FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interoperability at the Management Level of Building Automation Systems

An OPC UA Information Model for BACnet

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Andreas Fernbach
Matrikelnummer 0325790

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Wolfgang Kastner

Wien, 27.01.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interoperability at the Management Level of Building Automation Systems

An OPC UA Information Model for BACnet

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Andreas Fernbach
Registration Number 0325790

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o. Univ. Prof. Dr. Wolfgang Kastner

Vienna, 27.01.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andreas Fernbach
Aloys Wachstraße 2, 4650 Lambach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Mein Dank gilt zunächst einem langjährigen Mitarbeiter der Automation Systems Group, Wolfgang Granzer, der vertrauensvoll mit dem Angebot an mich herangetreten ist, innerhalb dieser Arbeitsgruppe eine Diplomarbeit zu verfassen. Durch seine kompetente fachliche Unterstützung ist es mir schließlich gelungen, in einem Thema, das für mich Neuland war, Fuß zu fassen und dieses inhaltlich zu einem guten Abschluss zu bringen.

Unserem sehr geschätzten Arbeitsbereichsleiter Wolfgang Kastner habe ich es zu verdanken, dass ich diese Arbeit im Zuge einer Anstellung am Institut für Rechnergestützte Automation verfassen durfte. So essentiell finanzielle Angelegenheiten auch sind, viel ausschlaggebender für meine Zukunft waren die vielen mir durch Wolfgang gebotenen Gelegenheiten, im Zuge von Konferenzbeiträgen meine Ideen der Community zu präsentieren. Dies zeugt von besonderem Vertrauen, das meine größte Wertschätzung genießt.

Meinen Eltern danke ich ganz speziell für die mir ermöglichte schulische und universitäre Ausbildung und für die in diesem Zuge notwendige Unterstützung. Eure Geduld habe ich wohl des Öfteren bis an ihre Grenzen strapaziert, was eurer Zuversicht hinsichtlich eines erfolgreichen Abschlusses dieser Lebensabschnitte aber trotzdem keinen Abbruch tat.

Zuletzt seien noch alle meine Freunde und Kollegen erwähnt, die mich auf meinem Weg durchs Studium begleitet haben. Besonders hervorheben möchte ich aus diesem Kreis meine liebe Freundin Astrid, die vor allem in schwierigen Situationen immer ein Ohr für meine Sorgen hatte und mich nicht selten mit großartiger freundschaftlicher Unterstützung wieder zurück auf den richtigen Weg gebracht hat.

Abstract

In modern building automation systems a plethora of different networking technologies exists. Therefore, interoperability between devices using various technologies is a key requirement. The use of Web Services as a platform- and technological-independent method of communication is a promising approach to address this challenge. Since IP extensions to available technologies are more and more established in building automation systems the network infrastructure and necessary protocols for Web Services communication are already present. However, providing appropriate concepts to model information that can be accessed in a generic way are still missing. OPC Unified Architecture (OPC UA) is a powerful and promising standard that aims at solving this challenge.

This work is dedicated to the development of an approach to map the interworking model of BACnet to OPC UA. This includes both the data representation of BACnet and the services used to access and modify this data. This way, an OPC UA information model emerges. Using this information model, BACnet applications can be represented in OPC UA and, thus, be accessed by OPC UA clients in a standard and well-defined way.

At the beginning, an introduction into the BACnet standard is given. After an overview of the protocol architecture, the BACnet application model is described. This is followed by a section about the object oriented way of representing process data. BACnet services necessary to access these data and for configuration purposes are illustrated in the last section of this chapter.

The next part of this work is dealing with a state-of-the-art standard in automation systems, namely OPC Unified Architecture. A brief review of the historical development process of this standard is given. The next section describes the overall architecture and the different parts of the OPC UA specification. The capabilities of information models and how they are derived from built-in elements of OPC UA is shown further on. In order to enable clients to access these information models, a set of services is defined in OPC UA. An excerpt thereof is presented in the following.

The main contribution of this work is to show how OPC UA is integrated in a BACnet network. The mapping of BACnet datapoints to an OPC UA information model constitutes the main part of this chapter. This includes also the way these datapoints are addressed in OPC UA. Finally, the mapping of BACnet services to OPC UA services is presented.

In order to prove the feasibility of the ideas developed in this work, an OPC UA server interfacing a BACnet network has been implemented and put into operation in the context of a test lab setup. For this purpose, an open source BACnet stack implementation was integrated in a server built on top of a closed source OPC UA server SDK. An OPC UA client available as freeware was taken to access process data of a BACnet network via the OPC UA server.

In the Appendix of this work an introduction to the security mechanisms of OPC UA is given. Since these mechanisms rely on software certificates, a strategy must be defined how to manage these certificates, i.e. an organised way of distribution, validation and revocation needs to be found. In general, there exist different concepts of how to achieve this goal. The Appendix gives an overview of these concepts and frameworks and discusses their positive and negative aspects depending on the structure of different environments in which OPC UA applications shall be embedded.

Kurzfassung

In modernen Gebäudeautomationssystemen existiert eine Vielzahl an verschiedenen Netzwerktechnologien. Aus diesem Grund ist Interoperabilität zwischen Geräten, die auf verschiedenen Technologien basieren, eine Schlüsselanforderung. Web Services als plattform- und technologieunabhängige Form der Kommunikation sind ein vielversprechender Ansatz, an diese Herausforderung heranzugehen. Da sich inzwischen IP-basierte Erweiterungen zu existierenden Technologien in Gebäudeautomationssystemen mehr und mehr etablieren, sind in vielen Fällen die Netzwerkinfrastruktur und die notwendigen Protokolle für die Kommunikation über Web Services bereits vorhanden. Geeignete Konzepte zur Informationsmodellierung und für den generischen Zugriff auf diese Modelle fehlen jedoch bislang. OPC Unified Architecture (OPC UA) ist ein mächtiger und vielversprechender Standard, der darauf abzielt, diese Problematik zu lösen.

Das Ziel dieser Arbeit ist es, einen Ansatz zu entwickeln, nach dem das Interworking-Modell von BACnet auf OPC UA abgebildet werden kann. Dies beinhaltet sowohl die Art und Weise wie Prozessdaten in BACnet repräsentiert werden, als auch die Services, die benötigt werden, um auf diese Daten zuzugreifen und diese zu verändern. Hierbei entsteht ein OPC UA-Informationsmodell, das dazu benutzt werden kann, BACnet-Anwendungen in OPC UA zu repräsentieren. Auf dieses kann ein OPC UA-Client auf standardisierte und wohldefinierte Weise zugreifen.

Zu Beginn dieser Arbeit steht eine Einführung in BACnet. Nach einem Überblick über die Protokollarchitektur wird das BACnet-Applikationsmodell beschrieben. Darauf folgt ein Abschnitt über die BACnet zugrundeliegende, objektorientierte Repräsentation von Prozessdaten. Die Services in BACnet, die für die Konfiguration der Geräte und für den Zugriff auf die Prozessdaten notwendig sind, werden im letzten Abschnitt dieses Kapitels beleuchtet.

Das folgende Kapitel beschäftigt sich ebenfalls mit dem Stand der Technik in Automationssystemen, im speziellen mit dem Standard OPC Unified Architecture. Ein kurzer Rückblick in die geschichtliche Entwicklung dieses Standards leitet dieses Kapitel ein. Dann folgt eine Beschreibung der Architektur und der verschiedenen Teile der OPC UA-Spezifikation. Die Möglichkeiten, die neue Informationsmodelle bieten und wie sie von den in OPC UA bereits vorhandenen Elementen abgeleitet werden können, werden im folgenden gezeigt. Um Clients den Zugriff auf diese Informationsmodelle zu ermöglichen, ist eine Anzahl von Services in OPC UA definiert. Ein Auszug daraus wird im nächsten Abschnitt gezeigt.

In dem Teil der Arbeit, der dem eigentlichen Beitrag gewidmet ist, wird gezeigt, wie OPC UA in ein BACnet-Netzwerk integriert werden kann. Die Abbildung von BACnet-Datenpunkten auf ein OPC UA Informationsmodell bildet den Hauptteil dieses Kapitels. Diese beinhaltet

auch den Transfer des BACnet-Adressschemas. Schlussendlich wird noch die Abbildung von BACnet-Services auf OPC UA-Services gezeigt.

Um die Machbarkeit der hier entwickelten Ideen zu evaluieren, wurde ein OPC UA-Server implementiert, der mit einem BACnet-Netzwerk interagiert. Zu diesem Zweck wurde ein Open Source BACnet-Stack in einen OPC UA-Server integriert, der auf einem Closed Source SDK für OPC UA-Server aufbaut. Ein OPC UA-Client, der als Freeware verfügbar ist, wurde verwendet, um auf die Prozessdaten aus dem BACnet-Netzwerk über den OPC UA-Server zuzugreifen.

Der Anhang dieser Arbeit beschäftigt sich mit den Securitymechanismen in OPC UA. Da diese Mechanismen auf Softwarezertifikaten basieren, müssen Strategien gefunden werden, wie diese Zertifikate in einem Automationsnetzwerk verteilt, validiert und widerrufen werden sollen. Hierfür sind verschiedene Konzepte bekannt, die in diesem Teil der Arbeit beleuchtet und im Kontext verschiedener OPC UA Anwendungen miteinander verglichen werden.

Contents

CHAPTER 1

Introduction and Motivation

1.1 Building Automation Systems

Nowadays, *Building Automation Systems (BAS)* are a well established way to provide automatic control of indoor conditions in functional buildings [11]. The core domains of BAS are heating, ventilation and air conditioning *(HVAC)* as well as lighting applications. The overall aim is to improve comfort, save energy and hereby reduce costs arising during the lifetime of a building. Relatively new fields of operation where BAS gain importance are security applications like access control. Also recently safety critical applications like fire alarm systems are integrated. Figure 1.1 shows the typical structure and components of a BAS.

In order to physically interact with the environment, sensors and actuators are deployed. Sensors gather information about the current state of the process under control by, for example, metering temperature, humidity, brightness or by detecting the presence of a human being. Actuators actively influence the condition of the environment under control. To call a few examples, setting the position of a valve of a heating system, switching and dimming light or by setting the airflow of a ventilation system are typical actuator functionalities. These applications are assigned to the *field level* of a BAS. Access to field level data is provided by standardised 0-10V or 4-20mA interfaces but also via *fieldbus* technologies like KNX [18], LONWorks [17] and M-Bus [12].

Control functionality, i.e. the execution of control loops on data prepared by the field level is typically achieved by so called *Direct Digital Control (DDC)* systems. The output data of these controllers is fed back to the actuators. Typically a decentralised approach is followed, which means that for example one DDC is set up for each floor of a building. Setpoints for physical values like the room temperature can be defined either locally by the residents of the building or in a centralised way via a backbone network by superior applications. The backbone network interface of the controllers located in this so called *automation level* also provides access to datapoints of the field level. A representative for a network standard applied in this tier of BAS

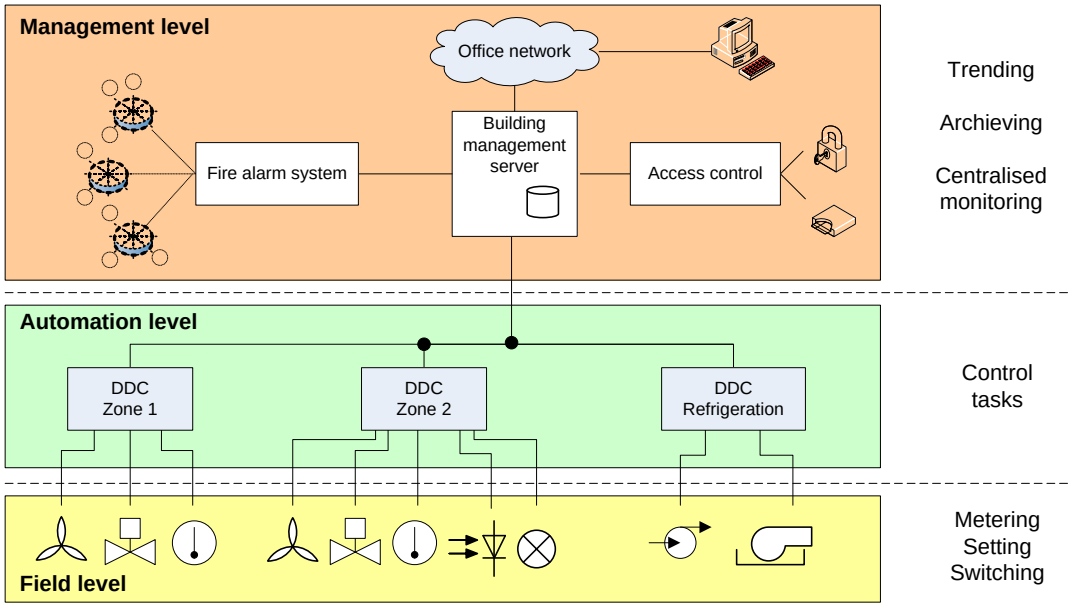

Figure 1.1: Three level model of BAS, adapted from [40]

is the Building Automation and Control Network (BACnet) [20], but also KNXnet/IP[1] devices can be found here.

At the top level of this hierarchy, the *management level* acts as an interface to *Building Management (BM)* and enterprise applications. Centralised access to datapoints, configuration of the system, visualisation, archiving and trending of process data are typical activities at this level. Another task of the management level is to provide interoperability between different systems and technologies used at the lower two tiers of the BAS (cf. Section 1.2). Applications belonging to the domains of safety and security are also integrated in this tier.

All these three levels together form a commonly agreed model of Building Automation Systems, the automation pyramid [11, 34]. The pyramid-shaped network topology in Figure 1.1 reflects this model. The automation pyramid separates the different functional aspects of BAS. At the bottom, there is the field level where interaction with the process under control takes place. The automation level is placed in the middle, aggregating and processing data delivered from below. On top of this model, the management level takes control over the infrastructure of the tiers below. The pyramid shape reflects the hierarchical control flow from the top level to the bottom level. Data reporting on the other hand takes place in the inverse direction. Equipment of an underlying tier provides data to the superior one. Another aspect that is illustrated by the pyramid is the number of devices operating in each level. A high amount of relatively plain and cheap devices can be found at the field level. The number decreases when going upwards, whereas the complexity and also costs for a single component rise.

[1] An IP based version of the KNX standard

1.2 Interoperability in Building Automation Systems

Especially at the interface between automation level and management level of BAS many different technologies join together. Individual vendors of automation equipment implement various protocol standards, like BACnet and KNXnet/IP as mentioned in the previous section. Since there is an inherent necessity of interfacing these different technologies by management level applications, a way must be found to learn them speak the same language. In other words, the goal is to achieve interoperability.

The classical, but not very convenient way to reach this goal is to integrate technology-specific interfaces (i.e. drivers communicating with the underlying networks) in the management level applications. The disadvantages of this approach are a lock-in to distinct technologies and vendors. Furthermore, little flexibility for future extensions including new network standards remains.

A more promising way is to define and agree on a *general application model* covering the functionality of these underlying systems. Hereby, a unified interface to these networks is introduced. Since IP based networks are not only commonly used at the management level of today's BAS but also at lower levels, the most suitable concept of communication is the use of Web Services (WS) [33]. WS have the advantage that they provide platform- and programming language independence. Based on the exchange of messages, WS are based on the Service Oriented Architecture (SOA) paradigm. This enables devices to exchange data independently of the underlying networking technologies.

Within this context, *OPC Unified Architecture (OPC UA)* is one of the most important standards supporting WS. While OPC UA is already well-established in industrial automation systems [27, 41], it gains importance within the building automation domain. In addition to the less used standards, such as oBIX [14] and BACnet/Web Services (BACnet/WS) [20], OPC UA can be used to provide a generic view to management clients that need global access to the entire BAS. However, to be able to use OPC UA at the management level interfaces to the underlying technologies are required.

Therefore, this work presents an approach how OPC UA can be integrated into one of the most important open BAS standards used at the all different levels of the automation hierarchy, namely BACnet. This thesis starts with an introduction into BACnet and its application model (cf. Chapter 2). In the following Chapter 3, the main concepts of OPC UA are described. In Chapter 4, a method of mapping the BACnet interworking model to an OPC UA information model is introduced. As a proof-of-concept, a prototype application of an OPC UA server interfacing BACnet/IP networks is presented in Chapter 5. The thesis is concluded with an outlook on ongoing research activities and future work (Chapter 6).

CHAPTER 2

BACnet

2.1 History and Development

The Building Automation and Control Network (BACnet) is an open communication standard for building automation systems. It was developed by the *American Society of Heating, Refrigerating, and Air Conditioning Engineers (ASHRAE)* and was standardized in 1995. Continuous maintenance and development are applied since then. Initially designed for the use at the management and automation level of the three tier automation hierarchy, nowadays BACnet has found a use in all kinds of building automation applications. Typical scenarios where BACnet is applied are control tasks in heating, ventilation and air-conditioning (HVAC) as well as classical lighting and shading systems. Access control and advanced lighting functionalities are the latest features of BACnet. The current standard is BACnet 2010 [20]. The ISO 16484-5:2010 [21] incorporates BACnet 2008.

2.2 Protocol Architecture

BACnet implements the *physical*, the *data link*, the *network* and the *application layer* of the ISO/OSI model. The data link layer and the physical layer which are affected by the selection of a so called network option are independent protocol standards and therefore not part of the BACnet specification. In the original BACnet standard, five network options describing the physical and the data link layer are defined:

- Ethernet
- ARCNET
- Master-Slave/Token Passing (MS/TP)
- Point-To-Point (PTP)
- LonTalk

Since this first version, two additional ones have been defined. A normative annex of the standard specifies the transmission of BACnet messages over IP (*BACnet/IP*) as a network option. The use of *ZigBee* [15] as a wireless network option for BACnet is specified since BACnet 2010. In the illustration of the protocol architecture given by Figure 2.1 these options for the physical and the data link layer can be seen.

The choice of a network option has no influence on the upper two protocol layers. It is even possible to use other data link/physical layer combinations since BACnet is not limited to these network options. This allows the combination of multiple network technologies in one BACnet network and thus, interoperability is provided at the upper protocol layers.

The protocol architecture encompasses also the network-, and the application layer of the ISO/OSI model. The omission of the transport-, session- and presentation layer results in this so called collapsed architecture. It was chosen to reduce the packet size and hence due to that the protocol overhead. This helps to save resources in end devices and allows the use of mass-produced, cheap processors in embedded BACnet devices.

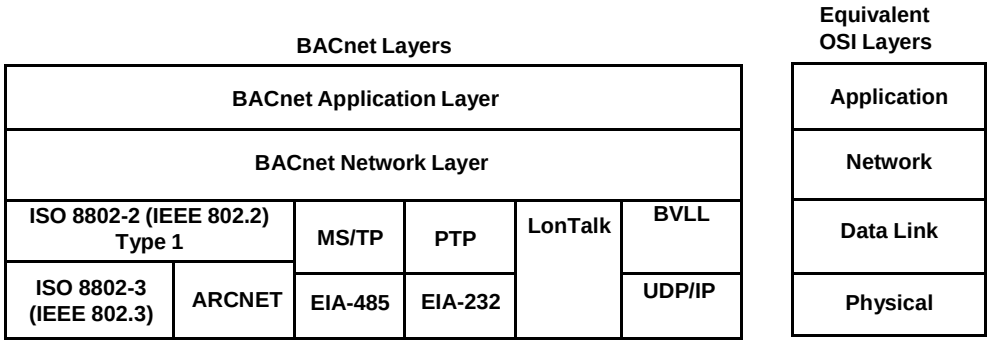

Figure 2.1: BACnet Collapsed Architecture [20]

The physical and the data link layer are unquestionably necessary in a building automation network. The network layer in this standard provides routing functionalities such as services for router discovery which are needed for communication between networks with different data link layer and physical layer options. This way, a so called BACnet *Internetwork* is formed.

Among others, two services as an interface to the application layer are provided, the `N-UNITDATA.request` and the `N-UNITDATA.indication` primitives. They represent an unacknowledged connectionless form of data transfer. The request service is called by the local application to initiate a data transfer, where the indication service informs the remote application about the reception of data.

Features usually implemented at the transport layer like flow control, segmentation and sequence control are moved to the application layer. This is justified by the very limitedness of

these services in a protocol based on a connectionless communication model. The session and the presentation layer are omitted completely since there are very few use cases where a need for typical services of these layers occurs.

2.3 Application Model

The application model of BACnet describes the relation of the application program to the application layer and the application layer to the underlying layers. As illustrated in Figure 2.2, it defines the *Application Process* as the part of the application which processes the information and handles the exchange of data between two peer applications. The application process in turn is divided into two parts: the *Application Program* and the *Application Entity* which is already part of the application layer. The former as well as the *Application Program Interface (API)* lying between the Application Program and the Application Entity are not specified in the standard. The *BACnet User Element* which forms one part of the Application Entity implements the service procedure portion of each application service. The service procedure portion manages the transaction context between communicating devices including the assignment of request and response messages to specific devices, retry mechanisms and the mapping of the activity of a device into BACnet objects (cf. Section 2.4). The other part of the Application Entiy is the *BACnet Application Service Entity (ASE)*. It represents a collection of five classes of services: *Alarm and Event*, *File Access*, *Object Access*, *Remote Device Management*, and *Virtual Terminal* which are responsible for different kinds of information exchange between the application processes. A selection of these services is described in Section 2.5 in more detail.

2.4 Data Representation

To allow remote devices to access process data, a "network-visible" representation of the stored data has been specified by BACnet. This representation follows an object-oriented approach known from the likewise called class of programming languages. Up to now, 30 different *BACnet ObjectTypes* are defined within the current BACnet standard. They differ in the composition of their so called *BACnet Properties* which can be seen as datapoints i.e., the logical representation of process data originating from the technical process under control. Figure 2.4 gives an example of such an object type definition. Each property has a unique identifier called *Property Identifier*, a designated *Property Datatype*, and a *Conformance Code* attribute. Data types can be primitives like bits, characters, strings and numbers in several formats or complex in order to combine more than one datum in a single property. The conformance code defines the access permissions of a property and specifies whether a property must be present in a distinct BACnet object or not. Valid values are `Readable(R)`, `Writable(W)` and `Optionally present(O)`.

Vendors of BACnet devices are free to define their own proprietary object types (referred to as nonstandard object types) – even the definition of proprietary property types is possible. However, there are three mandatory properties that must be defined for each BACnet object: `Object_Identifier`, `Object_Name` and `Object_Type`. The former two properties must be unique within a BACnet device. Since a BACnet object is always assigned to exactly one device (a BACnet object is never distributed across more than one device), the

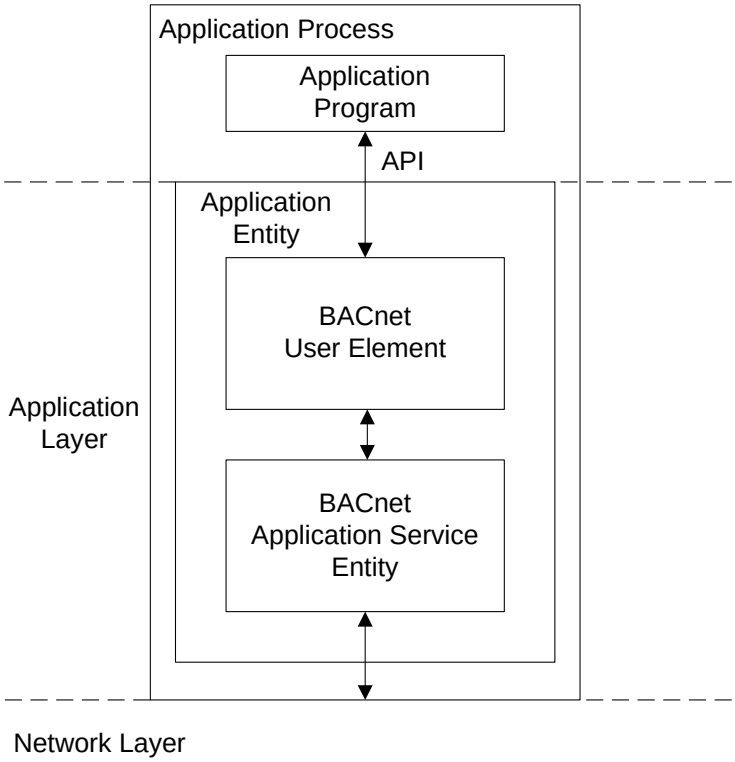

Figure 2.2: Model of a BACnet Application Process [20]

`Object_Identifier` or `Object_Name` property are used to uniquely identify a BACnet object within a device. The `Object_Type` property is actually an enumerated value which defines the kind of object.

Most BACnet object types available are generic ones like the `BACnet Binary Output Object` type and the `BACnet Analog Input Object` type. An extract of the definition of the former is shown in Figure 2.3. It is within the responsibility of the application program to map the control data of a dedicated application to certain BACnet objects. However, there are also efforts underway to standardise more application-specific object types in BACnet. For example, in *Addendum i* to BACnet 2010, the `BACnet Lighting Output Object` [23], a basic BACnet object for lighting has been defined. In the former *Addendum j* which is now part of BACnet 2010, object types for specific use in access control applications were specified. Figure 2.4 shows parts of the definition of the BACnet `Lighting Output Object`.

The `Present_Value` property usually holds a distinct physical value of the underlying process, for instance the actual state of the light. Concerning input or output objects, the `Present_Value` represents the signal values of the physical I/O ports of a BACnet controller. Other ones like the `Resolution` or the `Output_Type` provide additional meta information about the datapoint or the process under control.

Every BACnet device holds exactly one particular object called `Device Object`. An

Property Identifier	Property Datatype	Conformance Code
Object_Identifier	BACnetObjectIdentifier	R
Object_Name	CharacterString	R
Object_Type	BACnetObjectType	R
Present_Value	BACnetBinaryPV	W
Description	CharacterString	O
Device_Type	CharacterString	O
Status_Flags	BACnetStatusFlags	R
Event_State	BACnetEventState	R
Reliability	BACnetReliability	O

Figure 2.3: BACnet Binary Output Object [20]

Property Identifier	Property Datatype	Conformance Code
Object_Identifier	BACnetObjectIdentifier	R
Object_Name	CharacterString	R
Object_Type	BACnetObjectType	R
Present_Value	REAL	W
Progress_Value	REAL	R
Resolution	REAL	O
Binary_Present_Value	BACnetBinaryPV	O
Output_Type	BACnetLightingOutputType	R
Lighting_Command	BACnetLightingCommand	W

Figure 2.4: BACnet Lighting Output Object [23]

extract of the definition of this object type can be seen in Figure 2.5. The `Device Object` provides basic information about the BACnet device. In addition, its `Object_Identifier` and `Object_Name` must be unique within the whole BACnet (inter-)network. So they can be used to identify the BACnet device within the network. The `Object_List` property is an array of `Object_Identifiers` declaring the collection of BACnet objects instantiated in the device. Other properties represent the externally visible characteristics of the device like vendor information, firmware and protocol version, and local time and date.

2.5 Services

Services in BACnet are based on the client-server communication model. Typically, BACnet controllers interact with technical processes directly by their I/O ports or indirectly via underlying field devices. These controllers hold objects representing physical values of the processes and they act as servers. User control units or management workstations where values are set, process data are visualised and archived act as BACnet clients.

Property Identifier	Property Datatype	Conformance Code
Object_Identifier	BACnetObjectIdentifier	R
Object_Name	CharacterString	R
Object_Type	BACnetObjectType	R
Vendor_Name	CharacterString	R
Vendor_Identifier	Unsigned16	R
Model_Name	CharacterString	R
Firmware_Revision	CharacterString	R
Application_Software_Version	CharacterString	R
Object_List	BACnetARRAY[N]of BACnetObjectIdentifier	R

Figure 2.5: BACnet Device Object [20]

Generally spoken, there are *Confirmed Application Services* and *Unconfirmed Application Services* in BACnet. A device receiving a confirmed service request, needs to transmit a response to the sender. If an unconfirmed service is received, no response is required.

