

**OPC Unified Architecture**

**Specification**

**Part 1: Concepts**

**Version 1.00**

**July 28, 2006**

Specification Type:	Industry Standard Specification	Comments:	Report or view errata: <a href="http://www.opcfoundation.org/errata">http://www.opcfoundation.org/errata</a>
Title:	OPC Unified Architecture	Date:	July 28, 2006
	<u>Part 1 :Concepts</u>		
Version:	<u>Release 1.00</u>	Software: Source:	MS-Word <u>OPC UA Part 1 - Concepts 1.00 Specification.doc</u>
Author:	<u>OPC Foundation</u>	Status:	<u>Release</u>

**CONTENTS**

	Page
<u>FOREWORD</u> .....	vi
<u>AGREEMENT OF USE</u> .....	vi
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, and abbreviations .....	1
3.1 OPC UA terms .....	1
3.1.1 AddressSpace .....	1
3.1.2 Alarm .....	2
3.1.3 Attribute .....	2
3.1.4 Certificate .....	2
3.1.5 Client .....	2
3.1.6 Communication Stack .....	2
3.1.7 Complex Data .....	2
3.1.8 Event .....	2
3.1.9 EventNotifier .....	2
3.1.10 Information Model .....	2
3.1.11 Message .....	2
3.1.12 Method .....	2
3.1.13 MonitoredItem .....	3
3.1.14 Node .....	3
3.1.15 NodeClass .....	3
3.1.16 Notification .....	3
3.1.17 NotificationMessage .....	3
3.1.18 Object .....	3
3.1.19 Object Instance .....	3
3.1.20 ObjectType .....	3
3.1.21 Profile .....	3
3.1.22 Program .....	3
3.1.23 Reference .....	3
3.1.24 ReferenceType .....	4
3.1.25 RootNode .....	4
3.1.26 Server .....	4
3.1.27 Service .....	4
3.1.28 Service Set .....	4
3.1.29 Session .....	4
3.1.30 Subscription .....	4
3.1.31 Variable .....	4
3.1.32 View .....	4
3.2 Abbreviations and symbols .....	4
4 Structure of the OPC UA series .....	5
4.1 Specification Organization .....	5
4.2 Core Specification Parts .....	5
4.2.1 Part 1 – OPC UA Concepts .....	5
4.2.2 Part 2 – OPC UA Security Model .....	5
4.2.3 Part 3 – OPC UA Address Space Model .....	5
4.2.4 Part 4 – OPC UA Services .....	5

4.2.5	Part 5 – OPC UA Information Model.....	6
4.2.6	Part 6 – OPC UA Service Mappings .....	6
4.2.7	Part 7 – OPC UA Profiles.....	6
4.3	Access Type Specification Parts.....	6
4.3.1	Part 8 – OPC UA Data Access .....	6
4.3.2	Part 9 – OPC UA Alarms and Conditions.....	6
4.3.3	Part 10 – OPC UA Programs.....	6
4.3.4	Part 11 – OPC UA Historical Access .....	6
5	Overview .....	6
5.1	Introduction.....	6
5.2	Design goals.....	6
5.3	Integrated models and services .....	8
5.3.1	Security model .....	8
5.3.2	Integrated <i>AddressSpace</i> model .....	9
5.3.3	Integrated object model .....	9
5.3.4	Integrated services .....	10
5.4	Sessions.....	10
5.5	Redundancy.....	10
6	Systems concepts .....	11
6.1	Overview .....	11
6.2	OPC UA <i>Clients</i> .....	11
6.3	OPC UA <i>Servers</i> .....	12
6.3.1	Real objects .....	12
6.3.2	OPC UA Server application.....	12
6.3.3	OPC UA <i>AddressSpace</i> .....	12
6.3.4	Publisher/subscriber entities .....	13
6.3.5	OPC UA Service Interface .....	13
6.3.6	Server to Server interactions .....	14
7	Service Sets .....	15
7.1	General .....	15
7.2	SecureChannel Service Set.....	15
7.3	Session Service Set.....	16
7.4	<i>NodeManagement Service Set</i> .....	16
7.5	View Service Set.....	16
7.6	Attribute Service Set .....	17
7.7	Method Service Set.....	17
7.8	MonitoredItem Service Set .....	17
7.9	Subscription Service Set .....	17
7.10	Query Service Set.....	18

