPlatformUniCT`, 2017.

[43] IETF, "HTTP Over TLS." `https://tools.ietf.org/html/rfc2818`, 2000.

[44] K. Nayyeri and D. White, *Pro ASP.NET SignalR -Real-Time Communication in .NET with SignalR 2.1*. Apress, 2014.

[45] Andrew Banks and Rahul Gupta, *Message Queue Telemetry Transport (MQTT) - OASIS Standard*, 2014. Version 3.1.1.

[46] M. Collina, "MOSCA MQTT." `https://github.com/mcollina/mosca`.

[47] S. Tilkov and S. Vinoski, "Node.js: JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[48] OPC Foundation, *OPC Specification - Part 7: Profiles*, 2017. Version 1.04.

[49] M. Masse, *REST API Design Rulebook. Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2011.

[50] C. Nagel, *Professional C# 7 and .NET Core 2.0*. Wrox, 2018.

[51] Microsoft, ".NetCore Source Code." `https://github.com/dotnet/core`.

[52] O. Foundation, "OPC UA .NetCore Stack." `https://github.com/OPCFoundation/UA-.NETStandardLibrary`.

[53] IETF, "JSON Web Token (JWT)." `https://tools.ietf.org/html/rfc7519`, 2015.

[54] S. Cavalieri and M. S. Scroppo, "A proposal to make OCF and OPC UA interoperable," in *19th IEEE International Conference on Industrial Technology (ICIT 2018)*, 2018.

[55] S. Cavalieri, M. G. Salafia, and M. S. Scroppo, "Mapping OPC UA AddressSpace to OCF resource model," in *1st IEEE Industrial Cyber-Physical Systems (ICPS 2018)*, 2018.

[56] S. Cavalieri, M. G. Salafia, and M. S. Scroppo, "Towards interoperability between OPC UA and OCF," *Journal of Industrial Information Integration*, 2018. Pending Review.

[57] S. Cavalieri, M. G. Salafia, and M. S. Scroppo, "Interoperability between OPC UA and OCF," in *20th IEEE International Conference on Industrial Technology (ICIT 2019)*, 2018. Pending Review.

[58] S. Cavalieri, M. G. Salafia, and M. S. Scroppo, "Realising Interoperability between OPC UA and OCF," *IEEEAccess - The Multidisciplinary Open Access Journal*, 2018. Pending Publication.

[59] Open Connectivity Foundation, *OCF Bridging Specification*, 2017. Version 1.3.0.