The complete set of service classes defined in BACnet consists of the following:

- Object Access Services
- Alarm and Event Services
- File Access Services
- Remote Device Management Services
- Virtual Terminal Services

However, in this section only the three most relevant classes of services with respect to this work are introduced. The description of these service classes is not complete but focused on the most relevant aspects.

Object Access Services

Two important representatives of the object access service class are the confirmed `ReadProperty` and the `WriteProperty` services for getting and setting the value of a property.

The `ReadProperty` service takes the `Object_Identifier`, the `Property_Identifier` and optionally the `Property_Array_Index` of the property that has to be read as arguments. If the property is an array and only one specific item is of interest, the element position can be passed to the service by the `Property_Array_Index` argument. If succeeded, the service response of a `ReadProperty` contains the input arguments of the request and the value of the property to be read. Table 2.1 shows the parameters of the request, the indication, the response and the confirmation service primitives. The symbology used in this table has the following meaning:

Parameter Name	Request	Indication	Response	Confirmation
Argument	M	M(=)		
Object Identifier	M	M(=)		
Property Identifier	M	M(=)		
Property Array Index	U	U(=)		
Result(+)			S	S(=)
Object Identifier			M	M(=)
Property Identifier			M	M(=)
Property Array Index			U	U(=)
Property Value			M	M(=)
Result(-)			S	S(=)
Error Type			M	M(=)

Table 2.1: Structure of `ReadProperty` service primitives [20]

- M: The parameter is Mandatory for the primitive
- U: The parameter is a User option and may not be provided
- C: The parameter is Conditional upon other parameters
- S: The parameter is a Selection from a collection of two or more possible parameters

One of these codes (M, U, C, S) followed by a "=" means that the parameter is semantically equivalent to the parameter to its left in the table. This convention also applies for the Tables 2.2, 2.3 and 2.4.

The `WriteProperty` service, on the other hand, takes the `Object_Identifier`, the `Property_Identifier`, the `Property_Array_Index` and the `Property_Value` of the property that has to be written as arguments. The corresponding write `Priority` is also passed. The reason why BACnet supports a priority mechanism is that a conflict regarding the value of a commandable property which is accessed by multiple applications needs to be resolved. This is achieved by introducing `Priority_Array` properties with fields indexed from `1` to `16`. For each commandable property of a BACnet object there exists one `Priority_Array`. Index `1` represents the highest priority and index `16` the lowest. The array fields may contain a distinct value or NULL. When a `WriteProperty` service accesses a commandable property, the `Property_Value` parameter is written to the field with the index corresponding to the `Priority` parameter, whereas the remaining fields are preserved. The `Priority` parameter must therefore also be in the range of `1` to `16`. The application of the BACnet device holding the object with the commandable property continuously monitors the `Priority_Array`. It takes the first non-NULL value of the array starting at index `1` and interprets this as the current value of the commandable property. In case the `Priority_Array` contains only NULL values, the value of the `Relinquish_Default` property is taken, which can be seen as a default value.

Parameter Name	Request	Indication	Response	Confirmation
Argument	M	M(=)		
Object Identifier	M	M(=)		
Property Identifier	M	M(=)		
Property Array Index	U	U(=)		
Property Value	M	M(=)		
Priority	C	C(=)		
Result(+)			S	S(=)
Result(-)			S	S(=)
Error Type			M	M(=)

Table 2.2: Structure of `WriteProperty` service primitives [20]

The specific parameters of the `WriteProperty` service primitives are shown in Table 2.2. The same symbology like in Table 2.1 is used.

Besides these service related parameters the destination network address must also be known when invoking these services. Success is indicated to the client by sending a positive confirmation response. A typical scenario for an unsuccessful service call is an access violation, i.e. in case of the `WriteProperty` service being applied on a property with a read only conformance code.

Alarm and Event Services

In this section, only the services responsible for the *Change of Value (COV)* reporting mechanism shall be described. Other representatives for the Alarm and Event service class are the *Intrinsic Reporting* services and the *Algorithmic Change Reporting* services.

The COV reporting services enable one or more BACnet clients to be informed about a change of a process value under control of a BACnet server. A temperature value is a typical datapoint to apply a COV subscription on. This can be done on temporary or permanent base. If a BACnet object supports COV reporting, a client may send a subscription request (`SubscribeCOV` or `SubscribeCOVProperty`) to the server holding this object. The former results in the server reporting on changes of pre-defined properties of the specified object (for most objects the `Present_Value` and the `Status_Flags` properties), the latter regards only a distinct property of the specified object. Parameters are the `Subscriber Process Identifier` for assigning the subscription to a specific process in both server and client, the `Monitored Object Identifier` of the object of interest, an `Issue Confirmed Notifications` flag indicating if a confirmation from the client after receiving a notification is required and the `Lifetime` of the subscription. The `SubscribeCOVProperty` additionally requires the `Monitored Property Identifier`. The destination address also needs to be passed to these requests. Table 2.3 summarises the parameters of the `SubscribeCOV-Property` service primitives.

Parameter Name	Request	Indication	Response	Confirmation
Argument	M	M(=)		
Subscriber Process Identifier	M	M(=)		
Monitored Object Identifier	M	M(=)		
Issue Confirmed Notifications	U	U(=)		
Lifetime	U	U(=)		
Monitored Property Identifier	M	M(=)		
COV Increment	U	U(=)		
Result(+)			S	S(=)
Result(-)			S	S(=)
Error Type			M	M(=)

Table 2.3: Structure of `SubscribeCOVProperty` service primitives [20]

If for example the shift of the present value property of an analog input object exceeds the offset determined by the `COV_Increment` property of the object, a `COVNotification` service or a `ConfirmedCOVNotification` service, respectively is generated by the server. The client which has applied the subscription on this object or property (or every device in the network, if the service is transmitted as a broadcast) receives the notification and can hereby obtain the new property value passed by the `List of Values` argument of the `COVNotification`. The `List of Values` argument contains the property values that need to be reported. BACnet specifies for every standard object that may support COV reporting a list of properties that need to be transferred and criteria that trigger a `COVNotification`. For details about these definitions confer to Clause 13.1 of the BACnet standard [20]. For correct origin assignment of the notification the `Subscriber Process Identifier`, the `Initiating Device Identifier`, the `Monitored Object Identifier` are transferred as well. The `Time Remaining` argument informs the client about the duration till the subscription will end. The parameters of the `ConfirmedCOVNotification` service primitives are illustrated in Table 2.4.

Remote Device Management Services

In order to discover BACnet networks and to find devices holding specific objects the unconfirmed `Who-Is` and `Who-Has` services are available. If one device broadcasts an unrestricted `Who-Is` to the network, every device (including the sender) responds with an `I-Am` service carrying the network address and the `Object_Identifier` of the respective `Device_Object`. The `Who-Is` can also be used to search for devices where the corresponding device ID is within a specific range. However, if the lower and upper bound of this range are set to the minimum and maximum `Object_Id`, all devices respond. By reading the `Object_List` of the `Device_Object` of every device that has responded, all objects within the network can be discovered. If one BACnet device wants to determine the

Parameter Name	Request	Indication	Response	Confirmation
Argument	M	M(=)		
Subscriber Process Identifier	M	M(=)		
Initiating Device Identifier	M	M(=)		
Monitored Object Identifier	M	M(=)		
Time Remaining	U	U(=)		
List of Values	M	M(=)		
Result(+)			S	S(=)
Result(-)			S	S(=)
Error Type			M	M(=)

Table 2.4: Structure of `ConfirmedCOVNotification` service primitives [20]

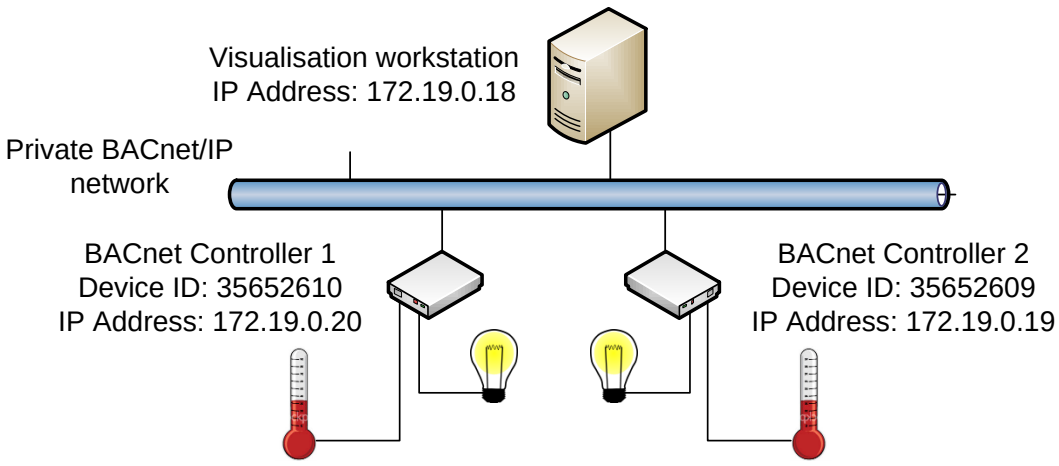

Figure 2.6: BACnet/IP network with two controllers and a visualisation workstation

address information of the device holding an object where the `Object_Name` or the `Object_Identifier` is known, it broadcasts a `Who-Has` service with the `Object_Name` or the `Object_Identifier` as a parameter. The device that finds the requested object in its database returns an `I-Have` message. This response carries the address of the device together with the `Device Object_Identifier` as well as the `Object_Identifier` and the `Object_Name` of the requested object.

The `I-Am` and the `I-Have` service may be broadcasted by BACnet devices every time and do not need to be preceded by a `Who-Is` or `Who-Has` service request.

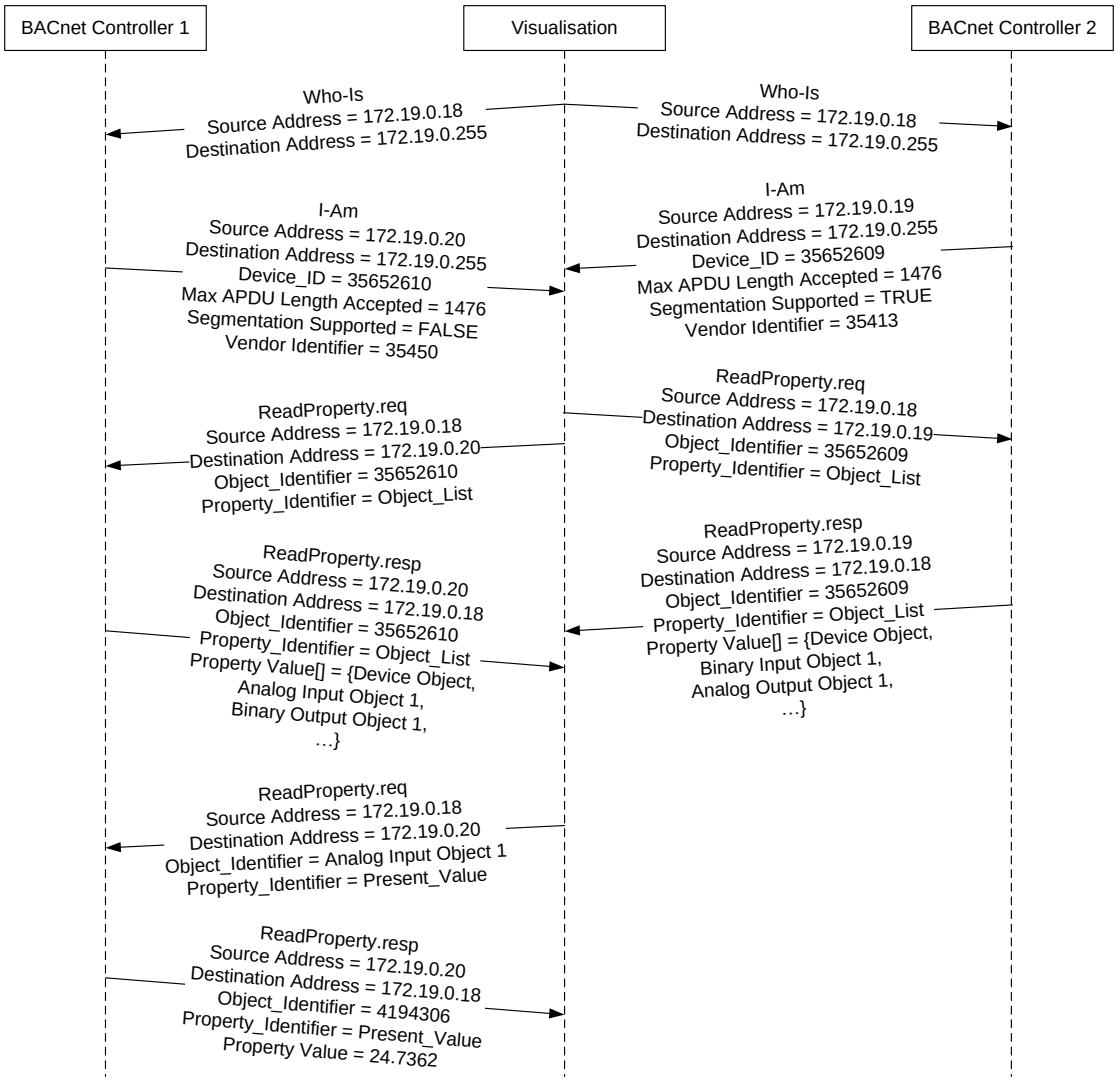

Figure 2.7: Network discovery and reading a `Present_Value` property

2.6 Use Case Example

This section shall give an example of how a BACnet network is discovered and how process data are gained from a BACnet object. Figure 2.6 shows the setup of a simple BACnet network consisting of two BACnet controllers acting as servers and one visualisation workstation acting as a client. These two BACnet controllers each implement an `Analog Input Object` representing a temperature sensor and a `Binary Output Object` which state is mapped to a binary light actuator. As obligate, each controller also holds a `Device Object` giving information about the properties of the controller and about the implemented BACnet objects.

Before the visualisation workstation can access any datapoint of the BACnet network, it has to discover which devices are actually present and which BACnet objects they currently hold. The first goal is achieved by broadcasting a `Who-Is` service request to the network. The sequence diagram in Figure 2.7 shows the communication between the three network devices. After receiving the `Who-Is` service request, both controllers answer with an `I-Am` service respond. By its arguments it delivers the network address, the `Device Object Identifier`, communication specific properties (`Max APDU Length Accepted`, `Segmentation Supported`) and the `Vendor Identifier` of the controller. Now, since the visualisation workstation has gathered the network addresses and the `Device Identifiers` it is able to do the next step in discovery, namely to determine which BACnet objects each controller implements. This is done by reading the `Object_List` property of the `Device Objects` of each controller. Issuing the `ReadProperty` service request with the `Device Object Identifier` determined before and the `Object_List` as arguments makes the controllers deliver the content of their `Object_List` properties to the visualisation client.

Now the visualisation workstation has completed the network discovery and has knowledge about which BACnet devices are currently present in the network and which BACnet objects these devices implement.

As seen in Figure 2.7 in the last `ReadProperty.req - ReadProperty.resp` sequence, an actual datapoint (the `Present_Value` of the `Analog Input Object 1` representing the value of a temperature sensor) can finally be accessed.

CHAPTER 3

OPC Unified Architecture

3.1 History and Development

In 1995, an association of vendors developing Human Machine Interface (HMI) and Supervisory Control and Data Acquisition (SCADA) software was founded. It targeted to address the drawbacks of the great plenty of vendor-specific fieldbus systems and protocols already available on the market but being not compatible among each other. The association was named *OPC Foundation*.

Its first release was a standard providing services for reading and writing process data. It was named *OLE for Process Control (OPC)*, since the protocol was based on *Microsoft OLE*. The idea behind OPC was that each vendor provides specific OPC drivers for (network) devices. These drivers link the individual (network) protocols to the OPC Application Programming Interface (API). This enables devices relying on different communication standards to exchange data and control information using the uniform OPC representation of data and services. In the beginning, Microsoft's *Component Object Model (COM)* and *Distributed COM (DCOM)* were used as APIs. This reuse of intellectual property enabled the foundation to focus on the development of important new features and quick adoption of the standard for the addressed use cases [35] which was an advantage of the OPC Foundation against other organisations. In addition to the original OPC standard which was later renamed to *OPC Data Access (OPC DA)*, additional specifications were defined. Examples are *OPC Alarm & Events (OPC A&E)* that describes the handling of event based information, and *OPC Historical Data Access (OPC HDA)* which specifies an interface to archived process data. These three different parts together with several other specifications form the *Classical OPC specifications*. They covered the majority of requirements in the domains of industrial and building automation.

Originally an advantage, the COM/DCOM dependency of these so called *classical OPC specifications* became more and more a limitation to many applications [38]. This is for several reasons. First, the limited remote access support of DCOM does not allow access over a Wide Area Network (WAN) like the Internet. Weak security mechanisms of DCOM do not make Internet connections recommendable, either. Second, the dependency on Microsoft Windows systems

is a constraint to some developers especially when designing software for low-power embedded systems. Third, there are compatibility issues of COM/DCOM between different Windows versions (e.g., Windows XP and Windows Vista/Windows 7). In addition to the insufficiency of COM/DCOM, another drawback of classical OPC was the weaknesses in modelling complex data and systems caused by the lack of object oriented concepts like using a type hierarchy. To eliminate these drawbacks, the *OPC Unified Architecture (OPC UA)* [19, 22] was released as a full replacement of the classical OPC specifications [31]. The main points of evolution of this new standard are:

- Combine all features from the classical OPC specifications into one specification
- Achieve platform independence by using Web Services and TCP based protocols for communication
- Allow remote access over the Internet
- Provide strong security mechanisms
- Use of a common object-oriented model for representing any kind of data
- Allow scalability in data complexity
- Offer the possibility to model meta information of process data
- Provide an abstract base model from which other user-defined models can be derived

3.2 Protocol Overview

Data modelling and transportation are the two core components of the OPC UA architecture. In the illustration of the foundation of the OPC UA standard (see Figure 3.1), two pillars corresponding to these two aspects can be identified. One of them is the *Transport* pillar, which describes a TCP based binary protocol for efficient communication and data exchange and a protocol based on Web Services, XML, and SOAP over HTTP (for more information about SOAP cf. [7]). It is intended to use HTTP(S) for data transportation in future. Both OPC UA protocols allow access of data via a WAN like the Internet. One advantage of the use of Web Services is that most of the network components like firewalls are already configured properly for passing them through. The exchange of data in OPC UA follows the client-server model.

The *Meta Model*, illustrated in form of another pillar, defines basic modelling constructs and rules how to model data. Here also the base types used for building type hierarchies are specified. Actual information models sit on top of these abstract definitions. Also concepts like state machines for modelling sequential control jobs are described in this part. A more detailed description of this part is given in Section 3.4.

Going upwards in Figure 3.1 there are two more parts lying on top of the two founding pillars. One of them is the OPC UA service part (Section 3.5). Services provide clients access to the information model lying on the servers.

The *Base OPC UA Information Model* is founded on the rules of the meta model. The structure of this part is shown in detail in Figure 3.2. Here the additional specifications known from the classical OPC standard like *Alarms & Conditions (AC)*, *Historical Access (HA)*, *Programs (Prog)* and new, automation specific *Data Access (DA)* features are included.

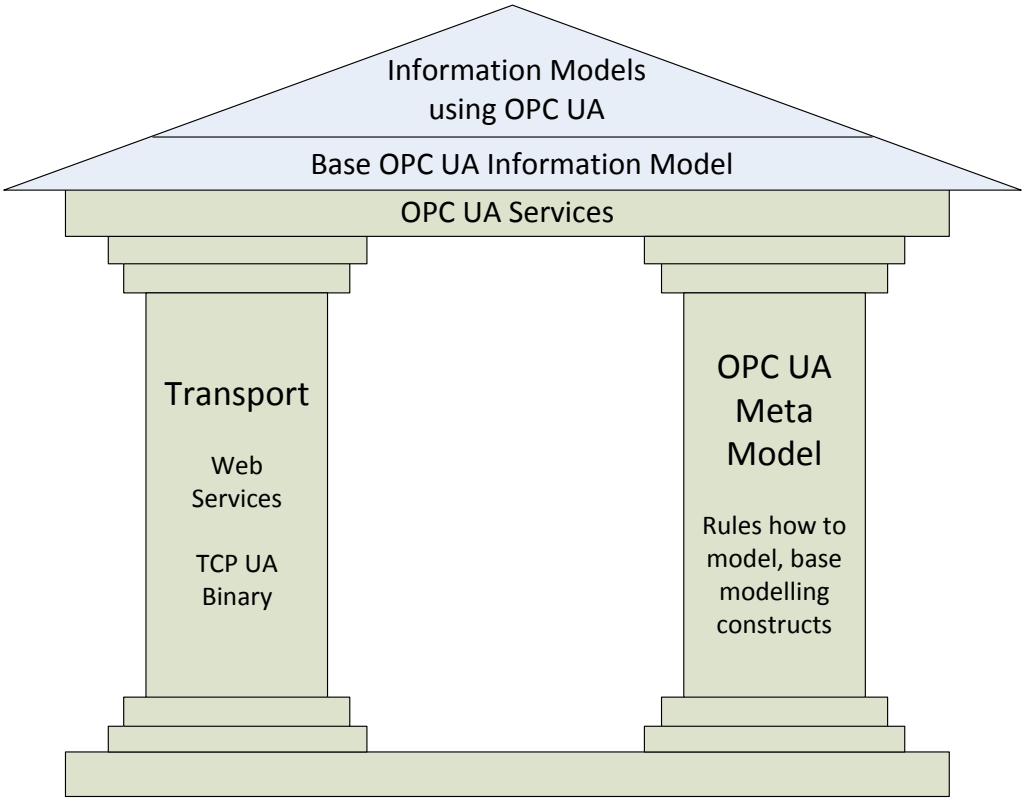

Figure 3.1: The foundation of OPC UA [35]

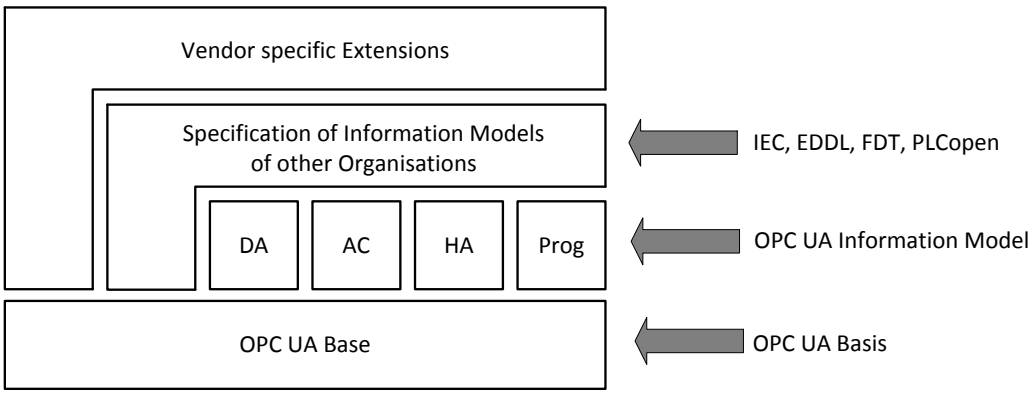

Figure 3.2: OPC UA layered architecture [35]

Standards published by, for example, the IEC[1] or other organisations use these OPC UA information models and build their own specific ones on top of it. The uppermost layer of information models is formed by vendor specific extensions designed for particular applications using the OPC UA Base, the OPC UA Information Models or other OPC UA based models.

3.3 Information Modeling in OPC UA

Contrary to classical OPC which only provides possibilities to represent basic data, OPC UA supports mechanisms to enrich data with specific semantics. For example, in addition to the measurement value of a sensor, information about the sensor type or the device that implements the sensor functionality can be modelled, too. This form of meta data can be interpreted by clients and used by applications to provide additional information related to process data. OPC UA defines the following rules regarding information modelling:

- Information is modelled in form of nodes carrying attributes and references linking the nodes (cf. Figure 3.3).
- Type hierarchies and inheritance are used as object-oriented principles.
- There is no distinction between the exposure of data and type information. The latter is needed by clients to interpret the data which is accessed.
- Information is modelled in form of a network of full-meshed nodes. There is no unique way to model information. Each use case requires a specific manner of modelling.
- The base information model as part of the specification is extensible with regard to defining subtypes of nodes and references between them.
- Information models only exist on OPC UA servers. Clients gain their knowledge about how data is modelled by fetching that information from the server.

Interoperability between devices of different vendors requires a uniform representation of data. In OPC UA, the idea is to define information models (i.e., data representations) for different application domains. Vendors can use these models to expose data of their applications or can even extend them by their own domain-specific knowledge. Clients do not have to distinguish between different vendors for their functionalities since they all have the same base model exposing data in common. Displaying current process data in a simple, generic user interface, access to historical data or event-driven update of data exposed or signalisation belong to these basic functionalities. If a server provides an information model with functionalities extending these basic ones, clients are able to interpret this more complex data by gathering the additional semantic from the information model. This way, advanced visualisation, more sophisticated computing or automated integration into other systems can be done with data provided by an OPC UA server.

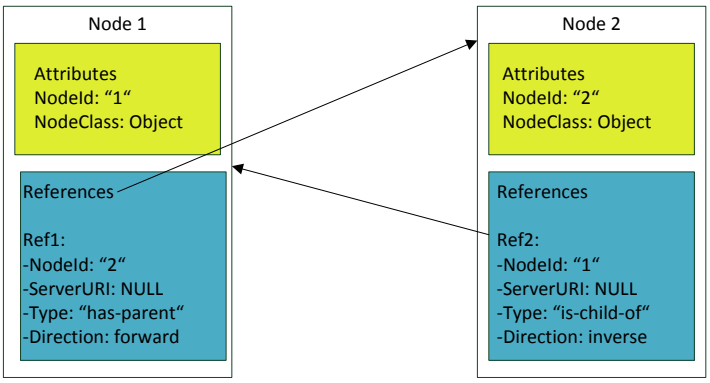

Figure 3.3: The concept of nodes and references [35]

3.4 Address Space

Information models in OPC UA are based on a meta model called *Address Space*. It contains definitions for basic data types, references, for creating variable types, object types, reference types, methods and further information entities. The underlying idea beneath the address space model is the concept of *Nodes* and *References*. Nodes in OPC UA consist of attributes which give a description of the node and references creating links between nodes (cf. Figure 3.3). Some attributes are inherent in all node classes, some are specific. Examples of common attributes are the `NodeId` for uniquely identifying the Node in the address space, the `BrowseName` which identifies a node when browsing through the address space, and the `DisplayName` attribute containing the name of the node to be displayed in a user interface. See [19] Part 3 for the entire list of attributes.

Each node is assigned to a distinct *node class*. In addition to the `Base` node class it can be distinguished between node classes defining types and node classes defining instances of types. Type definition nodes can be *abstract* or *concrete*. From abstract types no instance can be derived. They are used to aggregate common attributes of their subtypes and make the structure of a type hierarchy more organised. Concrete types are directly instantiable.

The following built-in instance definition node classes are defined in OPC UA:

- `Variable` node class: variables must always belong to another node (e.g., an object). The `Value` attribute holds a physical value of a technical process (if it is linked by a `HasComponent` reference) or provides meta information, i.e. characteristics like an engineering unit for the superior node (when referenced by `HasProperty`). In this case, a variable is called *Property*. Properties can neither be of a complex type nor have any subtypes.