**FIGURES**

Figure 1 – OPC UA Specification Organization.....	5
Figure 2 – OPC UA Target Applications .....	7
Figure 3 – OPC UA System Architecture .....	11
Figure 4 – OPC UA <i>Client</i> Architecture .....	11
Figure 5 – OPC UA Server Architecture .....	12
Figure 6 – Interactions between Servers .....	14
Figure 7 – Chained Server Example .....	15
Figure 8 – <i>SecureChannel</i> and <i>Session Services</i> .....	16

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these

materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>



## 1 Scope

This specification presents an overview on the OPC Unified Architecture concepts.

## 2 Reference documents

The OPC Unified Architecture Specification is organized as a multi-part document. While describing the concepts, this part will refer to these parts of the specification:

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 6] OPC UA Specification: Part 6 – Mappings, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part6/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part8/>

[UA Part 9] OPC UA Specification: Part 9 – Alarms and Conditions, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part9/>

[UA Part 10] OPC UA Specification: Part 10 – Programs, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part10/>

[UA Part 11] OPC UA Specification: Part 11 – Historical Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part11/>

## 3 Terms, definitions, and abbreviations

For the purposes of this specification, the following definitions apply.

### 3.1 OPC UA terms

#### 3.1.1 AddressSpace

The collection of information that an OPC UA Server makes visible to its *Clients*. See [UA Part 3] for a description of the contents and structure of the *Server AddressSpace*.

### 3.1.2 Alarm

A type of *Event* associated with a state condition that typically requires acknowledgement. See [UA Part 9] for a description of *Alarms*.

### 3.1.3 Attribute

A primitive characteristic of a *Node*. All *Attributes* are defined by OPC UA, and may not be defined by *Clients* or *Servers*. *Attributes* are the only elements in the *AddressSpace* permitted to have data values.

### 3.1.4 Certificate

A digitally signed data structure that describes the capabilities of a *Client* or *Server*.

### 3.1.5 Client

A software application that sends *Messages* to OPC UA *Servers* conforming to the *Services* specified in this set of specifications.

### 3.1.6 Communication Stack

A layered set of software modules between the application and the hardware that provides various functions to encode, encrypt and format a *Message* for sending, and to decode, decrypt and unpack a *Message* that was received.

### 3.1.7 Complex Data

Data that is composed of elements or more than one primitive data type, such as a structure.

### 3.1.8 Event

A generic term used to describe an occurrence of some significance within a system or system component.

### 3.1.9 EventNotifier

A special *Attribute* of a *Node* that signifies that a *Client* may subscribe to that particular *Node* to receive *Notifications* of *Event* occurrences.

### 3.1.10 Information Model

An organizational framework that defines, characterizes and relates information resources of a given system or set of systems. The core address space model supports the representation of *Information Models* in the *AddressSpace*. See [UA Part 5] for a description of the base OPC UA *Information Model*.

### 3.1.11 Message

The data unit conveyed between *Client* and *Server* that represents a specific *Service* request or response.

### 3.1.12 Method

A callable software function.

### 3.1.13 MonitoredItem

A *Client*-defined entity in the *Server* used to monitor *Attributes* or *EventNotifiers* for new values or *Event* occurrences and generate *Notifications* for them.

### 3.1.14 Node

The fundamental component of an *AddressSpace*.

### 3.1.15 NodeClass

The class of a *Node* in an *AddressSpace*. *NodeClasses* define the metadata for the components of the OPC UA Object Model. They also define constructs, such as *Views*, that are used to organize the *AddressSpace*.

### 3.1.16 Notification

The generic term for data that announces the detection of an *Event* or of a changed *Attribute* value. *Notifications* are sent in *NotificationMessages*.

### 3.1.17 NotificationMessage

A *Message* published from a *Subscription* that contains one or more *Notifications*.

### 3.1.18 Object

A *Node* that represents a physical or abstract element of a system. *Objects* are modelled using the OPC UA Object Model. Systems, subsystems and devices are examples of *Objects*. An *Object* may be defined as an instance of an *ObjectType*.

### 3.1.19 Object Instance

A synonym for *Object*. Not all *Objects* are defined by *ObjectType*s.

### 3.1.20 ObjectType

A *Node* that represents the type definition for an *Object*.

### 3.1.21 Profile

A specific set of capabilities, defined in [UA Part 7], to which a *Server* may claim conformance. Each *Server* may claim conformance to more than one *Profile*.

### 3.1.22 Program

An executable *Object* that, when invoked, immediately returns a response to indicate that execution has started, and then returns intermediate and final results through *Subscriptions* identified by the *Client* during invocation.

### 3.1.23 Reference

An explicit relationship (a named pointer) from one *Node* to another. The *Node* that contains the *Reference* is the source *Node*, and the referenced *Node* is the target *Node*. All *References* are defined by *ReferenceTypes*.

### 3.1.24 ReferenceType

A *Node* that represents the type definition of a *Reference*. The *ReferenceType* specifies the semantics of a *Reference*. The name of a *ReferenceType* identifies how source *Nodes* are related to target *Nodes* and generally reflects an operation between the two, such as “A Contains B”.

### 3.1.25 RootNode

The beginning or top *Node* of a hierarchy. The *RootNode* of the OPC UA *AddressSpace* is defined in [UA Part 5].

### 3.1.26 Server

A software application that implements and exposes the *Services* specified in this set of specifications.

### 3.1.27 Service

A *Client*-callable operation in an OPC UA *Server*. *Services* are defined in [UA Part 4]. A *Service* is similar to a method call in a programming language or an operation in a Web services WSDL contract.

### 3.1.28 Service Set

A group of related *Services*.

### 3.1.29 Session

A logical long-running connection between a *Client* and a *Server*. A *Session* maintains state information between *Service* calls from the *Client* to the *Server*.

### 3.1.30 Subscription

A *Client*-defined endpoint in the *Server*, used to return *Notifications* to the *Client*. Generic term that describes a set of *Nodes* selected by the *Client* (1) that the *Server* periodically monitors for the existence of some condition, and (2) for which the *Server* sends *Notifications* to the *Client* when the condition is detected.

### 3.1.31 Variable

A *Variable* is a *Node* that contains a value.

### 3.1.32 View

A specific subset of the *AddressSpace* that is of interest to the *Client*.

## 3.2 Abbreviations and symbols

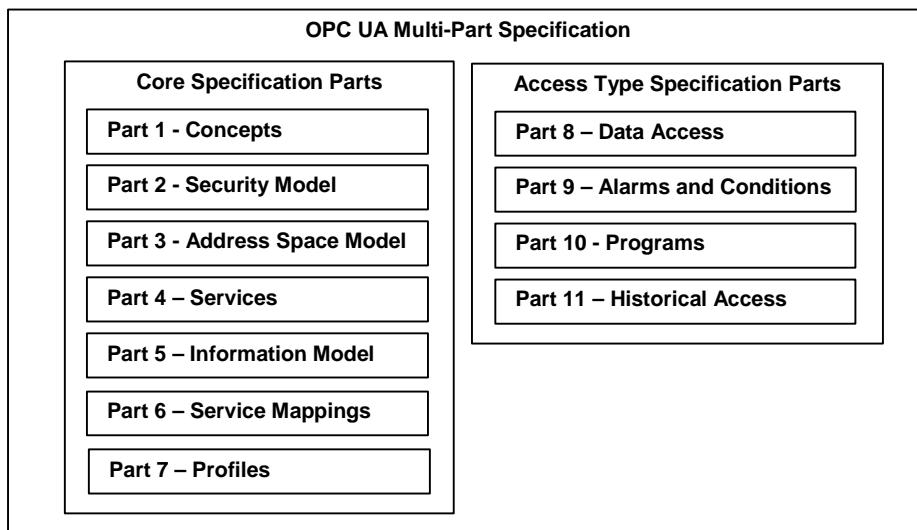
A&E	Alarms and Events
API	Application Programming Interface
COM	Component Object Model
DA	Data Access
DX	Data Exchange
HDA	Historical Data Access
HMI	Human-Machine Interface
MES	Manufacturing Execution System
PLC	Programmable Logic Controller

SCADA	Supervisory Control And Data Acquisition
SOAP	Simple Object Access Protocol
UA	Unified Architecture
UML	Unified Modelling Language
WSDL	Web Services Definition Language
XML	Extensible Mark-up Language

## 4 Structure of the OPC UA series

### 4.1 Specification Organization

This specification is organized as a multi-part specification, as illustrated in Figure 1.