[1] International Electrotechnical Commission [4], a not-for-profit, non-governmental standardisation organisation

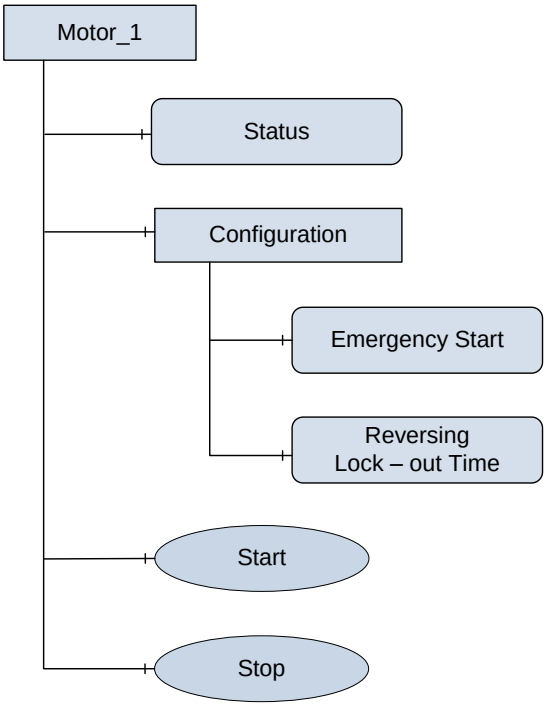

Figure 3.4: Example of a complex Object [35]

- `Object` node class: objects consist of variables, methods, and properties. They are used to model devices or components of the technical process under control, like a temperature controller or a motor controller.
- `Method` node class: methods are always referenced to an object. They represent functions that can be called by an OPC UA client (e.g., start and stop routines of a motor controller object).
- `View` node class: in order to reduce the scope of a client accessing an information model on a server, views can be used to make only parts of it visible. Depending on the use case, only the relevant part of the whole model can be made visible to the client.

Figure 3.4 shows an example of a complex OPC UA object which models an electric motor. The `Motor_1` object exposes a `Status` variable informing about the current state of the motor. Also the `Configuration` sub-object holds two variables representing two attributes of the motor configuration. This sub-object fulfils the purpose of structuring this model by grouping semantically related variables. At last, two methods (`Start` and `Stop`) are available providing the according functionality.

Users can extend the built-in information model by defining their own use case specific type definitions. These types are defined by inheriting them from built-in ones, enhanced with additional semantics and user-defined names (*simple types*) or by defining further sub nodes (*complex types*). The latter will be shown in Chapter 4.

The built-in type definition node classes are the following:

- `DataType` node class: defines the data type of the value attribute of a variable or variable type. `DataTypes` are organized in a type hierarchy, with the abstract `BaseDataType` on the top. Typical built-in `DataTypes` are `Boolean`, `String` or `Number`. Subtypes of the abstract `Number` DataType are `Integer`, `Float` and `Double`.
- `VariableType` node class: used to define the type of a variable. There are simple `VariableTypes` which only define the semantics and the data type to be used for the value attribute, where complex variable types hold a structure of nodes which enables structuring the variable type into sub values.
- `ObjectType` node class: specifies the type of an object. `ObjectTypes` can also be complex or simple where the difference is whether they expose a structure of other nodes beneath them or not. Complex ones can hold other objects, variables, and methods. This allows the engineer to create models of technical devices that reflect the entirety of the relevant device properties.
- `ReferenceType` node class: used to specify reference types. References in OPC UA derived from reference types are applied to create a link between two nodes. There are both abstract `ReferenceTypes` and concrete ones. The idea is the same as for `DataTypes`, namely to aggregate common attributes of sub types within an abstract super type and generate a more structurised type hierarchy this way. References can either be symmetric or asymmetric, depending on whether they have the same semantics in both directions or not. The `Symmetric` attribute of the `ReferenceType` indicates this property.
  Although a reference type is handled internally as a node, references do not have attributes directly accessible – only indirectly by browsing a node. However, reference types follow the same extensible concept as nodes. Users can likewise inherit special reference types from built-in ones in order to give them the required meaning. References are divided into hierarchical and non-hierarchical ones. Hierarchical reference types are typically used in type hierarchies (e.g., the `HasSubtype` reference) or when assigning properties to objects or variables by a `HasProperty` reference. The `HasTypeDefinition` is a typical non-hierarchical reference that relates an instance node to its type definition node.

The type definition (recognisable by the shadow behind the object rectangle) of the complex object in Figure 3.4 can be seen in Figure 3.5. This type definition has the same structure as the associated instance. All the sub-nodes of the `MotorType` object type are *InstanceDeclarations*, which means that they are also instantiated when instantiating the super-node. This way, the structure of a complex object type is preserved.

3.5 Services

The services in OPC UA are used to exchange data between OPC clients and servers. Each service is composed of a request and a response message. If a client wants to access data of the information model lying on the server, it calls a distinct method of the service set. After processing this request, the server returns a response to the client. Contrary to the classic OPC

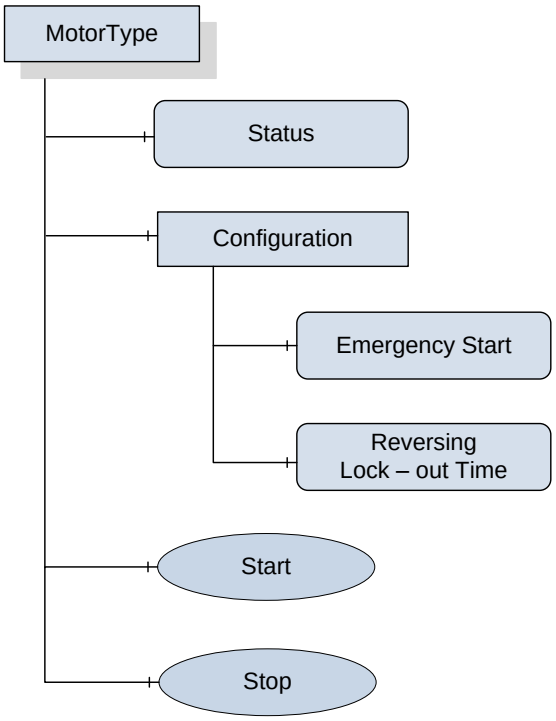

Figure 3.5: Example of a complex ObjectType [35]

specification, services in OPC UA are defined independently to the transport protocol and the platform used which requires an abstract service description (cf. Part 4 of [19]). The mappings of this abstract definition to specific transportation protocols, such as Web Services or OPC UA TCP is defined in Part 6 of [19].

OPC UA gets along with a very generic and reduced set of services. This is possible since information is provided by the server address space. There is no need for specialised methods for accessing different types of data or information. This can all be done by simple read services or write services.

The services described in the following are only a small selection of the complete set of services defined in OPC UA. It is a limitation to the ones relevant to this work.

Discovery Service Set

Before a client can start communication with a server, it must discover the set of available servers in its network first. This is achieved by the *Discovery Service Set*. A *Discovery Server* holds a list of available servers in a network. Each OPC UA server going online registers itself to this discovery server. A client sending the *FindServers* request to the discovery server gets a list of so called *Endpoints* of the available servers returned. With this information available, a client can establish a connection using the proper settings to the desired server.

View Service Set

To find the node holding the desired data in the information model on the server, the client can directly access it using its `NodeId` or it must browse to it from a starting node called the *Entry Point*. Following the outgoing references the client reaches the destination node holding the data. This is done in a recursive way by calling the `Browse` service for each node on this path. It returns an array of references originating in the node and pointing to a target node. Filtering mechanisms help to reduce the amount of data returned. When the requested node is reached, it can be identified by its `NodeId`.

Attribute Service Set

This set of services provides access to the attributes of nodes which are uniquely identified by the `AttributeId` and the `NodeId`. These are passed to the service methods as parameters. The most essential services are the `Read` and `Write` service. Depending on what kind of access is desired, one of these is called by the client. This is the most common use case to access data. Attributes of an array-type can be accessed element-wise by passing an index argument to the service method. But it is also possible to read or write the entire set or a range of elements of this attribute type as a composite.

In the following a description of the most relevant (with respect to this work) parameters of the `Read` service request and the according response as well as the `Write` service request and its response is given:

- `Read` request:
  `nodesToRead[]` is an array of `nodeIds` and `attributeIds` that should be read.
- `Read` response:
  The `results[]` array consists of the attribute values as results of the read operations. The order is the same as in the `nodesToRead[]` array.
- `Write` request:
  The `nodesToWrite[]` parameter is an array of `nodeIds`, `attributeIds` and `values` that should be written.
- `Write` response:
  The `results[]` array holds the `statusCodes` that give information about the success of the write operations. The order is the same like in the `nodesToWrite[]` array.

In order to access historical values or events on an OPC UA server, the `HistoryRead` and the `HistoryUpdate` services exist. The former is very similar to its `Read` sibling but applied for historical data, where the latter is used to change historical data in hindsight. Historical data are not directly visible in the address space, therefore these special services are defined to gain access to it.

MonitoredItem- and Subscription Service Set

In many cases it is required to be kept informed when the value of a datapoint changes or an event happens. This can be achieved in OPC UA from the `MonitoredItem` services set. It

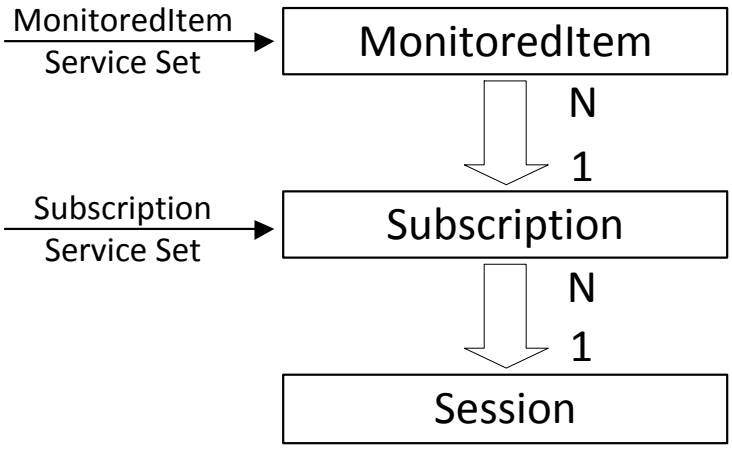

Figure 3.6: The relation between MonitoredItems and Subscriptions in a Session [35]

allows a client within an established communication *Session* to create `Subscriptions` on so called `MonitoredItems`. This context and the cardinality of the relations is shown in Figure 3.6. A `MonitoredItem` is an entity representing a source of information. Variable values, `EventNotifiers` and aggregated variable values are such possible sources of information. In the first and nearly most important use case, one or more variables in the server address space are continuously sampled. The sample rate is individually settable for each `MonitoredItem` by the client. The results of this sampling process are passed to the superior `Subscription` item. The `Publishing Interval`, an attribute of the `Subscription`, determines the time interval in which `Notifications` are generated by the `Subscription` item. A `Notification` is a message sent as a servers response (and is therefore also called `Publish Response`) to a `Publish Request` of the client which has applied the `Subscription` on the underlying source of information. To assure that `Notifications` can be generated by the server at any time a proper amount of `Publish Requests` needs to be pending.

`EventNotifiers` as another source of information are OPC UA objects combining several event sources. These can be for example changes of the configuration of the technical process or error conditions. `EventNotifiers` need not to be sampled since they are asynchronously triggered by the events they are representing. Data is passed to the `Subscription` item which again generates `Notification` messages.

As the third possible source of information it is also allowed to monitor aggregated variable values. The procedure is the same like for simple variable values, except that aggregated values are based on a calculation using simple ones as an input. These aggregated values are further transmitted as `Notification` (i.e. `Publish Response`) messages by the `Subscription` holding this `MonitoredItem`.

The point in time of transmitting `Notification` messages is determined by the server. After the client has sent a `Publish Request` to the server, the latter is not expected to answer immediately. This request can be queued until a `Notification` (`Publish Response`) is

ready to be sent.

In the following a description of the most relevant parameters of the CreateSubscription and CreateMonitoredItems service primitives is given.

- CreateSubscription request:
  The requestedPublishingInterval parameter determines the time interval between two Notifications returned by the server to the client. The value of this parameter is also used as the default sample rate of the assigned MonitoredItems.
- CreateSubscription response:
  The subscriptionId server-wide uniquely identifies the subscription created.
- CreateMonitoredItems request:
  The subscriptionId is the identifier of the Subscription the MonitoredItem should be assigned to. This Subscription will issue Notifications for the MonitoredItem.
  The itemsToCreate[] is an array of MonitoredItemCreateRequests containing the itemToMonitor (in turn holding the nodeId, the attributeId and other parameters specifying the monitoring process), the monitoringMode (to specify whether monitoring of the item is disabled, values are sampled but just queued, or Notifications are reported) and the requestedParameters variable. The latter defines, among other parameters, the sampling interval, filter rules and the queue size.
- CreateMonitoredItems response:
  The results[] parameter is an array of statusCodes giving information about the success of creating the requested MonitoredItems, Subscription-wide unique monitoredItemIds as identifiers of the MonitoredItems and parameters about the sampling intervals, queue sizes and filters applied on the MonitoredItem.

A detailed description of the complete set of OPC UA services and their parameters can be found in Part 4 of the OPC UA specification [19].

3.6 Use Case Example

In order to make the usage of OPC UA services more vivid, a typical use case of a client-server communication is shown in the following. A sequence diagram illustrating this communication can be seen in Figure 3.7.

It starts with the invocation of discovery services by the client with the goal to get information about the endpoints of OPC UA servers. It is assumed that there is a local discovery server present in the network which responses to the clients request. Now the client can send a GetEndpoints Request to the OPC UA server to get all the necessary information like the servers network address and the security settings to set up a connection.

Connection establishment between client and server is done next. This is achieved by setting up a secure channel and a session on top of it. After the session is activated by the Activate-

Session request and response service, the connection can be considered established. A more detailed description of this procedure especially regarding to security is given in Section A.3.

Now, by calling the Browse Request service containing one or more starting nodes, the client can explore the server address space and provide it either to another application interfacing the client or to the user in form of a graphic representation.

The next action in this use case is chosen to be a write access to one of the server address spaces nodes. Therefore, the client generates a Write Request containing the nodeId, the attributeId and the new value. This is answered by a Write Response informing the client about the success of the operation.

At this point the user decides to apply a data change Subscription on a distinct node value. First, a subscription needs to be created on the server. Then, the creation of a MonitoredItem with a variable value as a source of information follows. This MonitoredItem is assigned to the Subscription. After the server has acknowledged these requests, the client starts to send Publish Requests to the server. These requests are kept pending till a data change event happens and the new value can be transmitted in form of a Publish Response. If the value does not change within a defined period, the Publish Response of the server contains only a Keep Alive Message. Sending empty messages between server and client fulfils the purpose of detecting communication problems and keeping firewall ports open. The client needs to send new Publish Requests when it receives according response messages such that the server can continue sending Publish Responses.

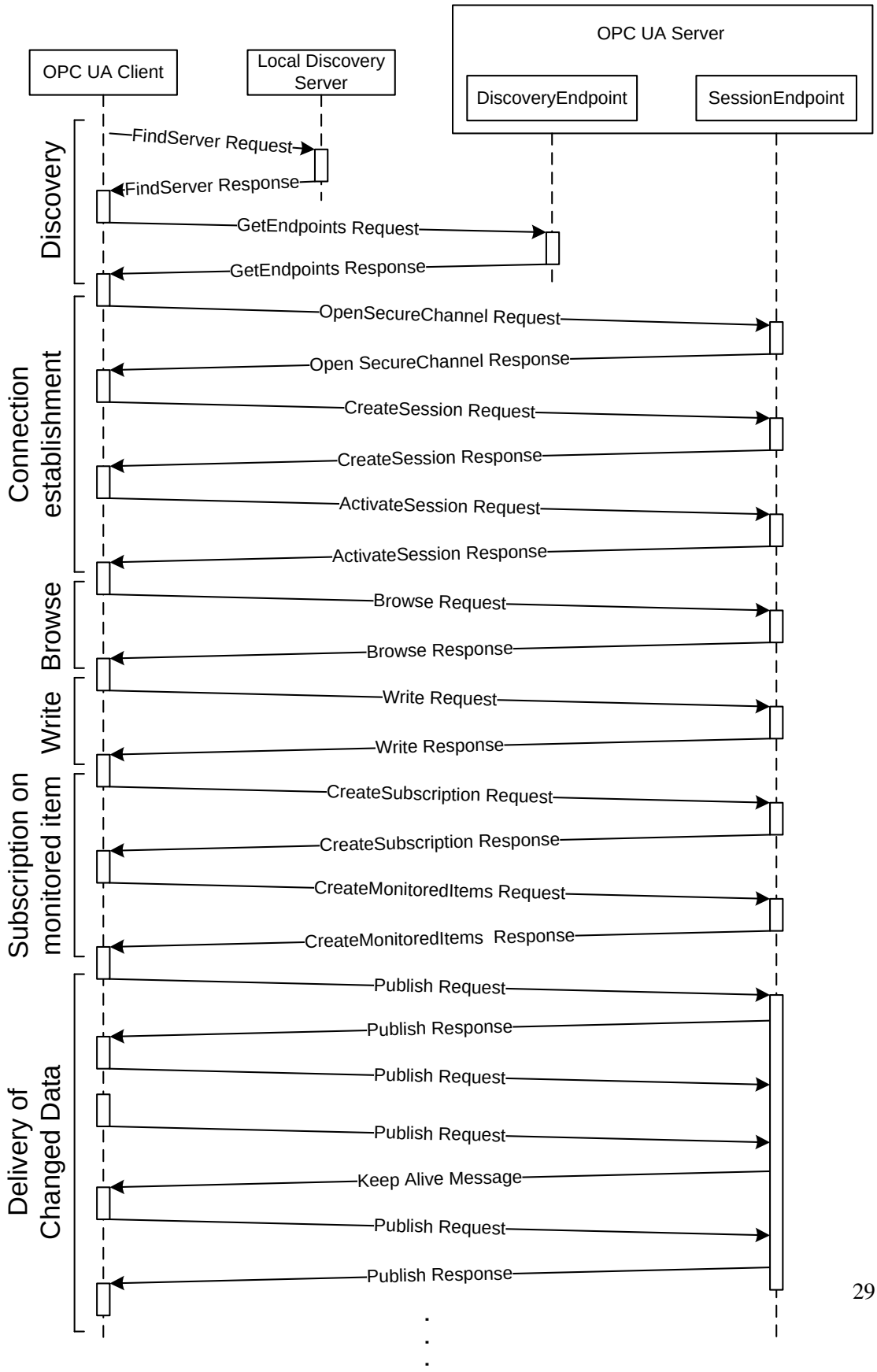

Figure 3.7: A typical use case of OPC UA services

CHAPTER 4

An OPC UA Information Model for BACnet

4.1 OPC UA at different levels of automation

The same challenges in the industrial automation domain the OPC foundation originally addressed when releasing the classical OPC standards exist in building automation networks. There are many different control and fieldbus protocols and technologies available but there exists only weak compatibility between them. OPC UA was primarily designed for application at the management level of the automation pyramid. Used at this tier, OPC UA can provide interoperability by abstracting the different, underlying networking technologies. It creates a uniform view of the process data and allows communication between network devices of different technologies. OPC UA clients, which are located within the same subnet or even in a remote network linked by a WAN can access datapoints of this process image for supervising purposes (for example, for visualization and trending applications). Another use case is taking over control of the process by an operator of a building management system.

In the meanwhile, there are efforts underway to integrate OPC UA in all different levels of the automation hierarchy. Examples are the OPC UA companion standards for IEC 61131-3/PLCopen [3] and Field Device Integration (FDI) [2]. The same that applies for the management level applies also for the control and the field level: Providing a common interface to devices using different technologies is a huge benefit when using them in a heterogeneous system. It results in a reduction of cost by making specific gateways unnecessary and it improves maintainability and scalability in a sense that components can easily be replaced or added.

Since OPC UA scales very well by adapting the set of profiles (cf. [19] Part 7) implemented and restricting them to the given requirements, OPC UA is applicable even down to field level devices. In this lowest level of automation systems mostly embedded devices with very limited hardware resources are called into action. Typically, the field devices in this environment acting as OPC UA servers do not have to provide the same amount of data like an enterprise level OPC

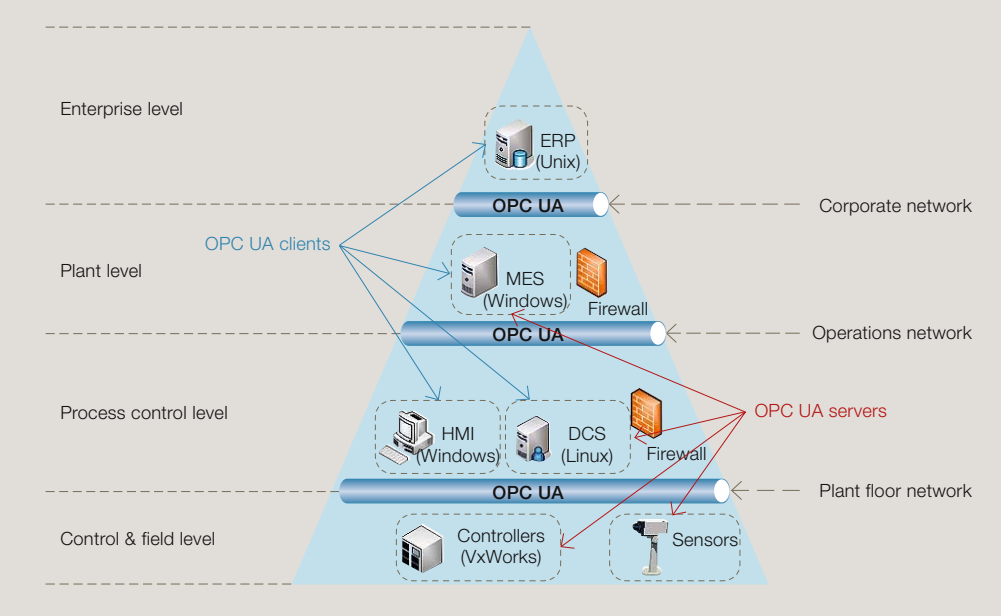

Figure 4.1: OPC UA in different levels of automation [42]

UA server [42]. On the other hand, demands on processing and response time are orders of magnitude higher. Since OPC UA also provides a binary data encoding which significantly reduces data and processing overhead, achieving reasonable performance even on low-end hardware is feasible.

Figure 4.1 shows how OPC UA can be applied in all different levels of automation. In this example, an industrial automation network divided in four tiers is shown. Slightly different terms are used in this domain, but the structure is similar to a building automation network and the requirements with respect to interoperability are comparable. Each level shown in Figure 4.1 has its dedicated network where data is exchanged in a horizontal manner (i.e. between devices of the same level). Devices at the lowest tier (field level) are already acting as OPC UA servers. At higher levels there are components which act simultaneously as OPC UA servers and clients. On their client side they interface the inferior level. With their server side they provide information to the superior network. This way, vertical communication between the tiers is established.

One further advantage of using OPC UA for data exchange in the whole pyramid is that a powerful security concept is available for the whole system (cf. Chapter A). As illustrated in Figure 4.1, the horizontal network traffic (i.e. communication between devices at different levels) is additionally filtered by firewalls. This assures that a security flaw in one level is unlikely to compromise devices of other levels.

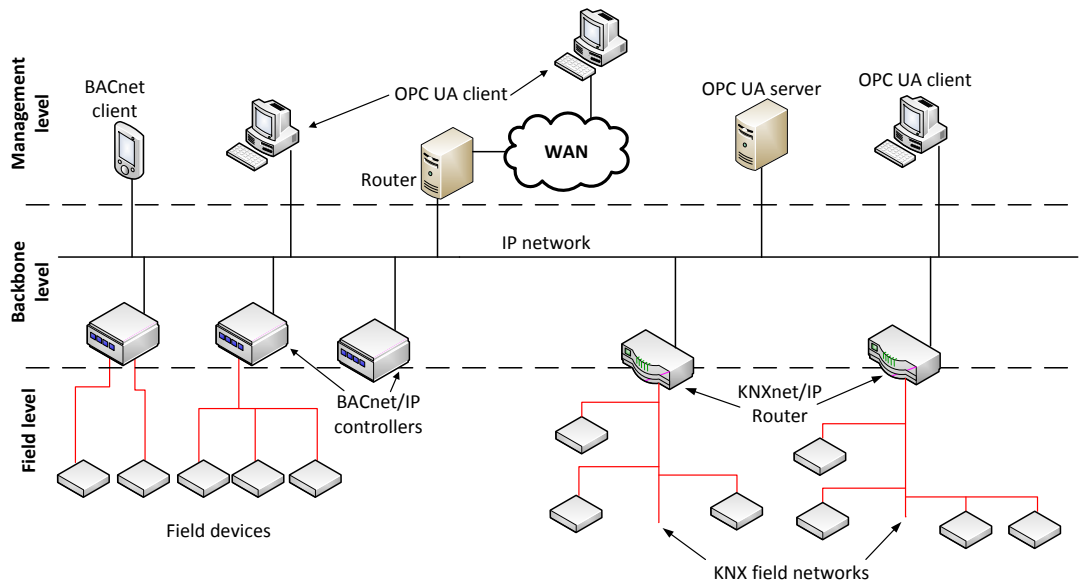

Figure 4.2: Use Case Example: OPC UA interfacing BACnet and KNX

4.2 OPC UA in a building automation network

Figure 4.2 shows a possible setup of an OPC UA server in a building automation network where at a backbone (automation) level two network protocols are applied: BACnet/IP and KNXnet/IP [**?**]. At the lowest level of the network hierarchy the communication between field devices and the superior controller is done by typical field protocols. At the BACnet side this could be for example LonTalk [17]. In the KNX part of the network, field devices communicate via KNX protocol over TP1 medium [16]. The KNXnet/IP routers encapsulate the KNX TP1 telegrams with a destination address in the superior level (the IP network) or at the other branch of the KNX network into KNXnet/IP datagrams. This is also done in the inverse direction, KNXnet/IP packets with a destination address in one of the KNX field networks are unwrapped and propagated as telegrams to the right KNX TP1 network.

In the use case example seen in Figure 4.2 the OPC UA server which is connected to the IP network is used to gather data from both different building automation networks to create a live process image of the whole network. This allows every datapoint of both the BACnet and the KNX network being accessible for an OPC UA client which is connected (whether locally or via a router and a WAN) to the server.

In the work presented, the focus is on building an OPC UA information model for BACnet. Using this information model, OPC UA servers and clients can be used to implement management applications that need to access data from BACnet networks. There is a significant resemblance in BACnet and OPC UA with respect to their data model. Both standards follow

an object oriented approach. However, the modelling concept in OPC UA is more advanced than in BACnet. The latter, for example does not support inheritance. Thus, defining a type hierarchy is not possible in BACnet. Section 4.3 presents how the interworking model of BACnet can be mapped to OPC UA. Another similarity exists in addressing the objects holding the process data. In BACnet, objects have an `Object_Identifier`, properties have a `Property_Identifier`. In OPC UA nodes are referenced by their `NodeId`. A mapping of these two addressing schemes is given in Section 4.4. Furthermore, the concepts of services used to access data are similar in both standards. Access services to read and write process data exist in both worlds (cf. Section 4.5). Alarm and event handling are also defined which allow for example, the monitoring of process variables and triggering of events or alarms in case a change of value happens or if a threshold is exceeded. Section 4.5 deals with the mapping of one representative of this BACnet class of services to OPC UA.

4.3 Mapping of BACnet Datapoints to OPC UA

Due to the advanced modelling capabilities of OPC UA the BACnet view of data can be mapped to OPC UA quite conveniently. The chosen approach is to transform BACnet objects to OPC UA complex objects. BACnet properties as members of BACnet objects are in turn mapped to OPC UA variables which are referenced by the corresponding OPC UA objects. In order to instantiate an entity in OPC UA, a type describing it has to be defined before. This needs to be done for objects, variables and references. BACnet-specific datatypes need to be defined in OPC UA as well.