**Figure 1 – OPC UA Specification Organization**

The first seven parts specify the core capabilities of OPC UA. These core capabilities define the structure of the OPC AddressSpace and the Services that operate on it. Parts 8 through 11 apply these core capabilities to specific types of access previously addressed by separate OPC COM specifications, such as Data Access (DA), Alarms and Events (A&E) and Historical Data Access (HDA).

Readers are encouraged to read Parts 1 through 5 of the core specifications before reading Parts 8 through 11. For example, a reader interested in UA Data Access should read Parts 1 through 5 and 8. References in [UA Part 8] may direct the reader to other parts of this specification.

### 4.2 Core Specification Parts

#### 4.2.1 Part 1 – OPC UA Concepts

This specification describes the concepts applicable to OPC UA Servers and *Clients*.

#### 4.2.2 Part 2 – OPC UA Security Model

[UA Part 2] describes the model for securing interactions between OPC UA *Clients* and OPC UA Servers.

#### 4.2.3 Part 3 – OPC UA Address Space Model

[UA Part 3] describes the contents and structure of the Server's AddressSpace.

#### 4.2.4 Part 4 – OPC UA Services

[UA Part 4] specifies the Services provided by OPC UA Servers.

#### 4.2.5 Part 5 – OPC UA Information Model

[UA Part 5] specifies the standard types and their relationships defined for OPC UA Servers.

#### 4.2.6 Part 6 – OPC UA Service Mappings

[UA Part 6] specifies the transport mappings and data encodings supported by OPC UA.

#### 4.2.7 Part 7 – OPC UA Profiles

[UA Part 7] specifies the *Profiles* that are available for OPC *Clients* and *Servers*. These *Profiles* provide groups of *Services* or functionality that can be used for conformance level certification. *Servers* and *Clients* will be tested against the *Profiles*.

### 4.3 Access Type Specification Parts

#### 4.3.1 Part 8 – OPC UA Data Access

[UA Part 8] specifies the use of OPC UA for data access.

#### 4.3.2 Part 9 – OPC UA Alarms and Conditions

[UA Part 9] specifies use of OPC UA support for access to *Alarms* and conditions. The base system includes support for simple *Events*; this specification extends that support to include support for *Alarms* and conditions.

#### 4.3.3 Part 10 – OPC UA Programs

[UA Part 10] specifies OPC UA support for access to *Programs*.

#### 4.3.4 Part 11 – OPC UA Historical Access

[UA Part 11] specifies use of OPC UA for historical access. This access includes both historical data and historical *Events*.

## 5 Overview

### 5.1 Introduction

OPC Unified Architecture (UA) is a platform-independent standard through which various kinds of systems and devices can communicate by sending *Messages* between *Clients* and *Servers* over various types of networks. It supports robust, secure communication that assures the identity of *Clients* and *Servers* and resists attacks. OPC UA defines standard sets of *Services* that *Servers* may provide, and individual *Servers* specify to *Clients* what *Service* sets they support. Information is conveyed using standard and vendor-defined data types, and *Servers* define object models that *Clients* can dynamically discover. *Servers* can provide access to both current and historical data, as well as *Alarms* and *Events* to notify *Clients* of important changes. OPC UA can be mapped onto a variety of communication protocols and data can be encoded in various ways to trade off portability and efficiency.

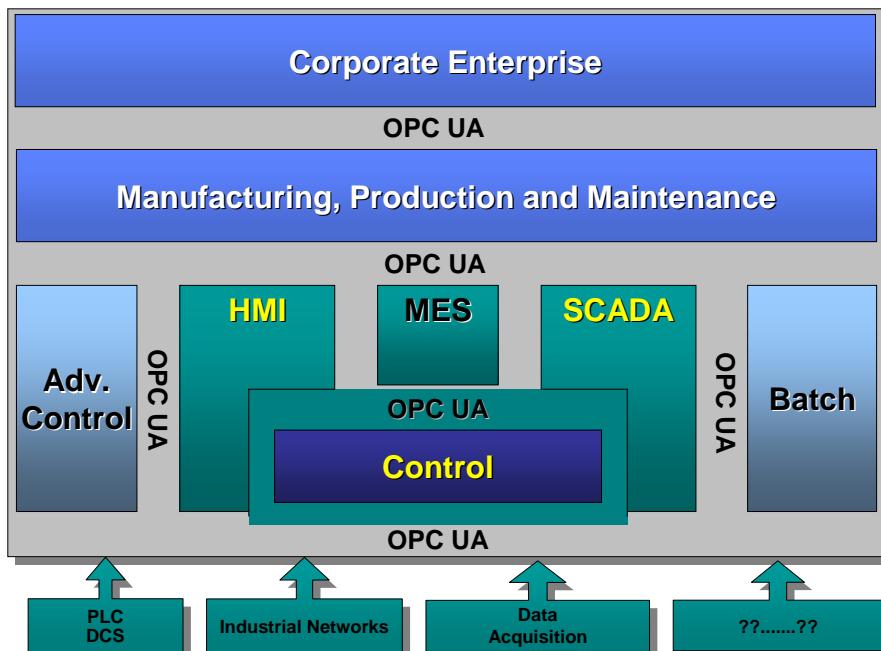
### 5.2 Design goals

OPC UA provides a consistent, integrated *AddressSpace* and service model. This allows a single OPC UA Server to integrate data, *Alarms* and *Events*, and history into its *AddressSpace*, and to provide access to them using an integrated set of *Services*. These *Services* also include an integrated security model.

OPC UA also allows *Servers* to provide *Clients* with type definitions for the *Objects* accessed from the *AddressSpace*. This allows standard information models to be used to describe the contents of

the *AddressSpace*. OPC UA allows data to be exposed in many different formats, including binary structures and XML documents. The format of the data may be defined by OPC, other standard organizations or vendors. Through the *AddressSpace*, *Clients* can query the *Server* for the metadata that describes the format for the data. In many cases, *Clients* with no pre-programmed knowledge of the data formats will be able to determine the formats at runtime and properly utilize the data.

OPC UA adds support for many relationships between *Nodes* instead of being limited to just a single hierarchy. In this way, an OPC UA Server may present data in a variety of hierarchies tailored to the way a set of *Clients* would typically like to view the data. This flexibility, combined with support for type definitions, makes OPC UA applicable to a wide array of problem domains. As illustrated below, OPC UA is not targeted at just the telemetry server interface, but also as a way to provide greater interoperability between higher level functions.



**Figure 2 – OPC UA Target Applications**

OPC UA is designed to provide robustness of published data. A major feature of all OPC servers is the ability to publish data and *Event Notifications*. OPC UA provides mechanisms for *Clients* to quickly detect and recover from communication failures associated with these transfers without having to wait for long timeouts provided by the underlying protocols.

OPC UA is designed to support a wide range of *Servers*, from plant floor PLCs to enterprise Servers. These Servers are characterized by a broad scope of size, performance, execution platforms and functional capabilities. Therefore, OPC UA defines a comprehensive set of capabilities, and Servers may implement a subset of these capabilities. To promote interoperability, OPC UA defines standard subsets, referred to as *Profiles*, to which Servers may claim conformance. *Clients* can then discover the *Profiles* of a Server, and tailor their interactions with that Server based on the *Profiles*. *Profiles* are defined in [UA Part 7].

The OPC UA specifications are layered to isolate the core design from the underlying computing technology and network transport. This allows OPC UA to be mapped to future technologies as necessary, without negating the basic design. Mappings and data encodings are described in [UA Part 6]. Two data encodings are defined in this part:

- XML/text

- UA Binary

In addition, two transport mappings are defined in this part:

- TCP
- SOAP Web services over HTTP

*Clients and Servers* that support multiple transports and encodings will allow the end users to make decisions about tradeoffs between performance and XML Web service compatibility at the time of deployment, rather than having these tradeoffs determined by the OPC vendor at the time of product definition.