DataType Definitions

Since the value attribute of an OPC UA variable is of a particular data type, the first thing to do is to define a data type hierarchy that represents the existing BACnet data types. Some of these BACnet data types can directly be mapped to the built-in OPC UA data types. For instance, the BACnet property data type `REAL` (e.g., used by the property `Present_Value` of a BACnet `Lighting Output Object` type) can be modelled as the OPC UA `Float` data type. However, there are more complex BACnet property data types that can not be represented by built-in OPC UA data types. Examples are the `BACnetObjectIdentifier`, the `BACnetObjectType` and the `BACnetLightingOutputType`. These BACnet data types have to be modelled as subtypes of OPC UA built-in data type `Structure` which can be used to represent complex data types. An exemplary part of it is shown in Figure 4.3. Relations between the type definitions in type hierarchies are always denoted by the `HasSubtype` reference. All user-defined BACnet data types are subtypes of the user-defined abstract data type `BACnetPropertyDatatype` which is inherited from the OPC UA built-in data type `Structure`. For each user-defined structured data type, at least one encoding has to be defined. This encoding is used by clients to correctly interpret the user-defined data representation. In the proposed model, `DefaultBinary` encoding is chosen for all user-defined data types. For every encoding, a description of the type (represented by a `DataTypeDescription-`

Type node) exists which in turn is a component of the BACnetPropertyDictionary. Within this user-defined dictionary, the entire encoding is described in XML format.

For the BACnet property type BACnetObjectIdentifier, this XML representation looks as follows[1]:

```
<StructuredType Name="BACnetObjectIdentifier">
        <Field Name="ObjectType"
               TypeName="Bit" Length="10">
        </Field>
        <Field Name="InstanceNumber"
               TypeName="Bit" Length="22">
        </Field>
</StructuredType>
```

VariableType Definitions

After having defined the BACnet data types, the BACnet properties have to be represented in OPC UA. To achieve this, user-defined OPC UA variable types are defined. Later on, they will be referenced by OPC UA objects representing BACnet objects. Each of the BACnet specific user-defined variable types is a subtype of the abstract user-defined BACnetPropertyType variable type. This abstract variable type exposes the user-defined OPC UA property BACnet-PropertyId which represents the BACnet Property_Identifier. It is inherited to each subtype when instantiating it. This attribute is unique for each BACnet property. A selection of such variable type definitions is shown in Figure 4.4. The Present_Value VariableType exposes an additional property, the BACnetPriority. This one represents the priority of write accesses in BACnet (cf. Section 2.5).

To create user-defined OPC UA variable types, the corresponding attributes of the new variable type have to be set. The DataType attribute is set to the corresponding built-in or user-defined OPC UA data type introduced before, except for the Present_Value variable type. In this case the DataType attribute is set to the abstract, OPC UA built-in BaseDataType. This allows to further specify this attribute when instantiating this variable type in order to reflect the real datatype of the BACnet property represented.

Examples for further attributes to be set are the BrowseName and the DisplayName which are both set to the human-readable name of the BACnet property defined in the standard. The ValueRank attribute provides information about the value attribute of the variable type being an array and if so, how many dimensions this array has. If this attribute is set to Scalar it means that the variable type represents a scalar datapoint. In case it is set to Any, the value attribute of the instance can be a scalar or a vector. This is the case for the Present_Value variable type definition. This allows to model BACnet objects with a Present_Value property being a scalar or an array using the same variable type definition.

ReferenceType Definitions

In the model proposed, there are two scenarios where references are used: To assign OPC UA variables standing for BACnet properties to OPC UA objects representing BACnet objects and

[1] For details about the XML representation refer to Part 3 of [19]

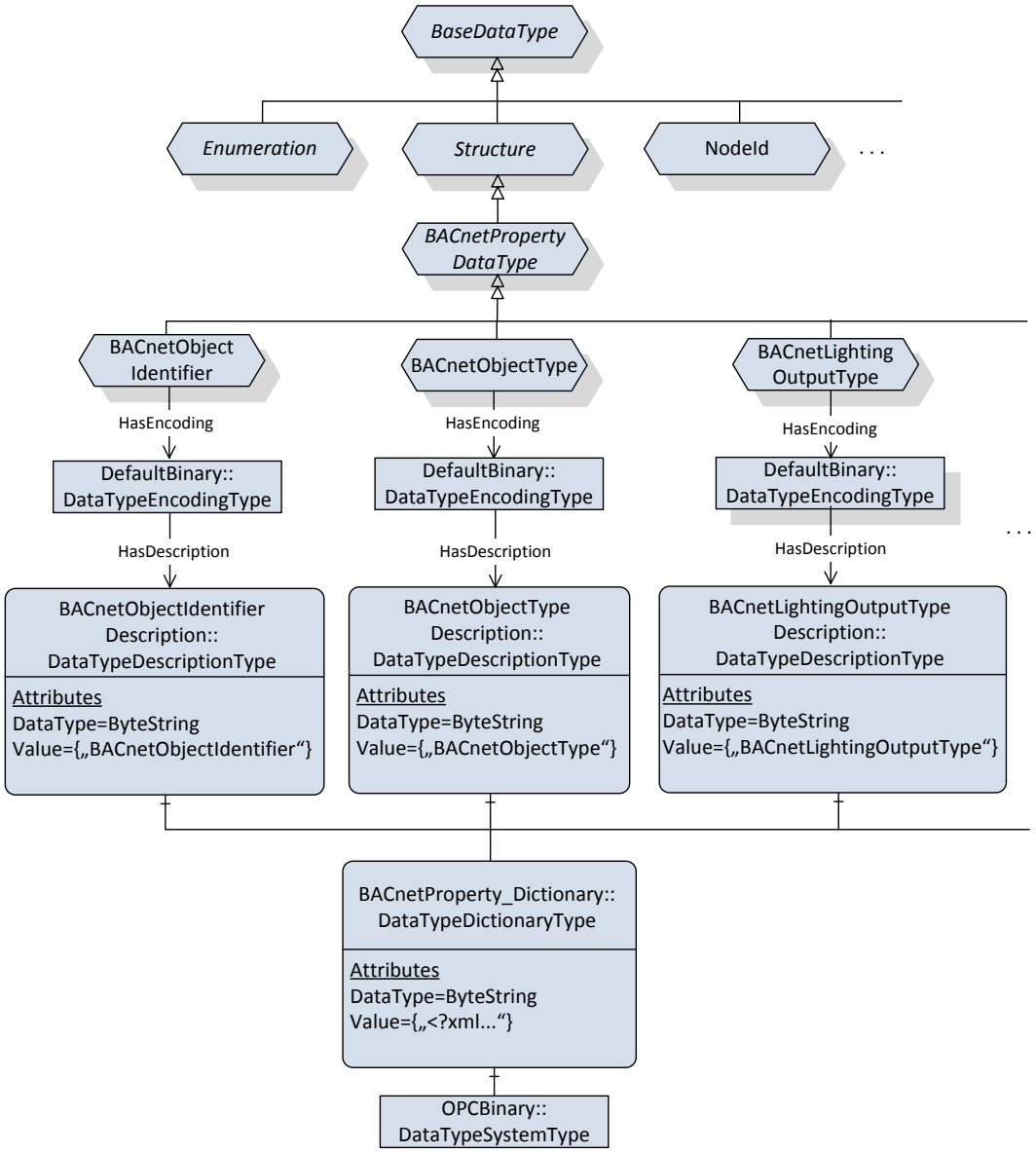

Figure 4.3: Datatype Definition

moreover to assign these OPC UA objects to further OPC UA objects representing BACnet devices. To express the special semantics of these references, the new reference types `HasBACnetProperty` and `HasBACnetObject` are introduced. These reference types are inherited from the hierarchical built-in one `HasComponent`. This type hierarchy can be seen in Figure 4.5.

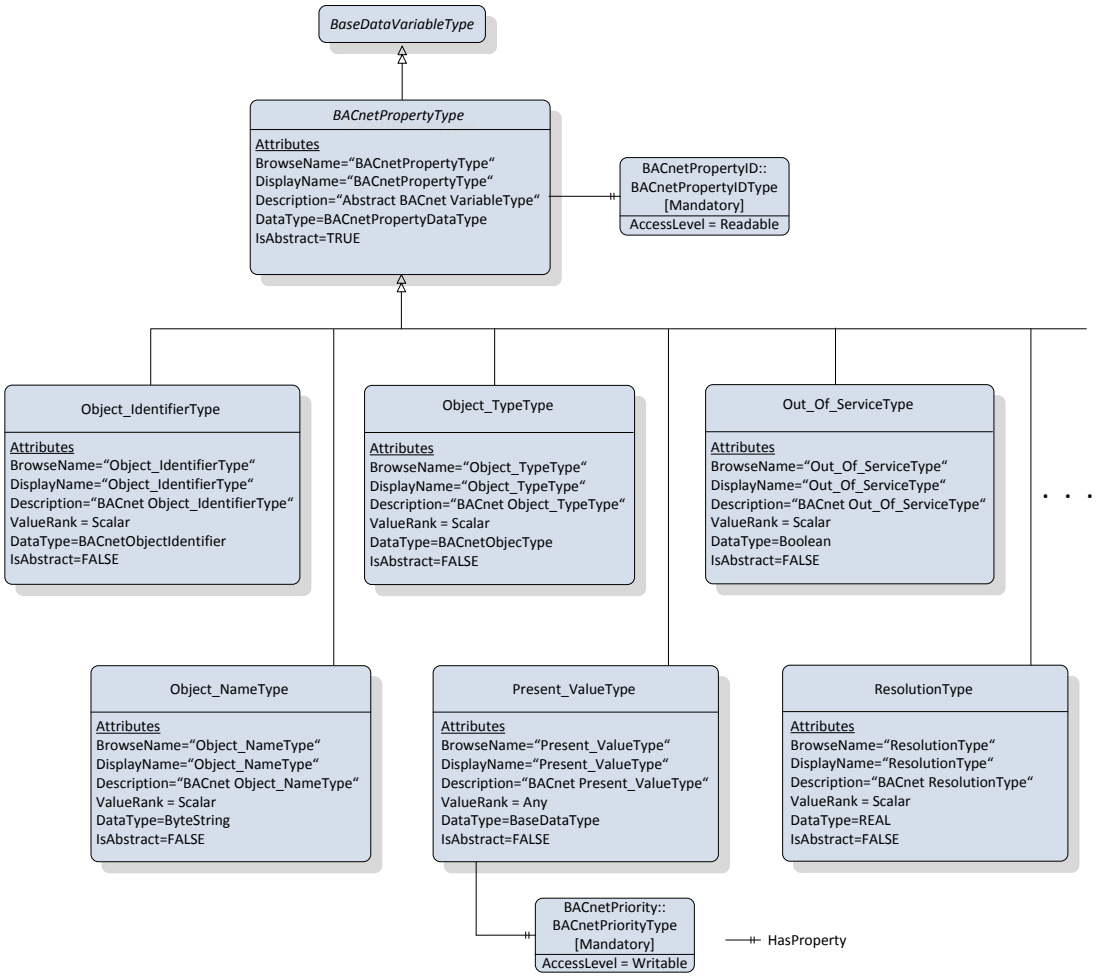

Figure 4.4: Variable type definitions

ObjectType Definitions

Now having all the necessary components available, the BACnet object types can be modelled in OPC UA. All BACnet object types are represented by user-defined OPC UA complex object types that are all subtypes of the abstract user-defined `BACnetObjectType` which in turn is a subtype of the built-in `BaseObjectType`. The `BACnetObjectType` contains the BACnet properties `Object_Identifier`, `Object_Name`, and `Object_Type` which are common to all BACnet objects. The assignment of the variables representing the BACnet properties to the corresponding object type is done by using the `HasBACnetProperty` reference mentioned before.

As can be seen in Figure 4.6, the variables reflecting the BACnet properties and the `BACnetDeviceObject` node have no shadows beneath them. This means that they are no type

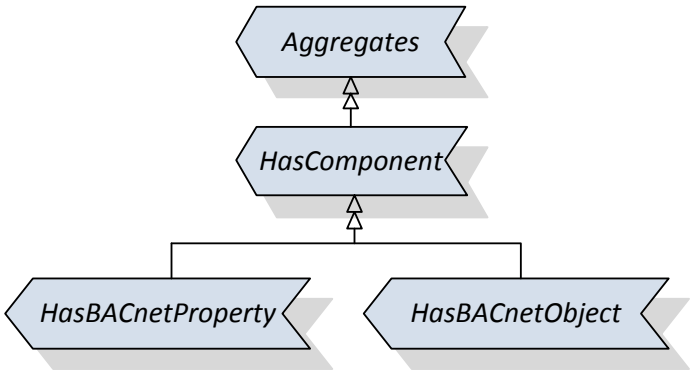

Figure 4.5: Reference type definitions

definition nodes like the `BACnetObjectType` node. Here so called *InstanceDeclarations* are present, which can be objects, variables or methods and are always subnodes of `ObjectType` nodes.

One other attribute needs to be set at variable instance declarations: The `AccessLevel`. This attribute informs the OPC UA client about access permissions to the particular variable. In the information model proposed the access permission facet of the conformance code of BACnet properties is mapped to this OPC UA node attribute. Possible values of the `AccessLevel` are `Readable` and `Writeable`.

When instantiating an `ObjectType` node exposing instance declarations, the latter are also instantiated (depending on the `ModellingRule` referenced). The key feature of this concept is that the relative paths from the `ObjectType` node to the instance declaration nodes are preserved when instantiating an object. This way, the structure of the type definition survives and exists in every instance derived from it.

`ModellingRules` give information about how an instance declaration will be treated when creating an instance of the type definition. In order to be instantiated together with the `ObjectType` node, each `InstanceDeclaration` referenced by a `TypeDefinition` must have a `ModellingRule`. Otherwise, they will not appear in the instance. In the model presented in this work, the `ModellingRules` `Mandatory` and `Optional` are used. The meaning corresponds to the name of the `ModellingRule`, a `Mandatory` one forces the `InstanceDeclaration` to be present in the instance, an `InstanceDeclaration` referencing an `Optional` one may be present in the instance. This way, the aspect of the conformance code of BACnet properties specifying whether a property must be present or not is modelled. A BACnet Conformance Code of `R` or `W` is mapped to a `Mandatory` `ModellingRule`, a Conformance Code of `O` is mapped to an `Optional` `ModellingRule`.

Inherited from the abstract `BACnetObjectType` all BACnet object types which are specified in the standard can be defined in OPC UA. Figure 4.6 shows an example how the BACnet `Device Object` type, the `Lighting Output Object` type and the `AnalogOut-`

putObject type are represented using this concept. As shown in this figure, only the object specific variables are defined – the common ones are inherited from the supertype. As it is common in OPC UA, the HasSubtype reference is used to model the relation between sub- and supertype. An example of how meta information can be modelled is also shown in Figure 4.6 in form of the EngineeringUnit node referenced from the Power variable of the Lighting Output Object. To model the assignment of a unit to the value of a variable, the UPC UA built-in reference HasProperty is taken.

If one compares the Present_Value node of the LightingOutputObjectType and the one referenced by the AnalogOutputObjectType, the datatype attribute of these two variables differs. The former one is set to BACnetLightingOutputType, the latter is set to the OPC UA built-in type REAL. As mentioned in Section 4.3 when defining the Present_ValueType, this attribute is set to a specific and concrete value when defining the instance declaration. Inheritance mechanisms of OPC UA allow to do this because the abstract BaseDataType is a supertype of all other datatypes.

In BACnet, each BACnet object is dedicated to exactly one BACnet device – BACnet objects are therefore never distributed across more than one BACnet device. Therefore, it is reasonable to take over this device-centric view – each BACnet device is represented as an OPC UA object instance of the user-defined object type BACnetDeviceType which in turn is a subtype of the standard OPC UA BaseObjectType (cf. Figure 4.6). The corresponding BACnet objects are assigned to the OPC UA object representing a BACnet device by using the user-defined HasBACnetObject reference.

The BACnetDeviceType in Figure 4.6 references an InstanceDeclaration of a BACnetDeviceObject. This InstanceDeclaration also exposes a Mandatory ModellingRule. This reflects the fact that on every BACnet device exactly one instance of the DeviceObject must be present.

Object Instantiation

After having presented how BACnet object and property types are modelled in OPC UA, it must be specified how instances of BACnet objects and properties are represented by the OPC UA server. Figure 4.7 shows an example how a BACnet device represented as an OPC UA object instance (BACnetDevice1) holding a BACnet Device Object and a BACnet Lighting Output Object is modelled. It has the same structure as its type definition node. Notice that the instances of the two objects also expose the properties all BACnet objects have in common (Object_Identifier, Object_Name, and Object_Type which are referenced by the abstract object type BACnetObjectType) and not only the properties specific to each object.

4.4 Mapping of the BACnet Addressing scheme to OPC UA

What is still remaining is how the BACnet properties exposed by BACnet objects can be addressed in the OPC UA information model. In BACnet, properties are addressed by the Property_Identifier which is unique within the object. In the information model proposed, this can be done by reading the BACnetPropertyID property that is dedicated to each BAC-

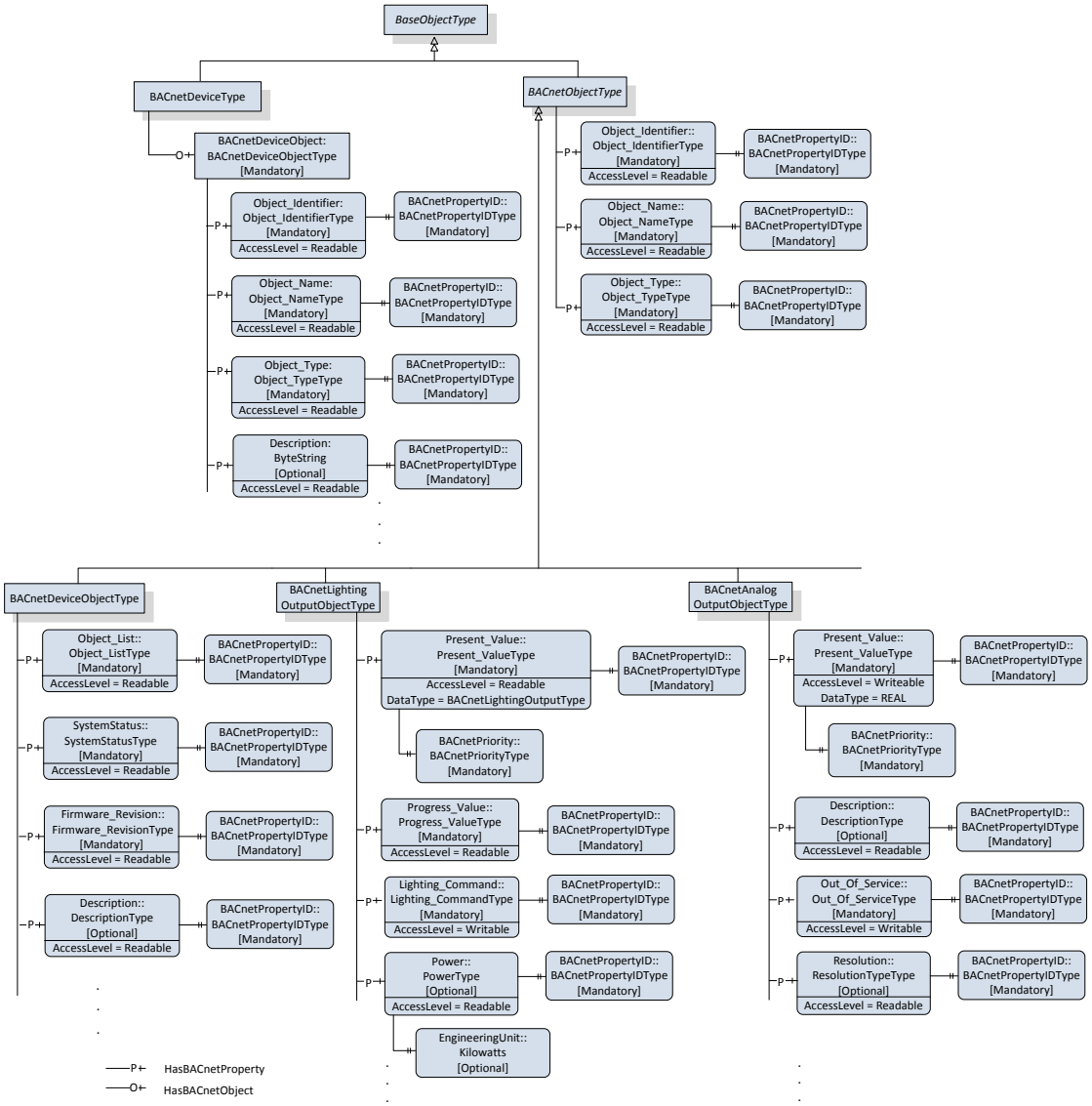

Figure 4.6: Object type definitions

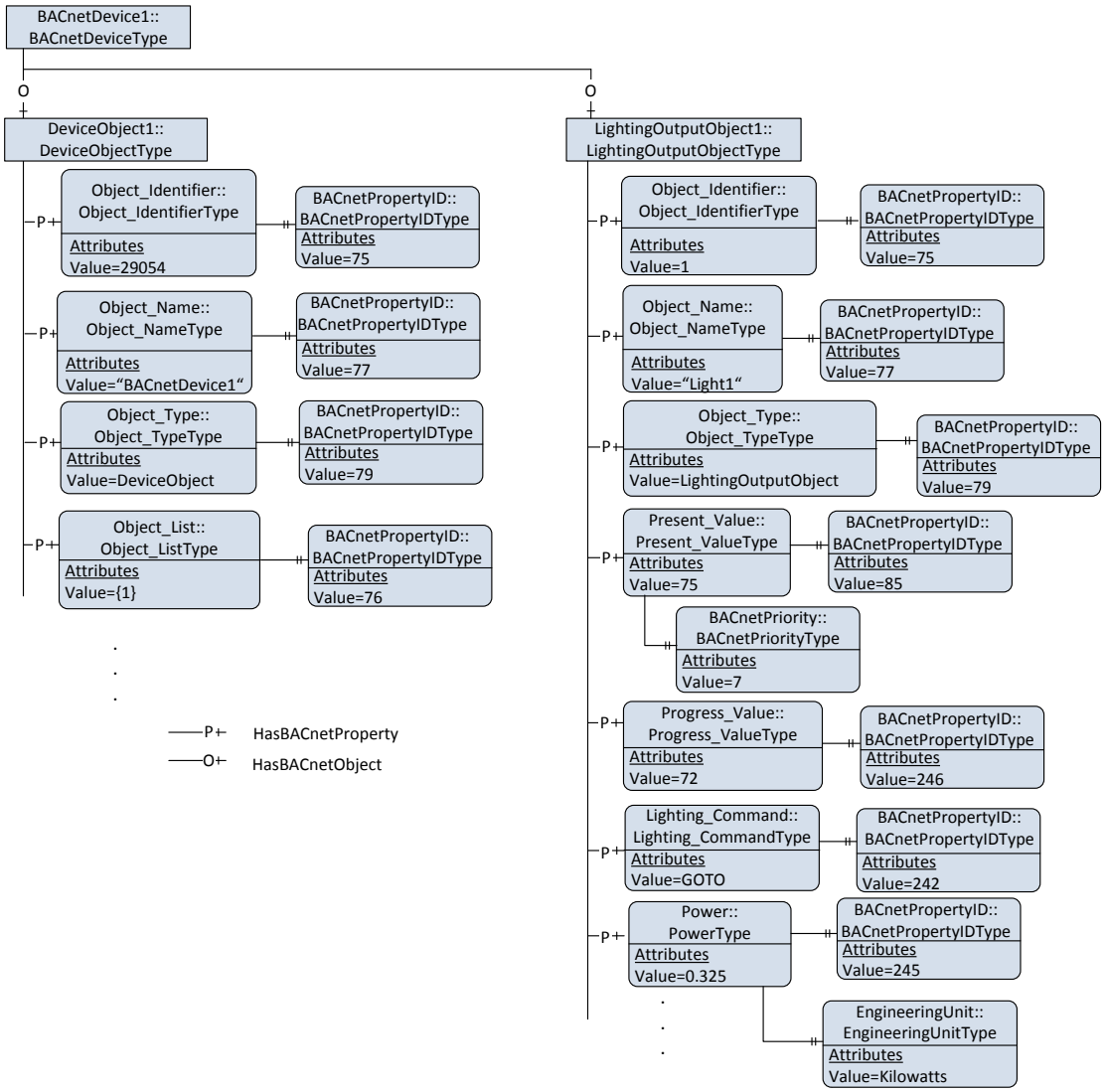

Figure 4.7: Instantiation of OPC UA objects representing a BACnet device with two objects

net variable definition. To address the BACnet object, the Object_Identifier which is unique within the device is used. The Object_Identifier can be determined by the reading the value of the Object_Identifier variable which is mandatory for each BACnet object. Finally, to address the device itself, the BACnet Device_Id or the Device_Name which are both unique within the whole BACnet network can be taken. To determine the BACnet Device_Id within the OPC UA model, the value of the Object_Identifier variable of the DeviceObject has to be read. To gather the Device_Name, the value of the Object_Name variable of the Device Object has to be retrieved. As a result, the combination of the value of the BACnetPropertyID property, the value of the OPC UA Object_Identifier variable, and the value of the Object_Identifier variable of the Device Object (or the value of the Object_Name variable of the Device Object) is used to address a BACnet property in the presented OPC UA model.

Figure 4.7 illustrates an instantiation of a BACnet Lighting Output Object. Consider, for example, an OPC UA client browsing to the Present_Value variable of the BACnet Lighting Output Object and attempting to read the value of it. To read its current value from the BACnet network, the OPC UA server needs to invoke the BACnet ReadProperty service. In order to generate this request, the address information used as parameters of the ReadProperty service have to be determined first. The network addresses is determined by the (technology-specific) BACnet driver interfacing the OPC UA server performing a network discovery before being able to access any BACnet device. Like the visualisation workstation mentioned in Section 2.6, the BACnet driver is also in the role of a client. It broadcasts a Who-Is request and gets I-Am responses from the network devices containing the network addresses and the Device_IDs of these controllers. Having this information, the driver can make an assignment of Device_IDs to network addresses. However, this procedure is transparent to the OPC UA server and therefore not part of the information model.

The other address parameters passed to the ReadProperty service request are gained from the BACnet information model. The Property_Identifier is determined by reading the BACnetPropertyID property of the Present_Value variable (in the proposed example 85). Afterwards, the value of the Object_Identifier variable is read (in the given example 1). Then, the Device_Id is determined by reading the Object_Identifier variable of the Device Object (in the proposed example 29054). This value is used to look up the network address of the according controller. Using the combination of the Property_Identifier, the Object_Identifier and the network address, the OPC UA server is able to generate the ReadProperty request and send it to the BACnet device. After having received the response (in the given example containing the value 75), the OPC UA server is able to forward the present value to the OPC UA client.

4.5 Mapping of BACnet Services to OPC UA Services

In a (building) automation network, where OPC UA and some domain specific network technologies are applied, OPC UA is implemented hierarchically spoken at the superior network layer and the domain specific one is lying below (cf. Section 4.1). Access procedures to process

data are always initiated at the upper level, where OPC UA clients reside. OPC UA servers provide the interface to the lower level of the automation network.

Derived from this fact, there is need for a transformation of OPC UA services to the domain specific services by the OPC UA servers. Applied to this case study of BACnet as a building automation network and OPC UA this results in following:

Service requests of an OPC UA client addressed to a server shall result in BACnet service requests generated by the BACnet driver interfaced by the OPC UA server. These BACnet service requests shall finally address devices in the BACnet network. The responses to these BACnet service requests received by the BACnet driver shall be propagated in form of OPC UA service responses to the OPC UA client.

This concept is applied to representatives of services used for accessing process data and also to an event handling service. A description of the service parameters used in this mapping can be found in Section 2.5 (BACnet Services) and in Section 3.5 (OPC UA Services).

OPC UA Attribute Services and BACnet Object Access Services

Mapping in this context means that an OPC UA `Read` or `Write` service request performed by an OPC UA client to an OPC UA server results in a BACnet `ReadProperty` or a `WriteProperty` service, respectively which is generated by the server and sent to the underlying BACnet network. In the course of these service calls, the arguments required by the BACnet object access services for addressing a datapoint have to be passed. This addressing information is gained from the information model by the server and also by the driver which has performed a network discovery as described in the section above (cf. Section 4.4).

Since write accesses are prioritised in BACnet, a way has to be found to represent this circumstance in the OPC UA information model. This is done by reading an additional property referenced by the `Present_Value` variable, namely the `BACnetPriority` property (cf. Figure 4.4). In case a write access is intended to be performed to a `Present_Value` variable, the `BACnetPriority` property must be read first. The result of this operation is passed to the BACnet `WriteProperty` service in order to propagate the right priority.

The parameters of the primitives of two services of the OPC UA attribute service set are mapped to the parameters of the corresponding BACnet object access service primitives. The mapping in detail looks like the following:

- OPC UA `Read` service request → BACnet `ReadProperty` service request:
  - The BACnet `Object_Identifier` and the `Property_Identifier` are determined by the OPC UA server using the information model and the current element (a `nodeId`) of the `nodesToRead[]` parameter of the OPC UA `Read service` request. The network address of the target device is determined based on the `Device_ObjectID`.
  - The `attributeId` parameter is checked by the server and only if it is set to `Value` the BACnet `ReadProperty` service request is generated. Otherwise, the

desired attribute is read from the information model and passed as an argument to the OPC UA `Read` service response.

- BACnet `ReadProperty` service response → OPC UA `Read` service response:
  - The current element of the `result[]` parameter of the OPC UA `Read` service response is set to the `Property Value` parameter of the BACnet `ReadProperty` service response. This is only done if the latter service was successful.
  - The current element of the `diagnosticInfos[]` parameter of the OPC UA `Read` service response is set to `Good` in case the desired operation was successful. Otherwise it is set to `Bad`.
- OPC UA `Write` service request → BACnet `WriteProperty` service request:
  - The current element (the `nodeId`) of the `nodesToWrite[]` parameter of the OPC UA `Write` service request is translated to a BACnet `Object_Identifier` and a `Property_Identifier` by the OPC UA server using the information model. The network address of the target device is determined using the `Device_ObjectID`.
  - The `attributeId` parameter is checked by the server and only if it is set to `Value` the BACnet `WriteProperty` service request is generated. Otherwise, the write operation is performed on the node in the information model only.
  - The `value` parameter of the OPC UA `Write` service request is passed to the `Property Value` of the BACnet `WriteProperty` service request. This is only done if the `attributeId` is set to `Value`.
  - The `Priority` parameter of the `WriteProperty` service request is set according to the value of the `BACnetPriority` property referenced by the `Present_Value` node of the current OPC UA object.
- BACnet `WriteProperty` service response → OPC UA `Write` service response:
  - If the BACnet `WriteProperty` service response indicates `Result(+)`, the current element of the `result[]` parameter of the OPC UA `Write` service response is set to `Good`, otherwise to `Bad`.
  - The current element of the `diagnosticInfos[]` is set to `Success` if the `WriteProperty` service returns with `Result(+)`. Else, this parameter is set to `Failed`.

Figure 4.8 shows the procedures of reading and writing a datapoint of the process image by an OPC UA client via an OPC UA server. It is assumed that a session is already established between the OPC UA server and client and that the driver has already performed a discovery on the BACnet network. For clearance, the sequence diagram shows only application layer related parameters. Network layer parameters (IP addresses) are omitted.

OPC UA Subscription Services and BACnet COV Subscription Services

As mentioned in Section 2.5, there are two Change of Value Subscription services in BACnet, the `SubscribeCOV` service which subscribes on a predefined set of properties (for the majority of of BACnet objects on the `Present_Value` and the `Status_Flags` property) as well as the

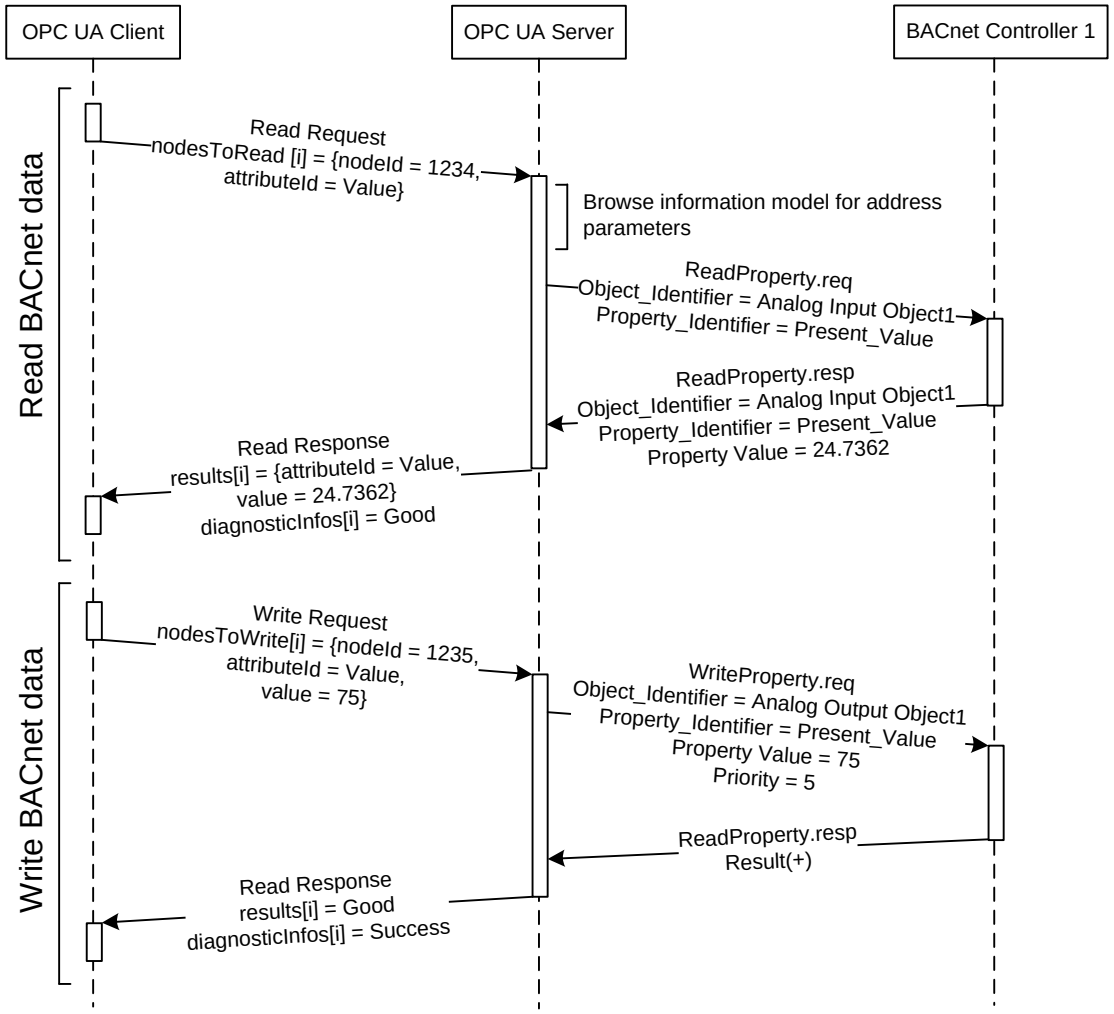

Figure 4.8: OPC UA client accessing BACnet data via an OPC UA server

more general SubscribeCOVProperty service where the set of properties can be passed as an argument. Depending on the application, one of these services can be chosen. Since more freedom is given by the SubscribeCOVProperty service, this one is considered more suitable for this mapping. Following this approach, the OPC UA client has complete freedom in selecting the datapoints, where the subscription shall be applied on.

Besides the value of a variable, there are also two other possible sources of information for a MonitoredItem in OPC UA, EventNotifiers and aggregated variable values. However, these fulfil other purposes than the BACnet COV services. So it can be stated that a subscription on a MonitoredItem with a variable value as a source of information in OPC UA has a similar semantics than applying a SubscribeCOVProperty on a BACnet property.

The procedure of an OPC UA client applying a subscription on a MonitoredItem is explained in Section 3.5, where also a use case example thereof is given. In the second step of this sequence, when a MonitoredItem is created on the server as a consequence of the CreateMonitoredItem call of the client, the parameters of the BACnet SubscribeCOV-Property request need to be set before it can be issued to the underlying BACnet network. The address data needed therefore is again gained from the information model.

The opposite use case of applying a subscription on a MonitoredItem is to remove the latter from its subscription and delete it as well. Deleting a MonitoredItem must consequently result in cancellation of the BACnet COV subscription. There is no explicit service in BACnet to achieve this, but it is done by calling the SubscribeCOVProperty service where the IssueConfirmedNotifications and the Lifetime parameters are omitted.

The mapping of CreateMonitoredItem request parameters to SubscribeCOVProperty request arguments in detail looks like the following:

- OPC UA CreateMonitoredItem request → BACnet SubscribeCOVProperty request:
  - The Subscriber Process Identifier parameter of the SubscribeCOV-Property request is set to the nodeId value gained from the current element of the itemsToCreate[] argument of the CreateMonitoredItem request. This results in a unique assignment of a BACnet COV subscription to an OPC UA variable, assumed there is only one OPC UA server interfacing the BACnet network. If two or more servers are present, it has to be taken care that they use different ranges of nodeIds, i.e. by using different name spaces. Deploying two or more servers modelling different BACnet properties with the same nodeId results in wrong assignment of COVNotifications to OPC UA variables by the servers.
  - The Monitored Object Identifier parameter of the SubscribeCOV-Property request shall be set to the value of the Object_Identifier of the object that holds the property of interest. This value can be read from the Object_Identifier variable exposed by the object in the server address space.
  - The Issue Confirmed Notifications flag can be set to TRUE if the OPC UA server is desired to confirm the received COVNotifications. Otherwise, it has to be set to FALSE. This is an implementation dependent setting.

  - The `Lifetime` parameter indicates how long the subscription shall be active. In this model, a default value which can be chosen depending on the application scenario is taken. As future work, this parameter could be integrated in the information model.
  - The `Monitored PropertyIdentifier` specifies which property to subscribe on. Its value is also gained from the information model and is set to the value of the `BACnetPropertyID` property referenced by the variable of interest.

- BACnet `SubscribeCOVProperty` response → OPC UA `CreateMonitoredItem` response:
  - In case the `SubscribeCOVProperty` response returns with `Result(+)`, the current element of the `result[]` parameter of the OPC UA `CreateMonitoredItem` response is set to `Good`, otherwise to `Bad`.
  - The current element of the `diagnosticInfos[]` parameter of the `CreateMonitoredItem` response is set to `Success` if the `SubscribeCOVProperty` response indicates `Result(+)`. Else, this parameter is set to `Failed`.

- OPC UA `DeleteMonitoredItem` request → BACnet `SubscribeCOVProperty` request:
  - The `Subscriber Process Identifier` parameter of the `SubscribeCOVProperty` request is set to the `nodeId` value gained from the current element of the `itemsToCreate[]` argument of the `DeleteMonitoredItem` request. The same restrictions valid for the `CreateMonitoredItem` and the `SubscribeCOVProperty` with respect to uniqueness of the assignment apply to this service mapping.
  - The `Monitored Object Identifier` argument of the `SubscribeCOVProperty` request is set to the `Object_Identifier` value of the object that holds the particular property. This value is read from the `Object_Identifier` variable of the object in the server address space.
  - The `IssueConfirmedNotifications` parameter as well as the `Lifetime` parameter are omitted in order to achieve the cancellation of the COV subscription.
  - The `Monitored PropertyIdentifier` is set to the `BACnetPropertyID` value referenced by the variable of interest. The `BACnetPropertyID` value is gained from the information model.

- BACnet `SubscribeCOVProperty` response → OPC UA `DeleteMonitoredItem` response:
  - In case the `SubscribeCOVProperty` response returns with `Result(+)`, the current element of the `result[]` parameter of the OPC UA `CreateMonitoredItem` response is set to `Good`, otherwise to `Bad`.
  - The current element of the `diagnosticInfos[]` parameter of the `CreateMonitoredItem` response is set to `Success` if the `SubscribeCOVProperty` response indicates `Result(+)`. Else, this parameter is set to `Failed`.

In case, a BACnet device issues a `COVNotification` which is received by the OPC UA server, the server has to change the internal value of the particular variable. Since the `COVNotification` contains the `Subscriber Process Identifier` which has been set to the value of the `nodeId` of the variable, correct assignment is achieved (network-wide uniqueness of `nodeIds` provided). The new value is propagated to the OPC UA clients which have applied a subscription on this variable during the next publishing interval.

It is also possible to update a subscription (i.e. change its parameters) on a `MonitoredItem` by an OPC UA client. The `ModifyMonitoredItems` service fulfills this purpose. A call of this shall result in the server issuing the `SubscribeCOVProperty` service and passing the new parameters. This procedure was not in the focus of this work. However, the parameter mapping is similar to `CreateMonitoredItem` and `SubscribeCOVProperty`.

Figure 4.9 shows a typical procedure of an OPC UA client applying a subscription on a node representing a BACnet property. As a consequence, the OPC UA server issues a `SubscribeCOVProperty Request` to the proper BACnet controller. Incoming `COVNotifications` from the controller trigger the OPC UA server to issue `Publish Response` messages on the OPC UA client's `Publish Requests`. These `Publish Responses` contain the new value of the BACnet property that has changed.

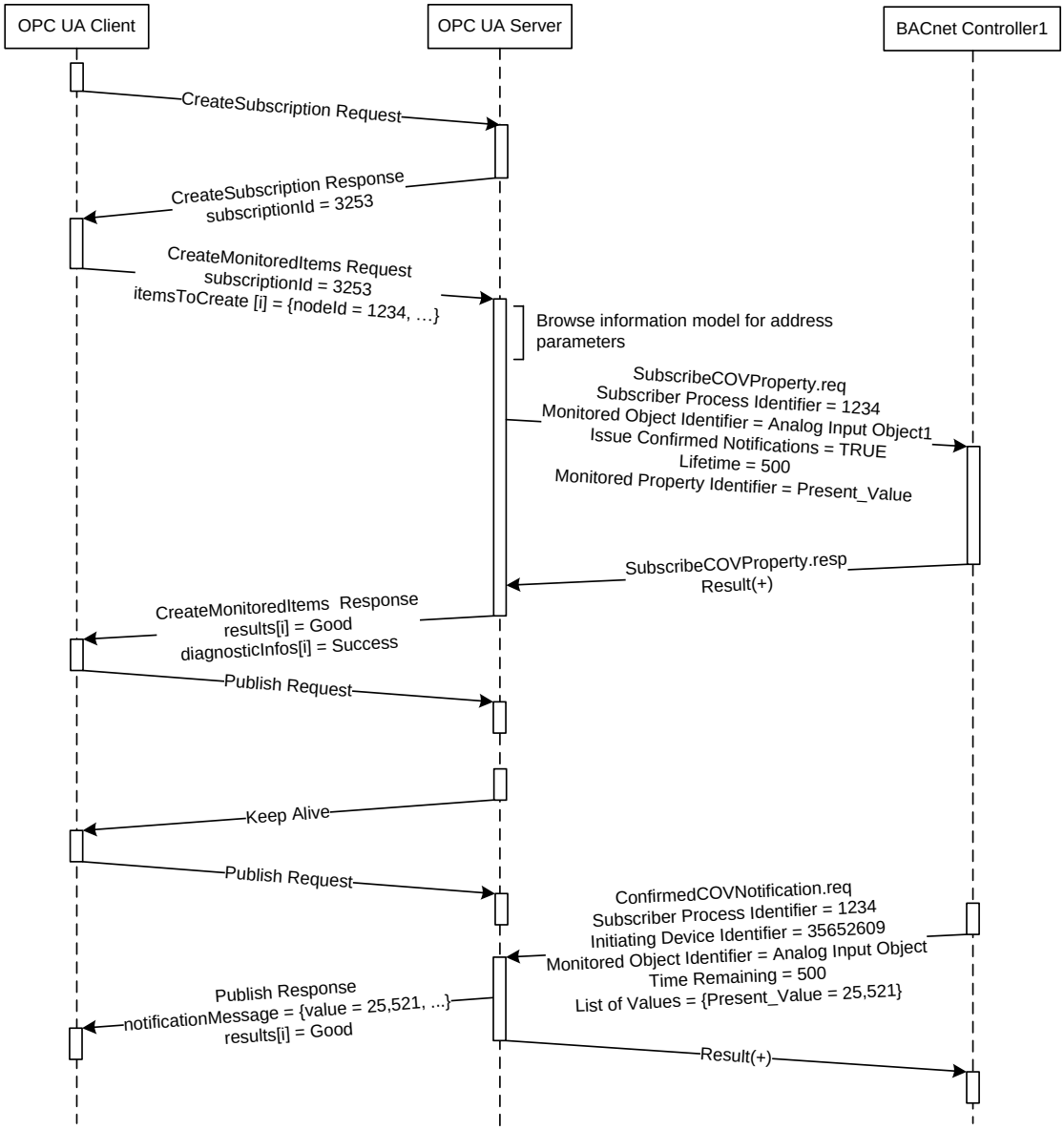

Figure 4.9: OPC UA client applying a subscription to a BACnet property via an OPC UA server

CHAPTER 5

Implementation of an OPC UA Server for BACnet

To evaluate the information model developed for BACnet, a proof-of-concept implementation has been written. The implementation uses parts of the *Comet Automation Toolkit* which was developed in the context of the EraSME project "Web-based Communication in Automation (WebCom)"[1] by the project partner HB-Softsolution[2]. The whole toolkit is written in Java and provides platform-independency. One of its parts is the *Comet Model Designer*, a graphical editor for XML-based OPC UA information models. Another one is the *Comet UA Server SDK*, a framework which was used to implement an OPC UA server being able to interface BACnet/IP networks.

5.1 Comet UA Model Designer

The *Comet UA Model Designer* was taken in the course of this work to generate an XML representation of the BACnet information model. As an editing tool, the Model Designer can be used to build up information models and to extend existing ones. It provides a graphical user interface that supports the user in applying definitions of data types, variable types, reference types, and object types. To achieve this functionality, various input elements like text fields, checkboxes and drop-down menus are available for defining the particular attribute values. Furthermore, instances may be derived from the previously created type definitions in a very comfortable way. The hierarchical structure of the resulting information model is expressed by a tree view. A screenshot of the Model Designer user interface (Figure 5.1) shows the definition of the BACnet `Lighting Output Object` (without completeness of properties) embedded in its type hierarchy. The information model created by the Comet Model Designer is finally exported to

[1] http://www.webcom-eu.org/
[2] http://www.hb-softsolution.com/

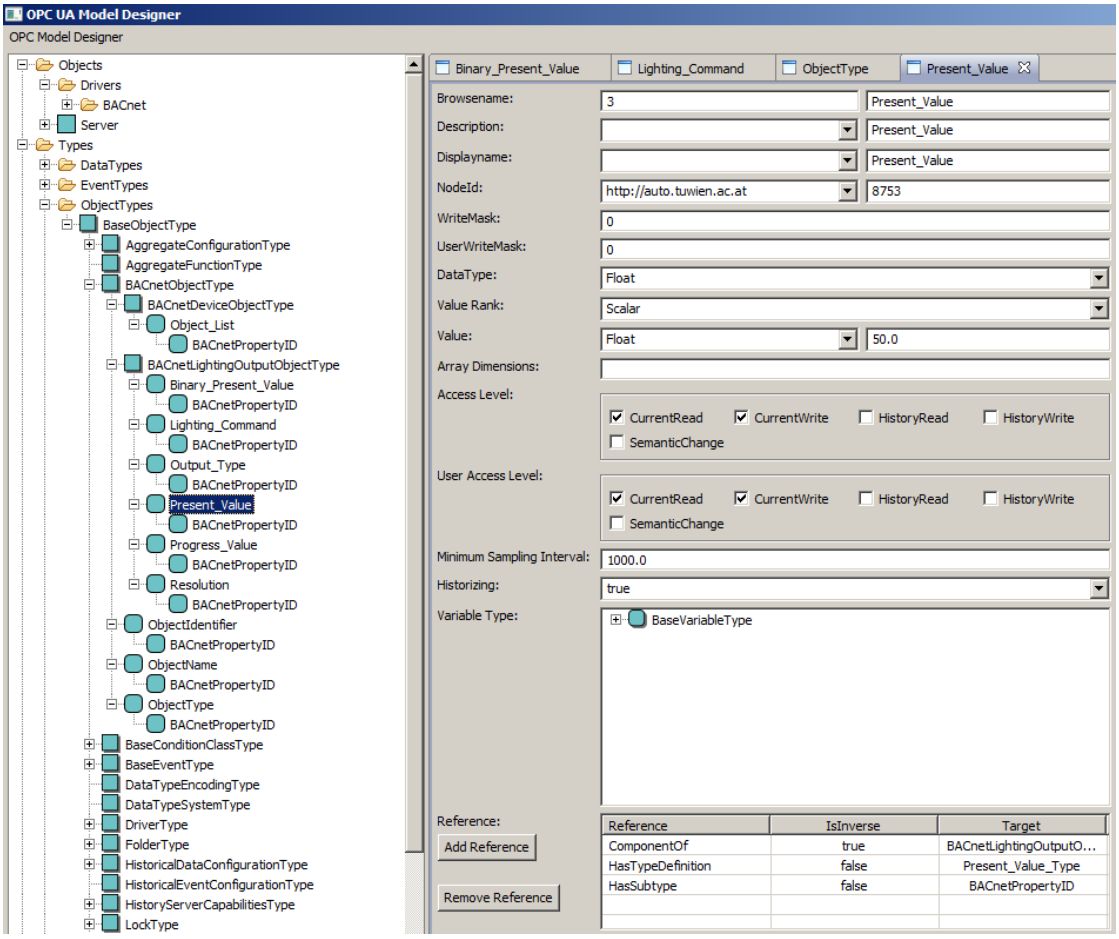

Figure 5.1: OPC UA Model Designer

an XML file. This file can be opened again by the Model Designer for further editing. Alternatively, it can be loaded by the Comet OPC UA Server. The format of these XML files follows the XML Schema Definition published by the OPC foundation [8].

The following listing shows an extract of the XML file representing the BACnet information model developed. It is the beginning of the sequence describing an OPC UA object which stands for a BACnet `Analog Input Object`:

```
<Node i:type="ObjectNode">
      <NodeId>
        <Identifier>ns=4;i=17093</Identifier>
      </NodeId>
      <NodeClass>Object_1</NodeClass>
      <BrowseName>
        <NamespaceIndex>0</NamespaceIndex>
        <Name>AnalogInputObject1</Name>
      </BrowseName>
      <DisplayName>
        <Locale>en</Locale>
        <Text>AnalogInputObject1</Text>
      </DisplayName>
      <Description>
        <Locale>en</Locale>
        <Text>BACnet Analog Input Object 1</Text>
      </Description>
      ...
</Node>
```

The attributes of the OPC UA objects are modelled as XML elements. The `NodeId` element, for instance, starts with a `<NodeId>` tag and ends with a `</NodeId>` tag. Its content in between consists of another element, the `<Identifier>ns=4;i=17093</Identifier>` where the namespace index of the node (`ns=4`) and the `nodeId` itself (`i=17093`) are defined. Other attributes that can be seen in this example are the `BrowseName`, the `DisplayName` and the `Description`.

5.2 Comet UA Server SDK

Another major part of the Comet Automation Toolkit is a Software Development Kit (SDK) for implementing Java based OPC UA servers. This server SDK is functionally separated into two parts: one is the core OPC UA server which is based on the OPC UA Java stack released by the OPC foundation. The second part of the server module consists of a driver framework which allows to implement drivers (interfaces) for particular network technologies that can be loaded into the core server.

Both the server and the client SDKs are available for development in the widely used *Eclipse*[3] IDE. For each SDK, a Wizard is provided that facilitates setting up a project.

Figure 5.2 shows the overall architecture of the Comet Server SDK. The central part is the Core Server based on the OPC UA Java stack published by the OPC foundation. It provides several interfaces:

- A network interface accepting connections from OPC UA clients via the backbone network. Here the incoming requests from the clients are handled and responses are sent back.
- A bidirectional software interface passing the requests of the client downwards to the technology specific driver based on the Driver SDK on the one hand, and providing the driver access to the OPC UA address space on the other hand.
- A file handler loading the information models in XML format. Via an XML parser, these files are read and the server address space is built up based on these data.

[3] www.eclipse.org

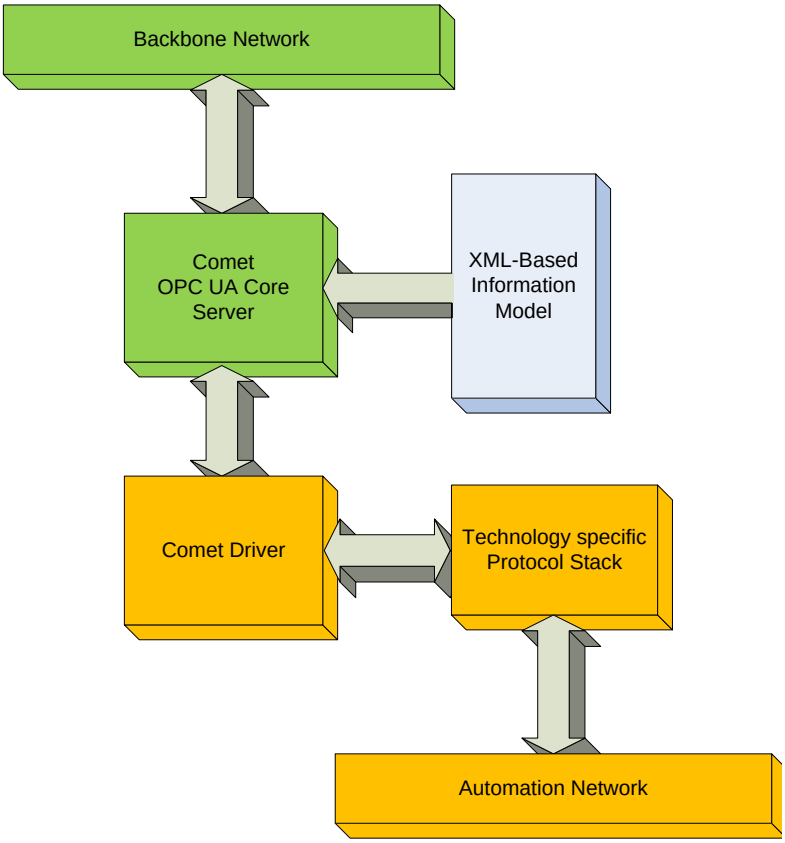

Figure 5.2: Comet OPC UA Server Architecture

The driver package communicates with the Core Server and interfaces the automation network specific protocol stack. Therefore, methods are provided that pass requests from the server to the protocol stack. To enable the automation network stack to access the server address space, there are also methods provided to fulfil this purpose. This way, address information regarding the underlying automation network can be gained and value changes in the process image can be propagated to the address space.

In order to abstract the access to the target network, a package implementing a specific protocol stack can be loaded. This package provides methods for issuing service calls and also handlers to deal with requests originating from the underlying network.

Comet OPC UA Core Server

The Core Server embodies a Java implementation of the OPC UA standard. It also supports a generic address space which means that it can load the standard OPC UA information model plus any user-defined information models out of one or more XML files. Furthermore, this address space can be considered dynamic. This means that changes (in the sense of adding and removing nodes) can be applied during runtime.

Server specific configuration parameters are loaded from another XML file. This way, the configuration is completely isolated from the server's source code.

Due to the use of Java, the Comet OPC UA Server is completely platform-independent, which is one of its key features. Another one is the capability to interface even multiple technology specific drivers. So it is able to provide a unified view to a physical process, where different network technologies are put in operation. This is achieved by each driver interface communicating with one specific network technology. Also multiple clients using multiple security profiles can connect to one server instance. One client is allowed to apply multiple subscriptions on nodes in the address space, too. These features make the server to very flexible and scalable tool.

Currently, the core server implements the following OPC UA *Facets* out of the *Profile List* defined in the OPC UA standard (cf. [19] Part 7):

- Base Server Behaviour Facet
- Basic DataChange Subscription Server Facet
- Core Server Facet
- Data Access Server Facet
- Enhanced DataChange Subscription Server Facet
- Method Server Facet
- Node Management Server Facet
- Standard UA Server

In the following, the folder structure of the Comet Server framework is illustrated. Figure 5.3 shows a screenshot of the package view provided by the *Eclipse* IDE.

- The main method is located in the `BACnetServerApp.java` file in the `src` folder. Here, a server instance is created and the server configuration is loaded (the content of this XML file is described beneath the item dedicated to the `serverconfig` folder):

  ```
  opcServer.loadServerConfiguration("serverconfig/serverconfig.xml");
  ```

  The information model is loaded from one or more XML files. The `nodeset.xml` denotes the standard OPC UA information model, whereas the `BACnet.xml` represents the BACnet specific part:

  ```
  opcServer.loadModelFile("InformationModel/nodeset.xml");
  opcServer.loadModelFile("InformationModel/BACnet.xml");
  ```

- The `Referenced Libraries` folder in Figure 5.3 exposes the jar archives including the class files of the server framework. They are all included in the server release, except the driver specific libraries that are also placed here. An example of such a driver library is the `Comet_BACnet.jar` which represents the BACnet driver. This package was developed in the course of this thesis (cf. next subsection).

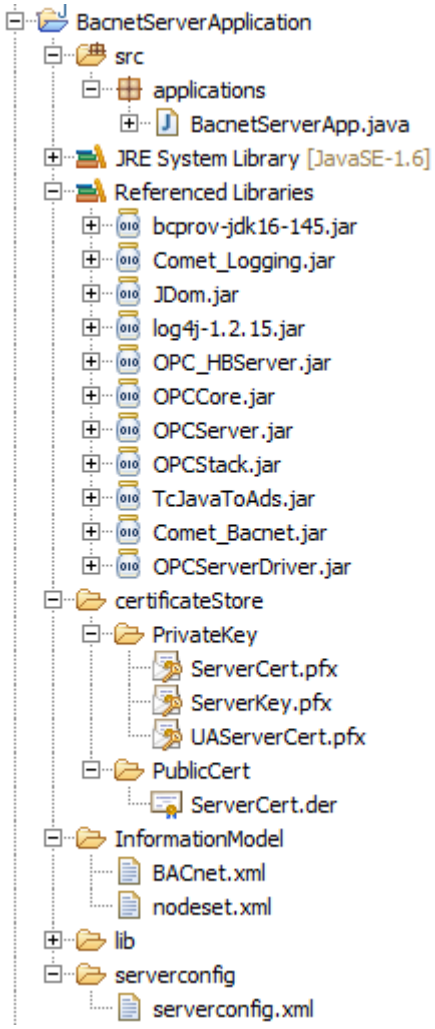

Figure 5.3: Comet Server Folder Structure

- The `certificateStore` folder contains the server certificates and the private key file. An introduction into operating an OPC UA server in a trusted environment and also into certificate management is given in Section A.4.
- Information model files in XML format must be located in the `InformationModel` folder. The XML schema these file have to follow is specified in an XML Schema Definition (XSD) file published by the OPC foundation (cf. [8]).
- The `lib` folder actually contains the jar archives which are shown beneath the `ReferencedLibraries`.
- The `serverconfig` folder contains the `serverconfig.xml` where various settings regarding the core server can be applied. Following the XML tags, it is straight forward to

find the fields where vendor information, security configuration, transport quotas, and supported security policies can be set. Notice that the local address of the network interface via which an OPC UA client connects to the server must be set as follows:

```
<BaseAddresses>
  <ua:String>opc.tcp://172.20.0.51:6006</ua:String>
</BaseAddresses>
```

In this example, the network address is set to 172.20.0.51 and the server listens on port 6006.

Comet Driver Framework

Drivers interfaced by the Comet Core server are responsible for providing the connections to the protocol stacks of the underlying networks. Depending on their technologies, the stack implementations can freely be chosen. In order to uniquely assign the nodes of a specific information model to a distinct driver implementation, the *Namespace Uniform Resource Identifier (URI)*, which is configured via the `DRIVER_NAMESPACE` member variable of the `CometDRVManager` class, must be equal to the namespace URI attribute of each node of the information model. This way, the server is able to distinguish between specific driver implementations based on the namespace URI of the node currently accessed. This is especially useful if more than one driver implementation is loaded by the server.

Figure 5.4 shows the folder structure of the driver framework. The source (`src`) folder contains the `ICometDriverConnection.java` and the `CometDRVManager.java` which are delivered with the driver framework. The interface methods provided by the driver framework and which are responsible for communication and data exchange with the stack implementations of the underlying network protocols are located in the `CometDRVManager.java`. The following methods are available:

- `prepareRead(NodeId nodeId)` is called after the server has received a `Read` request from a client and before `syncReadValue(NodeId, long senderState)` is called. This method is used to determine if a node of the server address space that is currently accessed by a client is a representation of a datapoint of the process image (i.e. if this access needs to be handled by the driver or not). The result of this decision is internally saved by the server by setting a flag to `SYNCREAD` or `NOREAD`, respectively.
- `prepareWrite(NodeId nodeId)` fulfills the same purpose as `prepareRead(NodeId)` but is used for `Write` requests.
- `syncReadValue(NodeId nodeId, long senderState)` is called after a client has sent a `Read` request to the server and the node needs to be accessed by the driver. This method takes the `NodeId` of the particular node and returns the value of the datapoint determined by the stack implementation.
- `syncWriteValue(NodeId nodeId, DataValue, long senderState)` has the same functionality as `syncReadValue(NodeId, long)` but for write accesses. Additionally, the value of the datapoint to be written must be provided as an argument.

- `registerNotification(Node node, MonitoredItemCreateRequest)` is called by the server if a client applies a subscription on a `MonitoredItem` with a variable node of the process image as data source.
- `unregisterNotification(NodeId nodeId)` shall inform the underlying system that a cancellation of a subscription on a `MonitoredItem` has been requested by the client. This cancellation request is propagated to the subscription mechanism of the underlying system by this method.
- `writeFromDriver(NodeId nodeId, DataValue value, long dpState)` must be called by the driver if a client has applied a subscription on a `MonitoredItem` with a variable node of the process image as data source and if the driver receives a notification from the underlying network that the value of a datapoint has changed. The new value must be passed to this method which propagates it to the server address space. The server again publishes the new value to the particular client which has applied the subscription.
- `readValue(NodeId nodeId)` can be called by the driver to read the value of a node of the server address space.

The jar archives exposed by the `Referenced Libraries` folder in Figure 5.4 are part of the driver framework release, except the `bacnet4j.jar` and the `seroUtils.jar`. These two files are specific for the implementation of the BACnet driver introduced below. The first one contains the BACnet/IP Java stack (BACnet4J, cf. Section 5.3), the second one is a dependency of BACnet4J.

The libraries described above and just logically linked to the `Referenced Libraries` folder are actually located in the `lib` folder which can be seen at the bottom of Figure 5.4.

In order to use the driver with the Comet OPC UA Core Server, the whole folder structure of the driver framework described above must be packed into a jar archive and put into the `lib` folder of the server framework. Packing is done either by hand or more conveniently by the Export wizard ($File \rightarrow Export... \rightarrow Java \rightarrow Jar$) of the Eclipse IDE.

5.3 BACnet Driver implementation

This section describes the implementation of a BACnet specific driver interfaced by the Core Server. The Driver Module contains methods to initialise network communication, discover devices present in the network and gather the necessary data from the information model to address a datapoint in the BACnet network. There are also methods for read and write access and for applying a subscription on specific datapoints. All the code sections described below are located in the `CometDRVManager.java`.

BACnet/IP Stack for Java

The driver implementation for the required interface to the BACnet/IP network is based on the open source *BACnet/IP for Java* stack[4], also called *BACnet4J*. It is a Java implementation of

[4] http://bacnet4j.sourceforge.net/

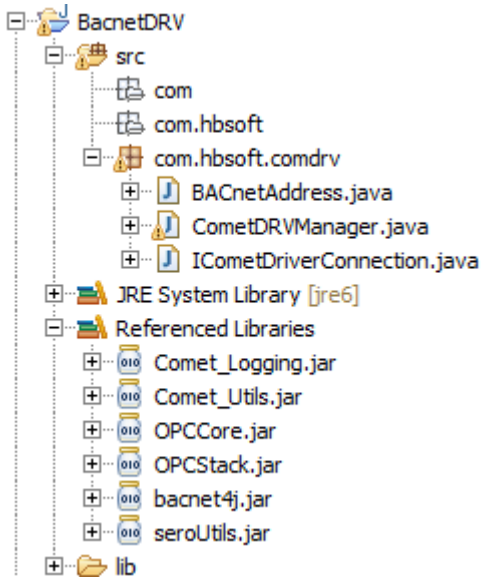

Figure 5.4: BACnet Driver Folder Structure

the BACnet/IP protocol supporting all BACnet services defined in the current specification [20]. Emulating a BACnet device by instantiating local BACnet objects is also a task of this implementation. BACnet objects can be individually created by combining the desired properties to a custom object. However, in the context of this work, the BACnet stack is only used to implement client functionalities, besides the instantiation of a local `DeviceObject`, which is mandatory for each BACnet device in a network.

There exists a rudimentary documentation in form of a Javadoc[5] (an HTML based documentation of the API) which is provided on the Sourceforge BACnet4J webpage. Example programs for basic applications are also included in the `test` subfolder within the package BACnet4J is delivered. Support is given via the forum[6] of *Serotonin Software*, the developer of BACnet4J.

BACnet4J is available as a jar archive. Notice that a classpath entry must be set for `seroUtils.jar` (also available via the Sourceforge project webpage) as well when using `bacnet4J.jar` [1].

Initialisation

When the server is started, the BACnet4J stack is initialised by assigning the (IP) address of the correct network interface and also assigning the device identifier to the local device. The driver is this way behaving like a BACnet device in the network exposing a `Device Object`. This is done by instantiating a new `LocalDevice`:

```
BACnetLocalDevice = new LocalDevice(13579, BACNET_BROADCAST_ADDRESS, BACNET_LOCAL_ADDRESS);
```

[5] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[6] http://mango.serotoninsoftware.com/forum/forums/show/12.page

A new event listener (the `Listener` class which is inherited from the `DefaultDeviceEventListener` must be defined) to process incoming service responses is added to the event handler of the local device:

```
BACnetLocalDevice.getEventHandler().addListener(new Listener());
```

Then the `Local Device` is initialised:

```
BACnetLocalDevice.initialize();
```

After this a `Who-Is` broadcast is performed to the network to determine which devices are present. This is done by the following statements:

```
InetSocketAddress addr = new InetSocketAddress(InetAddress.getByName(BACNET_BROADCAST_ADDRESS),
47808);
BACnetLocalDevice.sendUnconfirmed(addr, null, new WhoIsRequest());
```

The replies (`I-Am` messages) which provide the assignment of the device identifiers to the IP addresses are internally stored by the BACnet4J stack. Additionally the `Object_List` property of each device is read and printed to the local console.

Obtaining address information of a BACnet property node

A method that is called whenever an access to a node is performed is the `public BACnetAddress getBACnetAddress(NodeId)`, which determines if the node represents a BACnet datapoint. If so, it reads the components of a BACnet address (`PropertyIdentifier`, `ObjectIdentifier` and `DeviceIdentifier`) from the server address space and returns an object of the `BACnetAddress` class containing these parameters. The server address space is based on the information models loaded and contains an image of all its nodes including their attributes. The algorithm of the `getBACnetAddress(NodeId)` method, which is illustrated in form of a flowchart depicted in Figure 5.5, works the following way:

In a first step, all references linked to the start node, where the `nodeId` is passed to the method as an argument, are saved to an array. Since the node holding the property identifier is an OPC UA property of the start node, the array is searched for a reference pointing to a node with the browse name attribute set to `Property_Identifier`. If it is found, this reference is followed and the value of the target node is read and saved. The same array is also searched for an inverse reference with the browse name `HasBACnetProperty`. This one should point to the BACnet object node which holds all the BACnet properties. Again, an array containing the references linked to the object node is created. This array is searched for a reference pointing to a node with the browse name `Object_Identifier` and for an inverse reference with the browse name attribute set to `HasBACnetObject`. If the `Object_Identifier` node is found, its value is saved. If the `HasBACnetObject` reference is found, it is followed, to the object node representing the BACnet device. Again an array of its references is saved. This array is searched for a reference pointing to a node with the browse name `DeviceObject`. The same procedure of creating an array of its references is applied for this node. Again, the resulting array is searched for a reference pointing to a node with the browse name `Object_Identifier`. The target node holds the `DeviceObject_Identifier`, which value is also saved.

If all three components of the BACnet address have been found, an object of the `BACnetAddress` class containing these components is instantiated and returned by this method, otherwise `null` is returned.

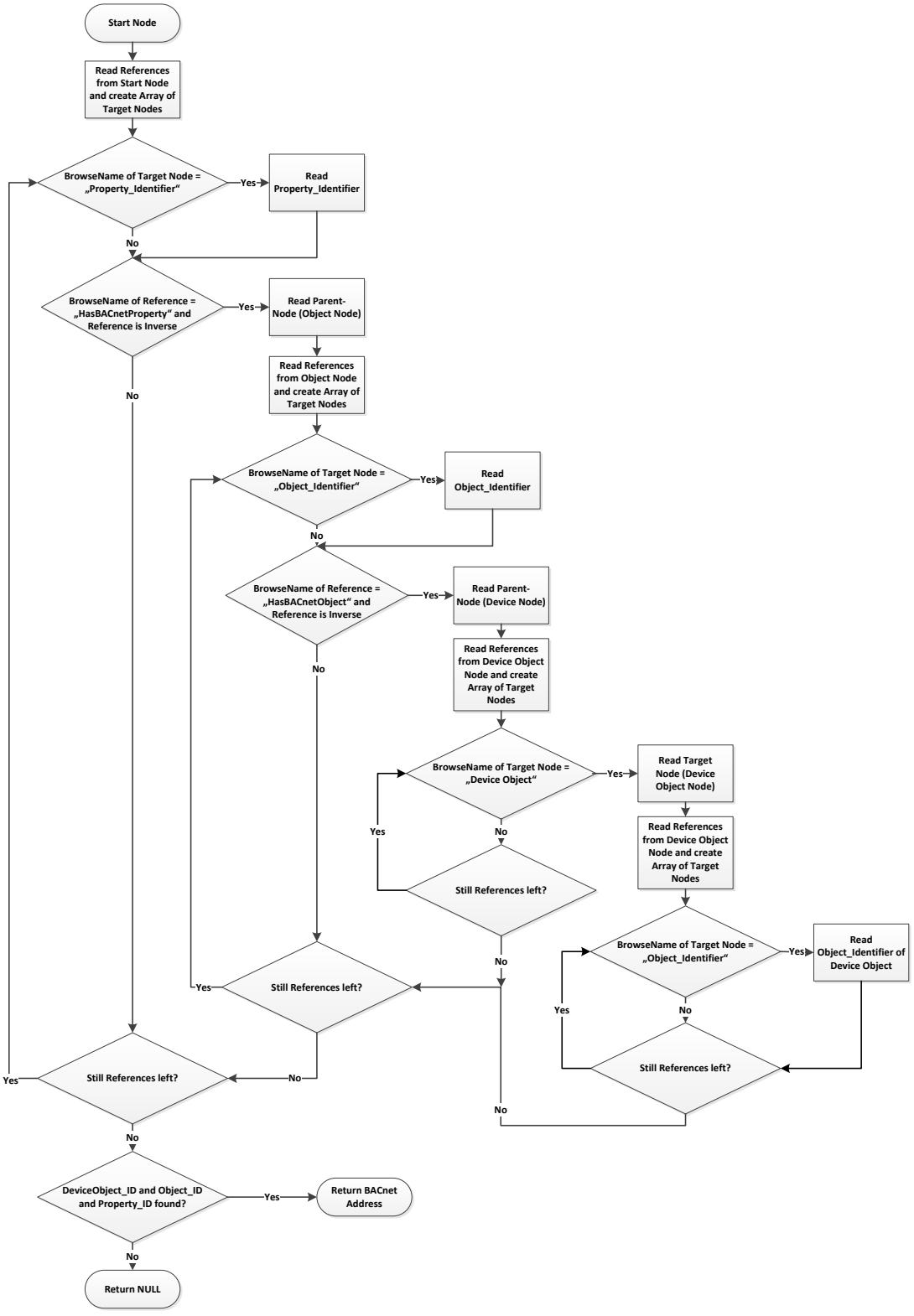

Figure 5.5: Flowchart of the Information Model Browse Routine

Read/Write Access

If an OPC UA Client browses to a variable of a BACnet object and performs a read or write request to its value, the server first calls the `prepareRead(NodeId nodeId)` method or the `prepareWrite(NodeId nodeId)` method, respectively. In these methods, the `getBACnetAddress(NodeId)` is called to determine if the particular node is a BACnet datapoint or not. According to this decision, an internal server flag is set to indicate the nodes affiliation. If the node is a variable representing a BACnet property, the server calls the `syncReadValue(NodeId nodeId, long senderState)` method or the `syncWriteValue(NodeId nodeId, DataValue, long senderState)` method, respectively.

To avoid recursive calls of the `syncReadValue(NodeId nodeId, long senderState)` method, there is a boolean member variable `DriverAccessLock`, which is checked for being true in the beginning of the method execution. If so, the method is immediately exited. Otherwise, `DriverAccessLock` is set to true and the method is executed. Without this locking mechanism it would be also executed when reading a node value by the `getBACnetAddress(NodeId)` since the server has, in case `prepareRead(NodeId nodeId)` has not been called for this specific `nodeId` and the affiliation flag has not been set, no knowledge about the node representing a BACnet property or not.

Before accessing the BACnet network, the proper BACnet4j `RemoteDevice` has to be found in the internal data structure containing the (IP) addresses of the network devices assigned to the `Device_Identifier`. As mentioned, this data structure has been created at initialisation time after scanning the BACnet network.

The following code section shows how this procedure is defined for read accesses:

```
public synchronized ReadResponse syncReadValue(NodeId nodeId, long senderState) {
if(DriverAccessLock == true){
  return null;
}
DriverAccessLock = true;
DataValue[] PropertyValues = null;
DiagnosticInfo[] DInfo = new DiagnosticInfo[1];
System.out.println("Read from node: " + nodeId.toString());
try{
  currentBACnetAddress = getBACnetAddress(nodeId);
  if(currentBACnetAddress != null){
    //find proper BACnet device
    for(RemoteDevice BACnetRemoteDevice : BACnetLocalDevice.getRemoteDevices()){
      if(BACnetRemoteDevice.getInstanceNumber() == currentBACnetAddress.getDeviceObjectID()){
        System.out.println("RemoteDevice: " + BACnetRemoteDevice.getInstanceNumber());
        //read BACnet property
        PropertyValues = ReadBACnetProperty(BACnetRemoteDevice,
        currentBACnetAddress.getObjectIdentifier(), currentBACnetAddress.getPropertyIdentifier());
        if(PropertyValues == null){
          DriverAccessLock = false;
          return null;
        }
        DInfo[0] = new DiagnosticInfo("Read from BACnet successful", null, null, 0, 0,
        nameSpaceIndex, 0);
        DriverAccessLock = false;
        return new ReadResponse(new ResponseHeader(DateTime.currentTime(), new UnsignedInteger(0),
          StatusCode.GOOD, DInfo[0], null, null), PropertyValues, DInfo);
    }
  }
}
catch(Exception e){
  e.printStackTrace();
  DInfo[0] = new DiagnosticInfo("Read from BACnet failed", null, null, 0, 0, nameSpaceIndex, 0);
  DriverAccessLock = false;
  return new ReadResponse(new ResponseHeader(DateTime.currentTime(), new UnsignedInteger(0),
```

```
    StatusCode.BAD, DInfo[0], null, null), PropertyValues, DInfo);
}
DriverAccessLock = false;
return null;
}
```

The `RemoteDevice`, the `Object_Identifier` und the `Property_Identifier` are passed to the `ReadBACnetProperty(RemoteDevice BACnetRemoteDevice, ObjectIdentifier ObjId, PropertyIdentifier PropId)` method to call the read service methods of BACnet4J and to perform the type conversion from BACnet data types to OPC UA data types. An array of `PropertyValues` of the OPC UA datatype `DataValue` is returned.

In case the `syncWriteValue(NodeId nodeId, DataValue, long senderState)` method is called, the type conversion from OPC UA data types to BACnet data types has to be done first. The following listing shows this for OPC UA datatype `Float`:

```
if(value.getValue().getValue() instanceof Float){
  propertyValue = new com.serotonin.bacnet4j.type.primitive.Real(value.getValue().floatValue());
```

After this, the associated `RemoteDevice` needs to be found. This is done the same way like in `syncReadValue(NodeId nodeId, long senderState)`. Then, the `WriteBACnetProperty(NodeId BACnetPropertyNodeId, RemoteDevice BACnetRemoteDevice, ObjectIdentifier ObjId, PropertyIdentifier PropId, Encodable propertyValue)` method is called, if all previous operations have been successful. These operations are shown in the following listing:

```
currentBACnetAddress = getBACnetAddress(nodeId);
if(currentBACnetAddress != null){
  for(RemoteDevice BACnetRemoteDevice : BACnetLocalDevice.getRemoteDevices()){
    //find proper BACnet device by instance number
    if(BACnetRemoteDevice.getInstanceNumber() == currentBACnetAddress.getDeviceObjectID()){
      WriteBACnetProperty(nodeId, BACnetRemoteDevice, currentBACnetAddress.getObjectIdentifier(),
        currentBACnetAddress.getPropertyIdentifier(), propertyValue);
      //read back property value
      Thread.sleep(300);
      propertyValue_rb = ReadBACnetProperty(BACnetRemoteDevice, currentBACnetAddress.getObjectIdentifier(),
        currentBACnetAddress.getPropertyIdentifier());
      break;
    }
  }
}
else{
  //BACnetAddr not found
  DriverAccessLock = false;
  return null;
}
```

The priority parameter of the BACnet `WriteProperty` service is gained from the information model within the `WriteBACnetProperty(NodeId BACnetPropertyNodeId, RemoteDevice BACnetRemoteDevice, ObjectIdentifier ObjId, PropertyIdentifier PropId, Encodable propertyValue)` method. A write access to a BACnet property with a priority set too low results in the value of the property not being changed effectively (for a description of the priority mechanism of write accesses in BACnet cf. Section 2.5). This can result in data inconsistency between the OPC UA server and the BACnet network. Therefore, a check needs to be performed by a subsequent read access (after a short waiting period to cover the delays of transmissions and processing the BACnet controllers take) to the

same property. The result of this check is passed to the server. This is done by OPC UA status codes:

```
if(propertyValue_rb[0].equals(value)){
  results[0] = StatusCode.GOOD;
  DInfo[0] = new DiagnosticInfo("Write to BACnet successful", null, null, 0, 0, nameSpaceIndex, 0);
}
else{
  results[0] = StatusCode.BAD;
  DInfo[0] = new DiagnosticInfo("BACnet Priority", null, null, 0, 0, nameSpaceIndex, 0);
}
```

The `ReadBACnetProperty(RemoteDevice BACnetRemoteDevice, ObjectIdentifier ObjId, PropertyIdentifier PropId)` creates a `PropertyReference` consisting of the `Object_Identifier` and the `Property_Identifier` which were passed as arguments. This in turn is together with the `RemoteDevice` passed to the BACnet4J `readProperties` method which then applies the BACnet `ReadProperty` service to the network.

```
PropertyReferences Pref = new PropertyReferences();
Pref.add(ObjId, PropId);
PropertyValues pvs;
DataValue[] DataValue_inst = new DataValue[READ_BUFFER_SIZE];

try{
  System.out.println("Read BACnetProperty " + PropId.toString() + " of Object " + ObjId.toString());
  //Perform ReadPropertyRequest
  pvs = BACnetLocalDevice.readProperties(BACnetRemoteDevice, Pref);
  System.out.println("Property Value = " + pvs.getString(ObjId, PropId) + "\n");
  Encodable Value = pvs.get(ObjId, PropId);
```

Now the data type of the `Value` variable is determined and a new OPC UA `DataValue` is created which holds the BACnet property value. This is shown for the BACnet `REAL` data type in the following listing:

```
if(Value instanceof com.serotonin.bacnet4j.type.primitive.Real){
  DataValue_inst[0] = new DataValue(new Variant(((Real)Value).floatValue()));
  return DataValue_inst;
}
```

This works analogously for other data types including not only for scalars but also for arrays.

The `WriteBACnetProperty(NodeId BACnetPropertyNodeId, RemoteDevice BACnetRemoteDevice, ObjectIdentifier ObjId, PropertyIdentifier PropId, Encodable propertyValue)` method additionally takes the `NodeId` of the BACnet property node in the server address space as an argument, since it has to determine the `BACnetPriority` thereof. This is done by checking if the BrowseName of the nodes referenced by the BACnet property node equals "`BACnetPriority`" and if true reading the value:

```
//set lowest priority by default
int Priority = 16;
Node BACnetPropertyNode = this.getNode(BACnetPropertyNodeId);
//Get priority from priority property
for(ReferenceNode Reference : BACnetPropertyNode.getReferences()){
  NodeId PriorityNodeId = NodeId.get(Reference.getTargetId().getIdType(), nameSpaceIndex,
    Reference.getTargetId().getValue());
  if(this.getNode(PriorityNodeId).getBrowseName().getName().equals("BACnetPriority")){
    Priority = readValue(PriorityNodeId).getResults()[0].getValue().intValue();
    System.out.println("Priority = " + Priority);
  }
```

```
}
System.out.println("Write BACnetProperty " + PropId + " of Object " + ObjId.toString() +
  ", Value = " + propertyValue.toString());
//Send WritePropertyRequest
try{
  BACnetLocalDevice.send(BACnetRemoteDevice, new WritePropertyRequest(ObjId, PropId, null,
  propertyValue, new com.serotonin.bacnet4j.type.primitive.UnsignedInteger(Priority)));
}
catch(Exception e){
  throw e;
}
```

Finally, the BACnet `WritePropery` service is generated by the `send(RemoteDevice d, ConfirmedRequestService serviceRequest)` method of the BACnet4J class `LocalDevice`.

Change Of Value Subscription

Following the mapping of the BACnet `SubscribeCOVProperty` service to the OPC UA concept of `Subscriptions` on `MonitoredItems` described in Section 4.5, the OPC UA server shall propergate an incoming subscription request of a client to the BACnet driver, which in turn is intended to apply a `SubscribeCOVProperty` service request to the desired BACnet property via the BACnet network. If the value of the particular property changes and a `COVNotification` is generated and received by the BACnet driver, this event shall be propagated to the server. The server publishes this information by transmitting a `Notification` service to the client having requested the subscription.

After receiving a subscription request from a client, the OPC UA server calls the `registerNotification(Node node, MonitoredItemCreateRequest)` interface method. The second argument, the `MonitoredItemCreateRequest` provides, besides the `nodeId` and the `attributeId`, additional parameters describing the monitoring of the data source which it is applied on. Since the `node` (and hereby the required `nodeId`) is passed as an extra argument of the `registerNotification(Node node, MonitoredItemCreateRequest)` method, the attribute to monitor is implicitly the `Value`. Further parameters are currently not regarded. The additional information delivered by the `MonitoredItemCreateRequest` is currently not further processed.

The body of `registerNotification(Node node, MonitoredItemCreateRequest)` looks like the following:

```
return apply_COVSubscription(node.getNodeId(), COVLifetime.default_time);
```

The `unregisterNotification(NodeId nodeId)` method, which is called if the server receives a cancel request on an existing subscription, looks very similar:

```
return apply_COVSubscription(nodeId, COVLifetime.cancel);
```

So the main part of the work is done in the `apply_COVSubscription(NodeId nodeId, COVLifetime lifetime)` method:

```
public synchronized StatusCode apply_COVSubscription(NodeId nodeId, COVLifetime lifetime){

if(DriverAccessLock == true){
  return null;
}
DriverAccessLock = true;
try{
```

```
  currentBACnetAddress = getBACnetAddress(nodeId);
  if(currentBACnetAddress != null){
    //find proper BACnet device
    for(RemoteDevice BACnetRemoteDevice : BACnetLocalDevice.getRemoteDevices()){
      if(BACnetRemoteDevice.getInstanceNumber() == currentBACnetAddress.getDeviceObjectID()){
        System.out.println("RemoteDevice: " + BACnetRemoteDevice.getInstanceNumber());
        int PId = ((UnsignedInteger)nodeId.getValue()).intValue();
        switch(lifetime){
          case default_time:
            BACnetLocalDevice.send(BACnetRemoteDevice, new SubscribeCOVPropertyRequest(
              new com.serotonin.bacnet4j.type.primitive.UnsignedInteger(PId),
              currentBACnetAddress.getObjectIdentifier(), new Boolean(true),
              DEFAULT_COV_LIFETIME, new PropertyReference(currentBACnetAddress.
              getPropertyIdentifier()), null));
            System.out.println("COV subscription applied");
            break;
          case cancel:
            BACnetLocalDevice.send(BACnetRemoteDevice, new SubscribeCOVPropertyRequest(
              new com.serotonin.bacnet4j.type.primitive.UnsignedInteger(PId),
                currentBACnetAddress.getObjectIdentifier(), null, null,
                new PropertyReference(currentBACnetAddress.getPropertyIdentifier()), null));
              System.out.println("COV subscription removed");
              break;
        }
        DriverAccessLock = false;
        return StatusCode.GOOD;
      }
    }
  }
}
catch(Exception e){
        e.printStackTrace();
        DriverAccessLock = false;
        return StatusCode.BAD;
}
DriverAccessLock = false;
return StatusCode.BAD;
}
```

After setting the `DriverAccessLock` flag to avoid recursive calls of the `getBACnetAddress(nodeId)` method, the BACnet address is determined and the proper remote device is looked up. If this has been successful, a `SubscribeCOVProperty` service request is generated by calling the `send` method of the BACnet4j stack with corresponding arguments. The parameters of the `SubscribeCOVProperty` service request depend on the required action, whether to apply a COV subscription or to cancel one. If a COV subscription should be applied, the `IssueConfirmedNotifications` (set to `TRUE`) parameter and the `Lifetime` parameter (set to a default value via the `DEFAULT_COV_LIFETIME` member variable of the `CometDRVManager` class) must be present (cf. [20] Section 13.15).

Otherwise, if a cancel request should be performed, these two parameters must be absent. Therefore, `null` is passed as arguments of the `SubscribeCOVPropertyRequest`.

In case a BACnet `SubscribeCOVProperty` request is not acknowledged by the target controller or the subscription failed, the `send` method of the BACnet stack throws an exception which is cought by the `apply_COVSubscription(NodeId nodeId, COVLifetime lifetime)` method. This results in a `StatusCode.BAD` returned to the server.

The `ObjectIdentifier`, the `PropertyIdentifier` and the `SubscriberProcessIdentifier` are mandatory arguments of the `SubscribeCOVProperty` service. The latter one is used to indicate the context of the subscription, i.e. which client and which process within the client has applied it. As proposed in Section 4.5, this parameter is set to the value of the `NodeId` of the corresponding OPC UA node to create a server-wide unique

assignment.

If a COV subscription is applied to a BACnet property and a COV notification is received by the BACnet4J stack, the `covNotificationReceived` method of the `DefaultDeviceEventListener` class is called:

```
class Listener extends DefaultDeviceEventListener {
    @Override
    public void covNotificationReceived(com.serotonin.bacnet4j.type.primitive.UnsignedInteger
      subscriberProcessIdentifier, RemoteDevice initiatingDevice, ObjectIdentifier
      monitoredObjectIdentifier, com.serotonin.bacnet4j.type.primitive.UnsignedInteger
      timeRemaining, SequenceOf<PropertyValue> listOfValues){
      .
      .
      .
    }
    .
    .
    .
}
```

The property value(s) that are affected by a change are provided by the `listOfValues` argument. To correctly assign the notification to a subscription, the `SubscriberProcessIdentifier` is also passed.

After a type casting which is shown exemplarily for the BACnet `REAL` datatype in the listing the `writeFromDriver(NodeId nodeId, DataValue value, long dpState)` method of the server interface is called:

```
if(listOfValues.get(BACNET_SEQUENCE_OFFSET).getValue() instanceof
  com.serotonin.bacnet4j.type.primitive.Real){
    writeFromDriver(new NodeId(nameSpaceIndex, subscriberProcessIdentifier.intValue()),
    new DataValue(new Variant(((Real)listOfValues.get(BACNET_SEQUENCE_OFFSET).
    getValue()).floatValue())), 0);
}
```

Again, the `Subscriber Process Identifier` holding the `NodeId` of the node representing the BACnet property is passed so that the server can update the value of the correct node in its address space and publish the new value to the OPC UA client having requested the subscription.

5.4 Interoperability Test Lab

In order to show the correctness of the information model and the OPC UA server developed in this work, an instance of a real-world BACnet controller, a *Siemens PXC64-U*, was modelled by means of the Comet UA Model Designer. The BACnet controller (shown in Figure 5.6) is connected via its I/O modules to a temperature sensor and the input of a lighting actuator. This lighting actuator acts as an electronic control gear (ECG) for a luminescent lamp which is controllable by its 0-10V input. The temperature sensor is a stock *PT1000* resistance thermometer. As a consequence of this peripheral equipment the controller implements one `AnalogInputObject` and one `AnalogOutputObject` as a BACnet representation of these devices. Additionally some other BACnet objects like `BinaryInputObjects`, `BinaryOutputObjects`, `BinaryValueObjects` and `AnalogValueObjects` which do not have peripheral counterparts, are instantiated. Table 5.1 gives a summary of all the datapoints parameterised on the controller, including the name of the datapoint (*DP Name*), the hardware address in the notation of `I/O module.Port`, a description, the BACnet `Device_Identifier`

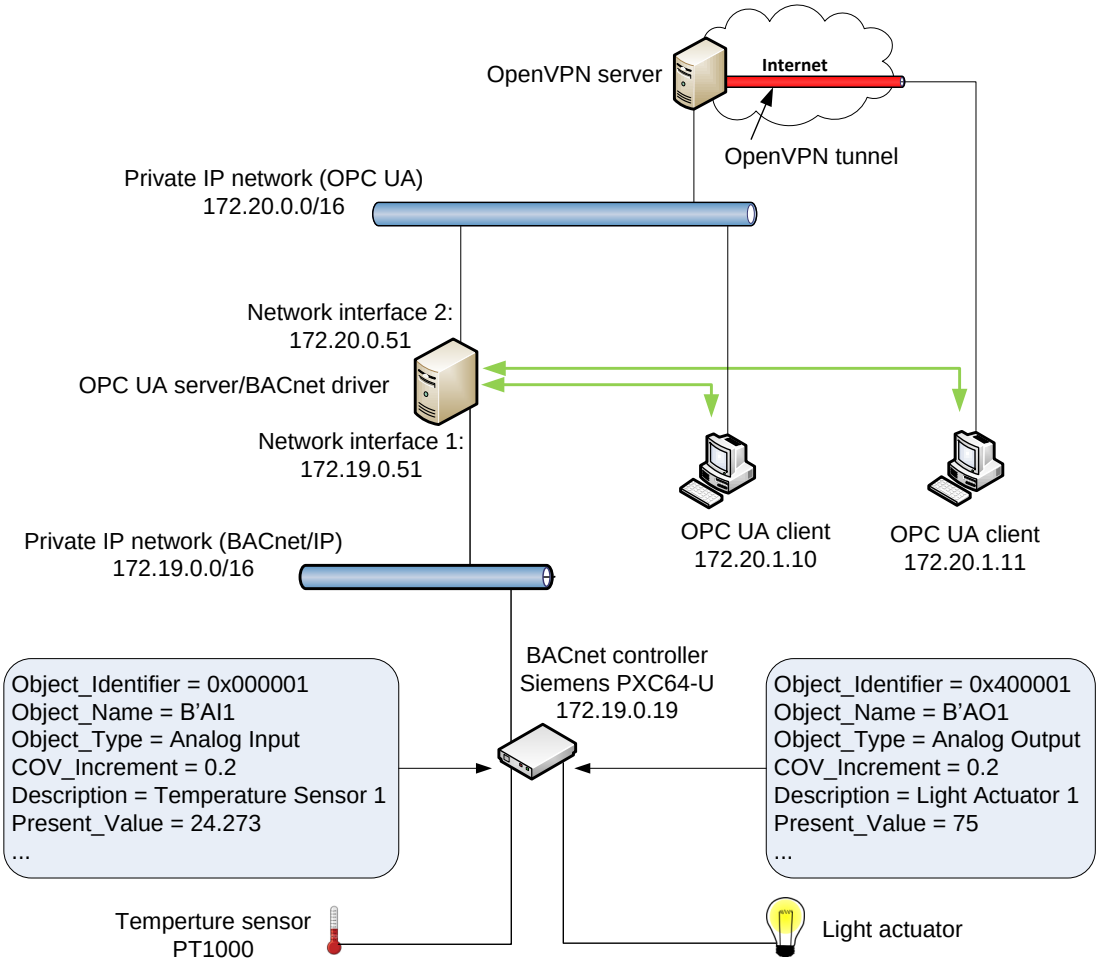

Figure 5.6: Test Lab Setup

and the `Object_Identifier`. The resulting OPC UA information model of this BACnet controller was developed with the Comet Model Designer, exported as an XML file and further loaded into the OPC UA server.

The OPC UA server with its BACnet interface is connected to the same private IP network (`172.19.0.0/16`) as the BACnet controller. This network can be considered as the automation level network, referring to the automation pyramid model. The other network interface used for OPC UA communication is connected to another private network (`172.20.0.0/16`). The latter private network is accessible from the Internet via a VPN connection. An OpenVPN[7] server takes care that only authenticated users can connect to this network. This security measure avoids vulnerability of the OPC UA server in case of a bad security configuration of the

[7] http://openvpn.net/

DP Name	I/O Address	Description	DeviceID / ObjectID
TEMP_1	1.1	Temperature Sensor 1	29054 / Analog Input Object 1
AI_2	1.2	not connected	29054 / Analog Input Object 2
BI_1	1.3	not connected	29054 / Binary Input Object 1
BI_2	1.4	not connected	29054 / Binary Input Object 2
LGHT_1	1.5	Light Actuator 1	29054 / Analog Output Object 1
AO_2	1.6	not connected	29054 / Analog Output Object 2
AV_1	-	Virtual Datapoint 1	29054 / Analog Value Object 1
DV_1	-	Virtual Datapoint 2	29054 / Binary Value Object 1

Table 5.1: List of Datapoints

server during testing. For normal operation a well configured firewall would be sufficient since OPC UA already provides strong security mechanisms (cf. Section A.2).

Any available OPC UA client can now be used to connect to the OPC UA server, for example the *UaExpert* which is released as Freeware by *Unified Automation*[8]. Figure 5.7 shows a screenshot of a part (the `AnalogInputObject` which represents the analog input of the controller the temperature sensor is connected to) of the server address space displayed by UaExpert. By applying a `Read` service to the `PresentValue` node, the temperature value can be accessed.

Further, the analog output of the BACnet controller where the light actuator is connected, can be controlled by performing a write access to the `PresentValue` node of the associated `AnalogOutputObject`. As an additional feature of this installation to provide feedback to operators at remote locations (especially useful for demonstrations), a simple IP webcam focused on the luminescent lamp is set up.

In order to apply a subscription to a datapoint, a specific node can be dragged and dropped to the *Data Access (DA)* view of the UaExpert, which is also shown in Figure 5.7. This triggers the OPC UA server to call the COV subscription method of the BACnet driver which in turn applies the `SubscribeCOVProperty` service of the BACnet stack to the specific BACnet property. The node values displayed by the UaExpert are updated continuously this way, depending on the `Publishing Interval` and the `Sample Rate` settings also adjustable via UaExpert.

[8] http://www.unified-automation.com

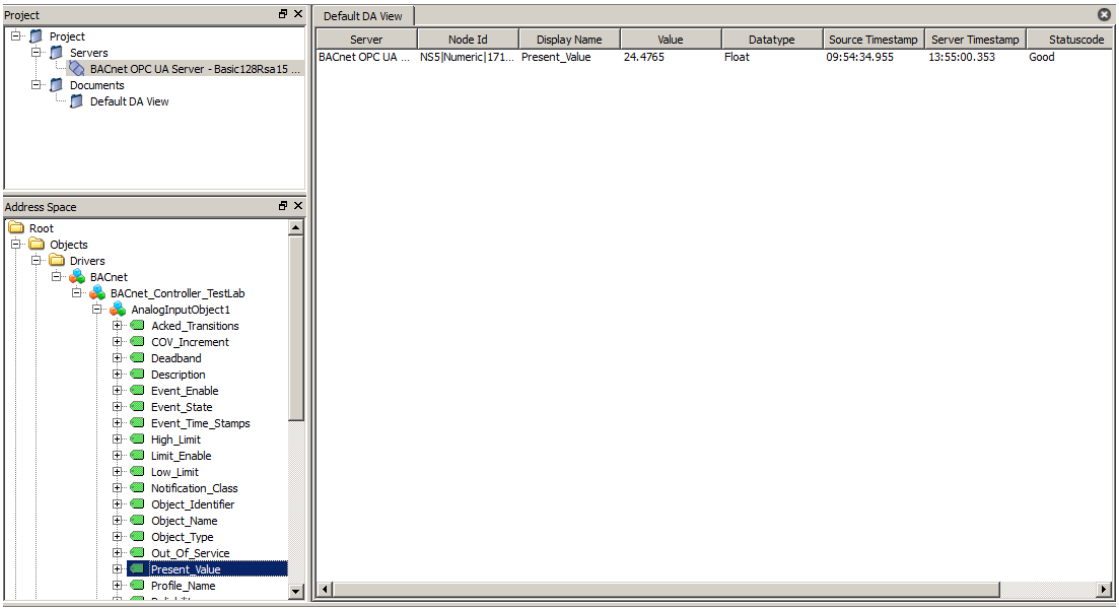

Figure 5.7: OPC UA Server Address Space explored by UaExpert

CHAPTER 6

Conclusion and Outlook

In this work an approach of developing an OPC UA information model for BACnet was presented. As an example, a few BACnet objects (`Device Object`, `Lighting Output Object`, and the `Analog Output Object`) were taken to show the way of mapping a BACnet object with its properties to an OPC UA information model. Also the BACnet addressing scheme was transferred to this model. Following this pattern, further existing BACnet objects can be integrated in this information model. On the way to a complete information model, the mapping of BACnet services needs to be refined. One example is the `Lifetime` parameter of the BACnet `SubscribeCOV` service, which has currently no reflection in the information model. Also for the BACnet services, which were not in the focus of this work, a mapping has to be found. Especially the alarm and event service sets, which allow to monitor BACnet properties and generate alarms on particular conditions, have great practical relevance and so a mapping to monitoring and subscription mechanism of OPC UA should be one of the next steps.

The proof-of-concept implementation of a BACnet driver for an OPC UA server developed in the course of this thesis can also be enhanced. On the one hand, the mapping of the remaining services mentioned above needs to be integrated into the driver. On the other hand, to make this server implementation more useful in a real life BAS, a dynamic address space concept should be introduced. This means that the server should generate its address space reflecting BACnet devices and their objects during initialisation time based on a scan of the BACnet network. Furthermore, the address space should be kept consistent during runtime by adapting it to changes of the BACnet configuration (i.e. when objects are removed or new ones are added).

To achieve the desired interoperability between different standards used in BAS, information models have to be introduced for other technologies, too. A similar information model that maps the interworking model of KNX into OPC UA was already presented in [30]. A final step in this process is to design a general information model representing the common aspects of BAS.

Also within the focus of the WebCom project was to implement a KNX driver for the OPC UA framework presented in Chapter 5. It acts as a proof of concept like the BACnet driver implemented in the context of this work. It can furthermore be integrated in the setup of the interoperability test lab introduced to show the feasibility of accessing both network technologies

in a unified way. Also further distributed tests over the Internet can be run with this extended setup.

In the course of the ongoing research project *Information Modelling in Automation*[1] *(iModelA)* which acts as the successor of WebCom, further efforts regarding this topic are underway. The goal of this project is to close the gap between the domains of building automation and industrial automation by including energy consumption data and device configuration data into the information models describing the underlying technologies. The BACnet information model will also be enriched under these aspects. The Comet UA server for BACnet described in this thesis is planed to be extended by interfaces to LONWorks and M-Bus.

Currently, a joint working group consisting of members of the OPC Foundation, the BACnet Interest Group Europe[2], the automation industry and the Automation Systems Group is laying down a companion specification for BACnet and OPC UA. The information model introduced in this thesis acts as a basis therefor. A first version of the new specification is expected to be published in spring 2013. Since working prototypes of OPC UA servers for BACnet are planed to be presented to the community, the Comet UA server for BACnet might play a further role in this context.

[1] http://imodela.org
[2] www.big-eu.org

List of Figures

List of Tables

Bibliography

[1] BACnet I/P for Java, bacnet4j.sourceforge.net/.

[2] Field Device Integration (FDI), www.fdtgroup.org.

[3] IEC 61131-3/PLCopen, www.plcopen.org.

[4] International Electrotechnical Commission, www.iec.ch.

[5] Java security overview, docs.oracle.com/javase/6/docs/ technotes/guides/security/overview/jsoverview.html.

[6] OpenXPI, www.openxpki.org.

[7] SOAP Version 1.2, www.w3.org/tr/soap.

[8] Xml schema definition for opc ua information models, opcfoundation.org/ua/2008/02/types.xsd.

[9] Online Certificate Status Protocol. RFC 2560, 1999.

[10] Public Key Cryptography Standard #10. RFC 2986, 2000.

[11] Building Automation and Control Systems (BACS) – Part 2: Hardware. ISO 16484-2, 2004.

[12] Communication systems for and remote reading of meters. EN 13757-2, EN 13757-3, 2005.

[13] Lightweight Directory Access Protocol. RFCs 4510-4519, 2006.

[14] oBIX 1.0 Committe Specification. OASIS, 2006.

[15] ZigBee 2007. ZigBee Aliance, 2007.

[16] KNX Handbook, System Specifications: Communication Media: Twisted Pair 1, 2008.

[17] Open Data Communication in Building Automation, Controls and Building Management – Control Network Protocol – Part 1-5. ISO 14908-1 - ISO 14908-4, 2008.

[18] KNX Specification. ISO/IEC 14543-3, 2009.

[19] OPC UA Specification. OPC Foundation, 2009.

[20] BACnet – A Data Communication Protocol for Building Automation and Control Networks. ANSI/ASHRAE 135, 2010.

[21] Building Automation and Control Systems (BACS) – Part 5: Data Communication Protocol. ISO 16484-5, 2010.

[22] OPC Unified Architecture. IEC 62541, 2010. Current status: Approved for FDIS circulation.

[23] BACnet – A Data Communication Protocol for Building Automation and Control Networks. ANSI/ASHRAE 135-2010: Addendum i, 2011. Status: 5th public review.

[24] Public-Key Infrastructure (X.509) (pkix), datatracker. ietf.org/wg/pkix/charter, Last visited November 2011.

[25] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations, Second Edition*. Addison-Wesley Professional, 2002.

[26] Dominick Baier. Support certificates in your applications with the .net framework 2.0, msdn.microsoft.com/en-us/magazine/cc163454.aspx. *MSDN Magazine*, 2007.

[27] R. Cupek and A. Maka. OPC UA for vertical communication in logistic informatics systems. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–4, 2010.

[28] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley and Sons, 2003.

[29] Andreas Fernbach, Wolfgang Granzer, and Wolfgang Kastner. Interoperability at the Management Level of Building Automation Systems: A Case Study for BACnet and OPC UA. *Proc. of 16th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '11)*, Septempber 2011.

[30] Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. KNX and OPC UA. In *Konnex Scientific Conference*, November 2010.

[31] T. Hannelius, M. Salmenpera, and S. Kuikka. Roadmap to adopting OPC UA. In *IEEE International Conference on Industrial Informatics*, pages 756–761, 2008.

[32] Wayne Jansen. Directions in security metrics research. *NISTIR 7564*, April 2009.

[33] A.P. Kalogeras, J.V. Gialelis, C.E. Alexakos, M.J. Georgoudakis, and S.A. Koubias. Vertical integration of enterprise industrial systems utilizing web services. *IEEE Transactions on Industrial Informatics*, 2(2):120–128, May 2006.

[34] Wolfgang Kastner, Georg Neugschwandtner, Stefan Soucek, and H. Michael Newman. Communication Systems for Building Automation and Control. *Proceedings of the IEEE*, 93(6):1178–1203, June 2005.

[35] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer, 2009.

[36] Microsoft Corporation. Best practices for implementing a microsoft windows server 2003 public key infrastructure, www.microsoft.com/download/en/confirmation. aspx?id=20677. *Microsoft TechNet*, 2005.

[37] OPC Foundation Randy Armstrong and Yokogawa Paul Hunkar. The OPC UA Security Model For Administrators. *OPC Foundation*, July 2010.

[38] Mai Son and Myeong-Jae Yi. A study on OPC specifications: Perspective and challenges. In *International Forum on Strategic Technology*, pages 193–197, 2010.

[39] Holt Sorenson. An Introduction to OpenSSL, Part Three: PKI- Public Key Infrastructure, www.symantec.com/connect/articles/introduction-openssl-part-three-pki-public-key-infrastructure. *Symantec*.

[40] Wolfgang Kastner und Georg Neugschwandtner. Datenkommunikation in der verteilten gebäudeautomation. *Bulletin SEV/VSE*, August 2006.

[41] J. Virta, I. Seilonen, A. Tuomi, and K. Koskinen. SOA-based integration for batch process management with OPC UA and ISA-88/95. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–8, 2010.

[42] Stefan-Helmut Leitner Wolfgang Mahnke. OPC Unified Architecture: The future standard for communication and information modeling in automation. *ABB Review*, March 2009.

APPENDIX A

Certificate Management in OPC UA Applications: An Evaluation of different Trust Models

A.1 Introduction

OPC Unified Architecture (OPC UA) provides a powerful and inherent security model. These mechanisms rely on software certificates. In an automation system where OPC UA is applied, a strategy must be defined how to manage these certificates, i.e. an organised way of distribution, validation and revocation needs to be found. In general, there exist different concepts of how to achieve this goal. Moreover, there are various, in some cases platform dependent frameworks available which assist the developer in implementing a suitable concept. The aim of this paper is to give an overview of these concepts and frameworks and discuss their positive and negative aspects depending on the structure of different environments in which OPC UA applications shall be embedded.

Contrary to the past OPC specifications, security is mandatory in OPC UA [19]. This should avoid bad experiences like made in the past with respect to developers of OPC products relying on the security mechanisms of the operating systems the OPC application runs on top of. This resulted in many systems being insecure and vulnerable. The security measures of OPC UA are unbundled of the operating system, so the developers of OPC UA products have almost full control over the security level of an application running in a distinct environment.

OPC UA provides a very flexible security model that can be adapted to the desired use case. There are different requirements on security, dependability and performance depending on the system in which OPC UA applications are embedded. Also the number of OPC UA products installed as well as the human and financial resources available have an influence on the question of which approach leads to the optimal solution. The challenge is to find a trade-off between

these requirements for each distinct environment. This work is intended to provide assistance in addressing this decision.

In the following section, an introduction into the OPC UA security architecture is given. The communication between an OPC UA server and a client is based on a session on top of a *Secure Channel*. For this purpose certain digital certificates are necessary. This leads to the question of how to establish trust relationships between OPC UA applications. Section A.4 is dedicated to this topic. In the following Section A.5 various software frameworks aiming at this target are presented. The paper ends with a discussion (Section A.6) of a guideline where several methods of managing certificates based on different models of trust are compared and evaluated.

A.2 The OPC UA Security Architecture

Secure connections between an OPC UA client and a server are based on a three-layer architecture. This is shown in Figure A.1. Connections established by the transport layer of the OSI Reference Model are based on server and client sockets. Here it is taken care of error detection and error recovery to achieve a reliable connection between the communication partners.

Based on this socket connection a secure channel is opened by the communication layer of the OPC UA protocol stack. The communication layer is responsible for exchanging data in a secure way. Therefore, multiple requirements have to be fulfilled:

- *Data integrity* is guaranteed by digitally signing the content transmitted.
- *Data confidentiality* is a assured by encryption of data.
- Applications have to identify other applications by *authentication* and *authorisation* mechanisms. ITU[1] X.509 certificates are applied for this purpose. The same is an option for users instead of password-based authentication and authorisation.

The application layer on top of this architecture provides services for transmitting data, calling methods and exchanging configuration data between server and client in a *Session*. Within a session, communication partners like users and certain products have to be *authenticated* and *authorised*. These tasks are managed by the OPC UA session services defined in the OPC UA specification Part 4.

A.3 Connection Establishment

In the following, the establishment of a secure channel between an OPC UA client and a server is described. Its main purpose is to exchange secret information for calculating symmetric keys used for data encryption and also for signing communication data. These operations are less CPU intensive using symmetric keys rather than using asymmetric ones.

In the beginning, the client either has preconfigured the connection settings to be used or not. This configuration includes the *security policy* (the algorithms used for signing and encryption as well as the algorithm for key derivation) and the session endpoint of the server. If the

[1] International Telecomunication Union, www.itu.int

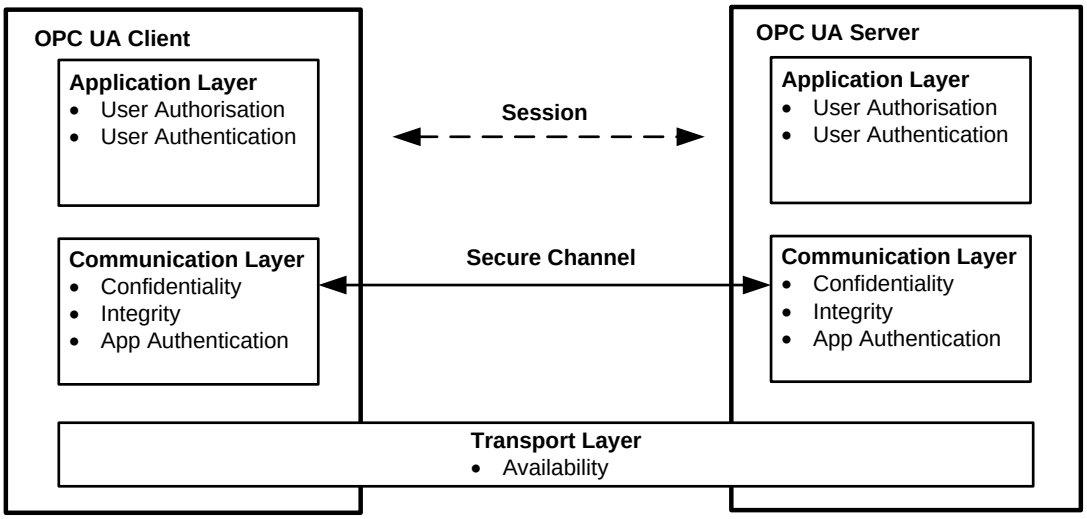

Figure A.1: OPC UA Security Architecture (cf. [19] Part 2)

client does not have this knowledge, it sends an unencrypted `GetEndpoints request` to the server. This is shown in Figure A.2. The server returns a `GetEndpoints response` which contains the supported security configurations and the *Server Application Instance Certificate*. An application instance certificate in OPC UA is used to identify a running application. After this certificate has been validated by a *Validation Authority*, the client sends an `OpenSecureChannel Request` which is already secured by the selected security policy to the server. The `OpenSecureChannel Request` contains the clients *Client Application Instance Certificate*. The server accesses the validation authority and checks if this certificate is valid. If so, it returns a similarly encrypted `OpenSecureChannel Response`. This indicates a secure channel being established.

In a second step, in order to establish a session on top of the already established secure channel, the client sends a symmetrically encrypted `CreateSession request` to the server. The `CreateSession response` of the server contains the *Server Software Certificates* which prove the functional capabilities of the server. Another purpose of these certificates is to identify a specific OPC UA product. These certificates are verified by a validation authority the client sends them to.

At last, the session needs to be activated. Therefore, the client sends an `ActivateSession Request` to the server. This message contains the user credentials and the *Client Software Certificate*. The user credentials are usually a username-password combination, but they can also be provided in form of another X.509 certificate. The server validates the client's software certificate by a validation authority. The server checks the user credentials either by a database lookup in case a username and a password is used or by a validation authority if a certificate is applied. If successful, it returns an `ActivateSession Response` to set up the session.

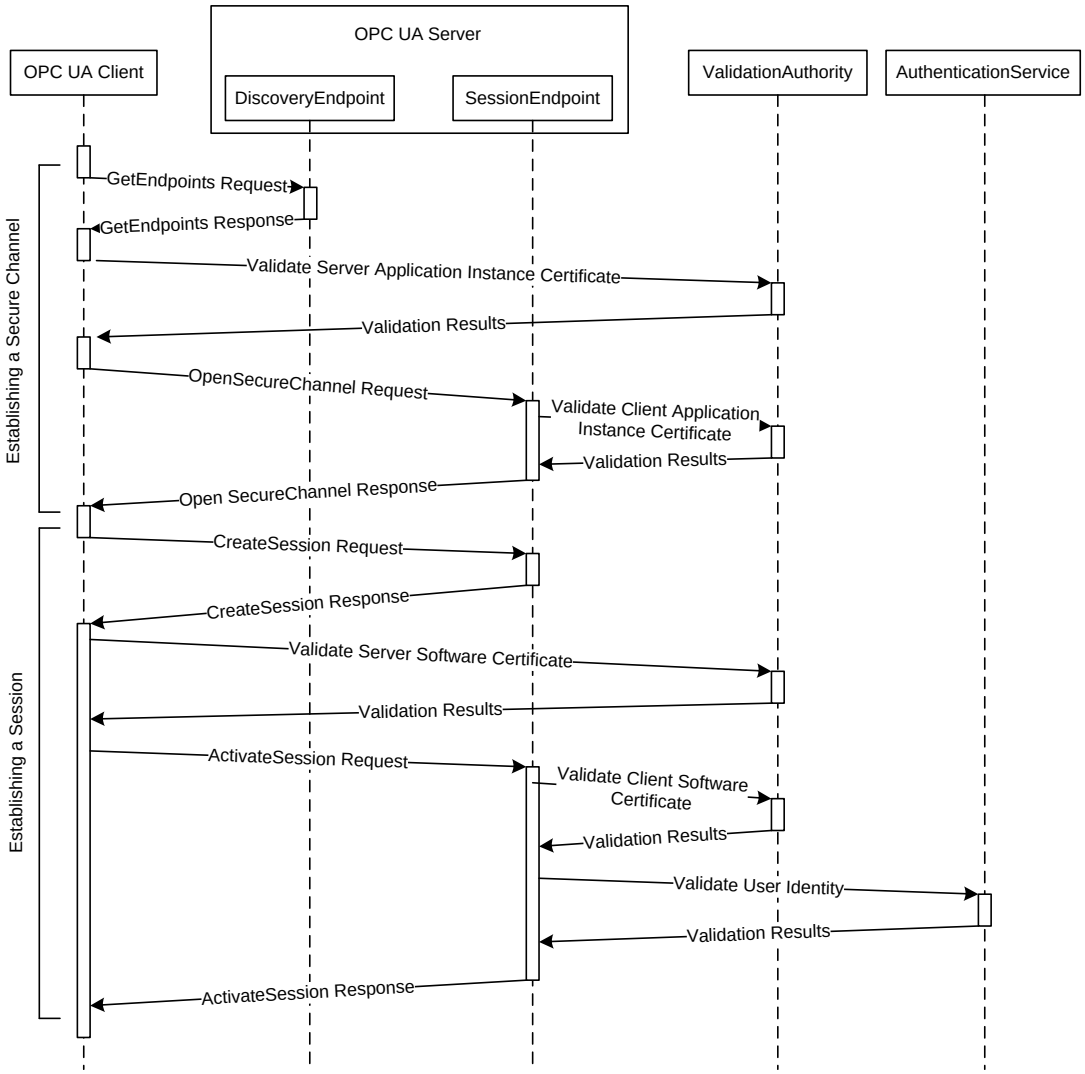

Figure A.2: Establishing an OPC UA Connection [35]

A.4 Certificate Management in OPC UA Applications

This section first gives an introduction into digital certificates and how they are created and used. There exist different rules describing which entity creates certificates and how they are validated. These sets of rules constitute various trust models which are presented in the following.

Certificates

A digital certificate is an electronic document basically containing a *Public Key* (for deeper information about public key cryptography cf. to [28]) and identity information about the owner of the certificate. A digital signature applied by a trusted third party, the *Certification Authority (CA)* or by the owner itself (*self signed certificates*) is used to bind these two attributes together. This way, every other entity can check if the integrity of the certificate is preserved. Other attributes which are also included in a digital certificate are a serial number, a version field, the issuer of the certificate (the CA or the owner) and the validity period. There may be also a field to be filled in with further information.

The public key of a public/private key pair bound to the owner of the certificate is used for example for message encryption between a client and a server.

There must be found a way of organising the creation, distribution, validation and revocation of certificates. Some technical and organisational infrastructure is necessary to achieve this goal. The OPC UA standard does not define how such an infrastructure should look like. However, there exist some general concepts how to implement such an infrastructure. The following section introduces these concepts where each of them is suitable for a different application scenario.

Trust Models

Trust between *End Entities (EEs)* is achieved by either trusting in their associated certificates or trusting in a third party (*trusted third party*) that has previously authenticated the other entity. This is reflected in the different trust models introduced in the following.

A trust model can be organised in two ways, hierarchical by using one or more CAs or user-centric (decentralised) by applying the models of *Direct Trust* or *Web of Trust*.

- A *Web of Trust* gets along without any CA as a trusted third party. It only consists of EEs which make their own decision of whom to trust or not. This principle is applied in *Pretty Good Privacy (PGP)* and its open source siblings *OpenPGP* and *GnuPG*. This model does not scale well, since every EE needs to store a certificate of each EE it trusts. Also finding a trust path from one EE to another EE in big sets can consume a lot of computational power.

- The *Direct Trust Model* does not need any trusted third party, either. There are individual trust relations between the EEs. These trust relations have to be set individually for each EE. Therefore it does not scale well for big projects, either. This model is a very labour intensive one because usually the certificate distribution must be done manually (*out-of-band*). This method is only suitable for small environments but unreliable und inefficient for large scales.

- In a *Public Key Infrastructure (PKI)* there is one CA or even more CAs as trusted third parties in a hierarchy of inheritance which are organised like shown in Figure A.3. A hierarchical organisation of CAs also results in a trust hierarchy, where a sub CA always trusts its super CA. Sub CAs can be assigned to particular units of an enterprise. This model scales well for big projects.

  Another way of organising CAs is a full-meshed architecture, which is a convenient approach if there is a lot of communication between different units of an enterprise. This way, trust paths are kept short since there is a direct relationship between the CAs instead of going up to the common root and back down the other branch. On the other hand, path discovery may be more difficult since there may be multiple choices.

  Each way of structuring a PKI has its advantages and disadvantages. A pro for a single CA is that it is easy to maintain. On the other hand, it has limiting effects on the size of the organisation. A multiple CA architecture scales well for big organisations but this advantage has to be bought by an administrative effort multiplied by the numbers of CAs present within the system.

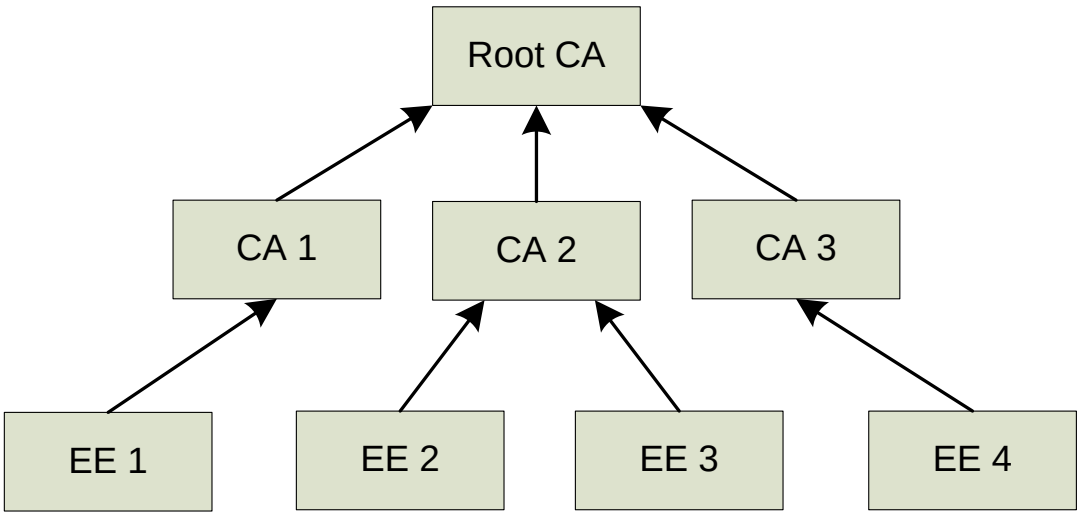

Figure A.3: Hierarchical Trust Model [35]

Public Key Infrastructure

Concluding the different aspects of the particular trust models it can be claimed that a PKI is the most suitable one for the majority of OPC UA applications. The direct trust model is only applicable for very small organisations and the Web of Trust does not scale well, either. Therefore, a closer look at the structure of a PKI is taken. A PKI consists of the following entities which are illustrated in Figure A.4:

- An end entity (EE) can be either an OPC UA product or a user. It requests and uses the certificates issued by the CA.

- The CA is the trusted third party in a PKI. It generates documents based on the identity of end entities and the CA's private key. These documents are issued as certificates to other end entities.

- A *Registration Authority* (RA) is not an essential but optional component of a PKI. It performs tasks on behalf of the CA like verifying the identity of an end entity and checking, if an end entity is allowed to have a certificate, have its certificate renewed or revoked. After this verification, the RA forwards the EE's request to the CA.

- The *Validation Authority* (VA) has the purpose of validating certificates that EEs provide to it and returns the result of this calculation to the EEs. The validation process is performed by verifying the signature, checking the validity period and if the certificate has not been revoked. It must also be examined, if the usage of the certificate is within the specified purpose.

In a PKI there exists a so called *Certificate Lifecycle*. Figure A.4 shows the different entities in a PKI and how they interact within a certificate lifecycle. It starts with the request of certificates by the EEs. After verification of the request by the RA, the request is propagated to the CA. The next step is to distribute the certificates among the EEs that issued the request. Now the EEs (OPC UA applications or users) can take the certificates for authorising and authenticating themselves or for message encryption. Since there is a limited period of time in which certificates are valid, they need to be renewed or updated in case they are expired. If necessary, certificates can also be revoked. This is the case if e.g. the private key associated to the certificate is compromised or the certificate is not needed anymore. The usual approaches of setting up a PKI that manages the certificates lifecycle mainly differ in the methods of distribution and revocation.

There are the following ways of distributing certificates issued by a CA among the requesting EEs:

- Out-of-band: This method is performed manually by transporting the certificates on a storage medium (e.g. disk, usb-stick) to the EE or transferring it by email to the target entity. Here it is imported into a local repository. This approach is an easy solution for small environments but does not scale well for big ones since it is labour intensive and unreliable because of the human component involved.

- Certificates can also be published in a central, well known, public repository like a *Lightweight Directory Access Protocol (LDAP)* [13] server. This can be seen as a Web server which provides access to a database containing certificates. The database content is controlled by a CA. This approach provides automatic download of certificates which makes it to a reliable solution. On the other hand, a single server is always in danger to be confronted with *Denial of Service (DoS)* attacks. Additional network traffic on an extra channel is also caused following this approach.

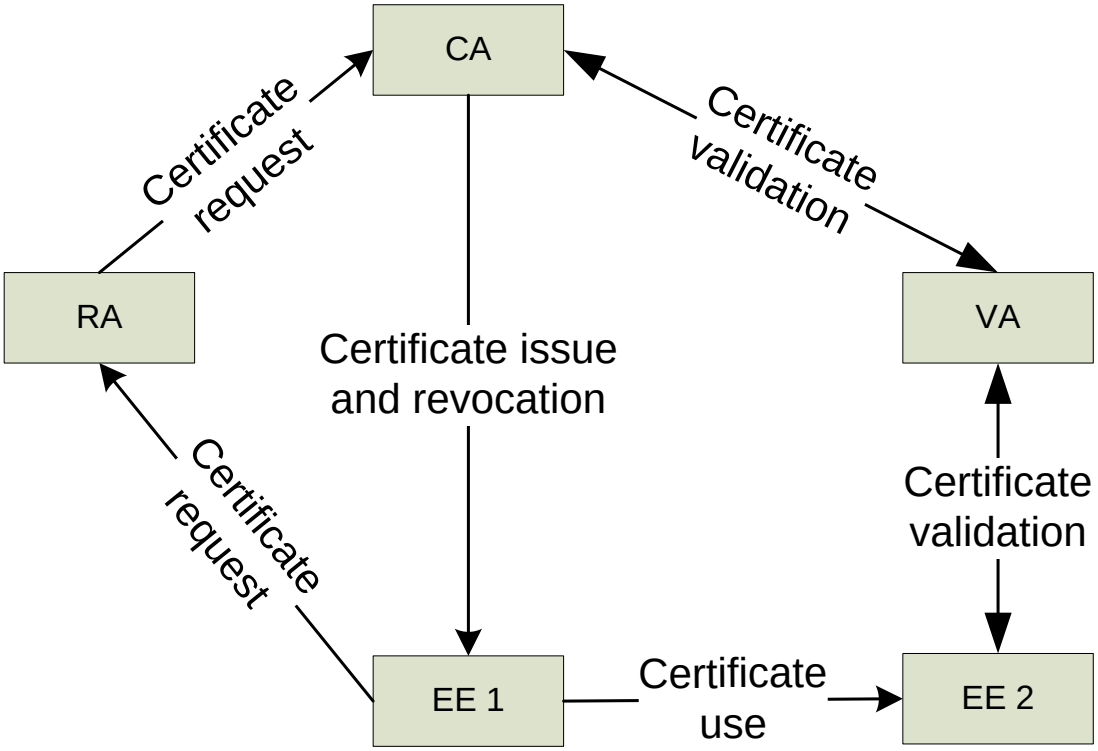

Figure A.4: PKI enities and their interaction

- In-band distribution means that an application-specific communication protocol is used for exchanging certificates. This way, no additional channel or protocol is necessary. *Secure/Multipurpose Internet Mail Extensions (S/MIME)*, *Transport Layer Security (TLS)* (cf. [25]) or even services from OPC UA can be used for this purpose.

For certificate revocation there also exist multiple choices:

- Following the *offline* approach so called *Certificate Revocation Lists (CRL)* are downloaded by the EEs from public, well-known locations like LDAP, FTP, HTTP servers at certain intervals. A CRL contains information about all the revoked certificates of a trust domain. It is signed by the CA that publishes the list. A CRL always reflects the past. It cannot provide any information about the current validity of a certificate. The recency of a CRL depends on the update intervals, which are determined as a trade-off between network load and the desire of having an up-to-date CRL.

- The other choice is using *online* mechanisms to check if a certificate is valid. EEs connect to a service provider every time they use a certificate and thereby have its validity checked. A common way to achieve this is by using the *Online Certificate Status Protocol (OCSP)*

[9]. It can provide real-time information, depending on the source of information the OCSP server relies on. Contrary to CRLs, also positive information can be disseminated about the validity of a certificate, i.e. an OCSP server can explicitly declare a certificate valid.

A.5 Applicable PKI Frameworks for OPC UA

There is a big variety of proprietary and open-source PKI frameworks available today, like OpenSSL[2], Microsoft Windows Server[3], OpenXPKI[4], VeriSign Managed PKI Services[5]. Most popular OPC UA SDKs published by the OPC foundation and the diverse software products based on these SDKs are written in .NET, Java and ANSI C. Therefore, a small guideline of which framework to use to operate OPC UA products in a trusted environment depending on the programming language and other criteria will be given.

The OPC Foundation released two tools that also can be used to manage OPC UA applications and certificates, the *UA Configuration Tool* and the *UA Certificate Generator*. Since there already exists a Whitepaper [37] giving a description of these tools and a guideline about the administrative procedures that lead to a secure environment for OPC UA applications, there is no further discussion about these tools in this work.

Windows Server, .NET

A Windows based PKI is a convenient commercial solution available for a reasonable price. Many enterprises use a Windows environment anyway, so it is only about using additional features of Windows Server and the client operating systems. The Windows *ActiveDirectory Certificate Services* exist since Windows NT 4.0 and can therefore be considered as very mature.

Windows Server machines act as the CAs in a PKI. RAs can be set up by also Windows based *Internet Information Services (IIS)* Web servers. This allows *Web Enrolment*, i.e. an automatic way of certificate dissemination among the EEs. User identity information is also gathered by an IIS server and verified by using ActiveDirectory. LDAP services are also available via ActiveDirectory.

A step by step guideline of how to setup a Windows based PKI is provided by Microsoft TechNet [36].

VAs are represented by the local Windows Certificate Stores of each machine, which provide an abstract, unified way to access certificates. OPC UA applications based on .NET must implement routines to access certificates from the certificate store. A paper that describes how to deal with certificates in .NET applications and also containing example code can be found at the MSDN Magazine Website [26].

[2] http://www.openssl.org/
[3] http://www.windowsserver.com/
[4] http://www.openxpki.org/
[5] http://www.verisigninc.com/

OpenSSL

OpenSSL is an open source toolkit written in C. It implements the *Secure Sockets Layer* (SSL v2/v3) and *Transport Layer Security* (TLS v1) protocols as well as a general purpose cryptography library. It is released under an Apache-like licence. Supported platforms are UNIX-like operating systems and Windows.

Besides the C libraries supporting cryptographic services, OpenSSL also provides a command-line program called *openssl* with all the functionalities necessary for managing a PKI. For this purpose, the toolkit must be installed on every machine of the environment. Issuing certificate requests, the generation of RSA keys and X.509 certificates as well as commands to revoke certificates and to generate a CRL are the main features of this tool.

All these steps can be performed in a manual way where the administrator acts as the CA and the RA. This approach is suitable for small environments. In order to handle certificate management in bigger projects, script based execution of these operations is recommendable. Further information containing a step-by-step instruction of how to set up a PKI with OpenSSL can be found at the Symantec webpage [39].

OpenXPKI

OpenXPKI is an open source software implementation targeting from small installations to enterprise-level (large-scale) PKIs. It is designed for Unix-like operating systems and is released under an Apache licence. The key features of OpenXPKI are the support of multiple CA instances on a single application instance in order to set up a trust hierarchy and the capability of full automatic CA rollover. This way, continuous operation of the PKI is assured without administrator intervention, in case one CA certificate expires and another CA has to take over. High flexibility is achieved by an XML-file controlled workflow engine that allows extending the basic PKI operations.

An OpenXPKI installation can act as a CA, a RA or an EE, depending of its configuration. Certificates, private keys and revocation information are stored in a database system that can be chosen out of the most popular ones like MySQL and Oracle. Documentation for developers is provided on the project Webpage [6].

This solution will mainly be applicable for large scale installations because it causes quite an effort to set up and configure an OpenXPKI based PKI. This work is best done by professional developers. The complexity of this framework is a result of its high flexibility and modular design.

Java

There is a powerful security API delivered with the Java SDK. It also includes classes (`java.security`) related to PKI applications. Their functionalities encompass support for X.509 certificates, CRLs and PKIX-compliant [24] certification path building and validation [5]. Classes that provide a key store (a secure repository for cryptographic keys) and a certificate store are also available. This makes the Java security API mainly interesting for the use in EEs of a PKI.

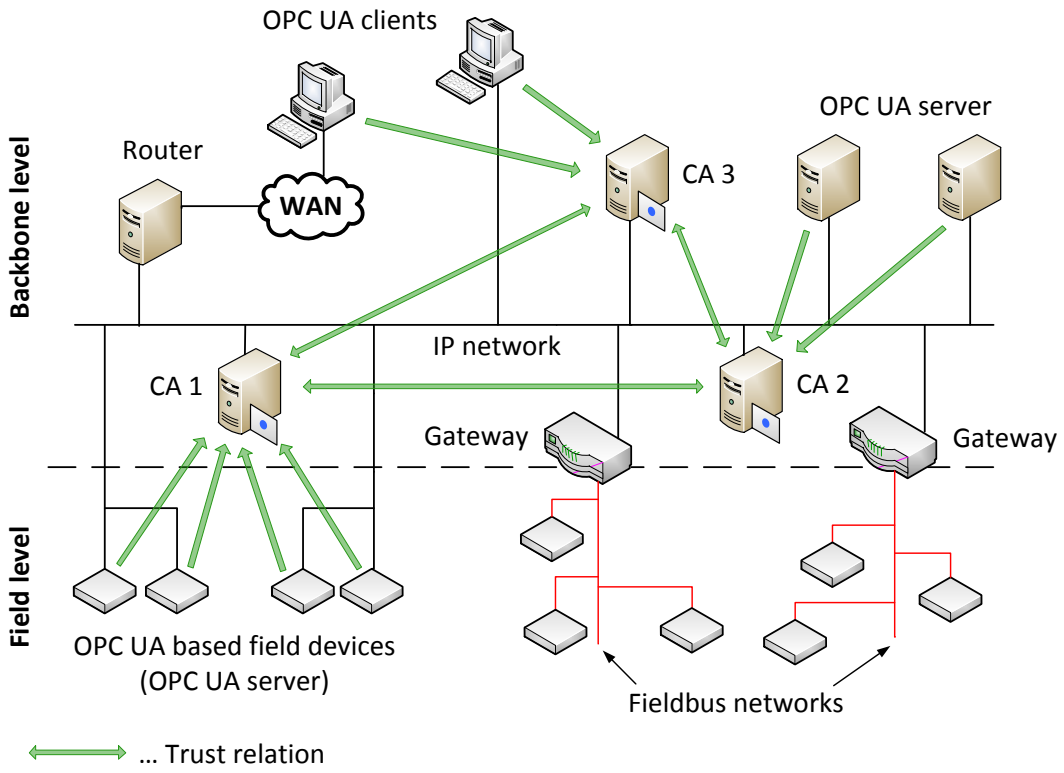

Figure A.5: 2-Tier automation network establishing a hybrid trust model

Additionally, there is a command-line program named `keytool` which can be used for creating and managing key stores. The main features of this tool are:

- Create public/private key pairs and self-signed certificates
- Display, import and export X.509 certificates stored as files
- Issue PKCS#10 [10] (a standardised message format) certificate requests to be sent to CAs
- Import certificate replies obtained from the CAs as responses to certificate requests
- Designate public key certificates as trusted

More details about this tool and the Java security API can be found in the Java SE documentation available at the Oracle Website [5].

A.6 Discussion

The question about the best trust model for a given application is a hard one to find an explicit answer on. Moreover, the development of proper security metrics which would provide methods

for quantitative comparison of different approaches is still an ongoing field of research. The likelihood is that it is not possible to find metrics for every security aspect [32]. The decision on the particular trust model implementation does not only depend on hard facts like size, structure and the equipment already in use but is also based on personal taste. One may prefer the use of free software where it is often more effort to set up a working system. Also hiring external personnel for this purpose may be necessary. Another one favours commercial software which is usually bought as an out-of-the-box product including support. This again results in less effort for the own personnel.

But there are several criteria that lead the decision which solution to apply to a distinct direction:

The *budget* available has a limiting influence on this choice. A single CA PKI is naturally cheaper since there is less maintenance effort compared to the multiple amount of effort caused by a hierarchical or a full-meshed architecture. Higher maintenance effort results in higher personnel costs. For small applications also the direct trust model can be imaginable. This represents the cheapest solution since there is no extra security-related equipment necessary.

The availability of *human resources* has a similar effect on the decision upon a trust model. For small organisations, a single CA architecture or also the direct trust model will be sufficient. This can be seen from two sides: Little human resources operating a small set of machines in a trusted environment issue little requests for certificates. This keeps on the other side the infrastructure small, i.e. the equipment to manage these requests and the personnel maintaining the equipment. The opposite applies to large scales, which will result in an architecture with multiple CA. The decision whether to deploy a hierarchy of CAs or a mesh of CAs depends on other criteria.

Dependability may be a goal to achieve in case danger threatens humans or material if some equipment is not available caused by a faulty trust infrastructure. In a single CA or hierarchical CA architecture there is always a single point of failure: The single CA or the root CA, respectively. On the contrary in a meshed architecture no fail of the whole PKI takes place if one CA is compromised or out of service. Only the users or EEs affected, that have a trust relationship to the CA involved. Recovery is also easier since a new certificate only needs to be distributed among these few affected EEs.

It is finally convenient to map the *structure* of the organisation to the model of trust. This will mostly be applicable if it is already clear that a single CA architecture or the direct trust model will not fulfil the requirements because a distinct size of the organisation is already exceeded. If a hierarchical structure is inherent with the organisation, i.e. there is mainly communication between sub units and super units, then the trust relationship should be organised this way. On the other hand, if there is also considerable communication between units on the same level, the meshed PKI architecture is probably the right choice.

An idea, how trust relations can be organised in a real-world automation system is given in Figure A.5. It illustrates a 2-tier automation hierarchy consisting of a *field level*, where measuring and setting of physical values takes place and a *backbone level* where data from the lower tier is aggregated for visualisation and trending. Configuration of the system and providing an interface to management and enterprise applications are further tasks of the backbone level.

In this example, OPC UA is used at both tiers of the automation system. Having OPC UA

servers at the backbone level to enable standardised and uniform access to process data of the lower level [29], like shown in the right part of Figure A.5 can be seen as the classical approach. Though, the current trend is leading to use OPC UA all down to the field level [2], which is illustrated in the field device network in the left lower part of the figure.

The trust model used in this example follows a hybrid approach. The OPC UA devices (EEs) are part of PKIs and receive their certificates from different CAs. There is one CA for each group of devices: one for the OPC UA field devices, one for the OPC UA servers interfacing a distinct network technology and one for the OPC UA clients. This way, a separation between these groups regarding certificate management is achieved. This provides on one hand a distinct level of dependability and keeps on the other hand the number of EEs per CA limited. In order not to lose dependability gained by the multiple CA approach, there is no single root CA in this system but trust between the CAs is established by implementing a full-meshed architecture. Yet, a compromise is made with respect to scalability of the number of CAs since the number of trust relationships between CAs grows polynomially.

A.7 Conclusion and Outlook

In this paper different methods of certificate management (trust models) in OPC UA applications have been presented. As a reason of the limited scalability of the Web of Trust and the Direct Trust Model, for medium and large scale environments the Public Key Infrastructure evaluated as the most convenient approach of managing certificates. Nevertheless the Direct Trust Model can be an option for an automation system in which a very limited amount of OPC UA devices are installed. As research on security metrics advances, it might be feasible in the future to quantitatively compare the aspects of trust models which have not been discussed in this work.

The choice between the presented frameworks on one hand depends on the operating system preferred or already in use in the environment, respectively. There are frameworks available for Windows, Unix-like platforms as well as the platform-independent Java security API. On the other hand, different features provided by distinct frameworks have influence on this decision.

Finally, a discussion about how the structure of the environmental system where OPC UA applications are installed, the resources available and a request to dependability shall affect the strategy of managing certificates.

Since this is a very practice-related topic, experiences made by companies operating in the field of automation which integrate and maintain OPC UA applications should be taken into account to approve this theoretical work. A survey evaluating the feedback given by OPC UA integrators and end-users could give more insight into this topic.

Acknowledgement

This work was funded by FFG (Austrian Research Promotion Agency) under the EraSME/COIN projects “Web-based Communication in Automation (WebCom)” P824675 and “Information Modeling in Automation (iModelA)” P834800.