OPC UA is designed as the migration path for OPC clients and servers that are based on Microsoft COM technology. Care has been taken in the design of OPC-UA so that existing data exposed by OPC COM servers (DA, HDA and A&E) can easily be mapped and exposed via OPC UA. Vendors may choose to migrate their products natively to OPC UA or use external wrappers to convert from OPC COM to OPC UA and vice-versa. Each of the previous OPC specifications defined its own address space model and its own set of Services. OPC UA unifies the previous models into a single integrated address space with a single set of Services.

### 5.3 Integrated models and services

#### 5.3.1 Security model

##### 5.3.1.1 General

OPC UA security is concerned with the authentication of *Clients and Servers*, the authentication of users, the integrity and confidentiality of their communications, and the verifiability of claims of functionality. It does not specify the circumstances under which various security mechanisms are required. That specification is crucial, but it is made by the designers of the system at a given site and may be specified by other standards.

Rather, OPC UA provides a security model, defined in [UA Part 2], in which security measures can be selected and configured to meet the security needs of a given installation. This model includes standard security mechanisms and parameters. In some cases, the mechanism for exchanging security parameters is defined, but the way that applications use these parameters is not. This framework also defines a minimum set of security *Profiles* that all UA *Servers* must support, even though they may not be used in all installations. Security *Profiles* are defined in [UA Part 7].

##### 5.3.1.2 Session establishment

Application level security relies on a secure communication channel that is active for the duration of the application *Session* and ensures the integrity of all *Messages* that are exchanged. This means users need to be authenticated only once, when the application *Session* is established. The mechanisms for establishing secure communication channels and application *Sessions* are described in [UA Part 4] and [UA Part 6].

When a *Session* is established, the *Client* and *Server* applications negotiate a secure communications channel and exchange software *Certificates* that identify the *Client* and *Server* and the capabilities that they provide. OPC Foundation-generated software *Certificates* indicate the OPC UA *Profiles* that the applications implement and the OPC UA certification level reached for each *Profile*. The details of each *Profile* and the *Certificates* are specified in [UA Part 7]. *Certificates* issued by other organizations may also be exchanged during *Session* establishment.

The *Server* further authenticates the user and authorizes subsequent requests to access *Objects* in the *Server*. Authorization mechanisms, such as access control lists, are not specified by the OPC UA specification. They are application- or system-specific.

### 5.3.1.3 Auditing

*User level security* includes support for security audit trails, with traceability between *Client* and *Server* audit logs. If a security-related problem is detected at the *Server*, the associated *Client* audit log entry can be located and examined. OPC UA also provides the capability for *Servers* to generate *Event Notifications* that report auditable *Events* to *Clients* capable of processing and logging them. OPC UA defines standard security audit parameters that can be included in audit log entries and in audit *Event Notifications*. [UA Part 5] defines the data types for these parameters. Not all *Servers* and *Clients* provide all of the auditing features. *Profiles*, found in [UA Part 7], indicate which features are supported.

### 5.3.1.4 Transport security

OPC UA security complements the security infrastructure provided by most web service capable platforms.

Transport level security can be used to encrypt and sign *Messages*. Encryption and signatures protect against disclosure of information and protect the integrity of *Messages*. Encryption capabilities are provided by the underlying communications technology used to exchange *Messages* between OPC UA applications. [UA Part 7] defines the encryption and signature algorithms to be used for a given *Profile*.

## 5.3.2 Integrated AddressSpace model

The set of *Objects* and related information that the OPC UA *Server* makes available to *Clients* is referred to as its *AddressSpace*. The OPC UA *AddressSpace* represents its contents as a set of *Nodes* connected by *References*.

Primitive characteristics of *Nodes* are described by OPC-defined *Attributes*. *Attributes* are the only elements of a *Server* that have data values. Data types that define attribute values may be simple or complex.

*Nodes* in the *AddressSpace* are typed according to their use and their meaning. *NodeClasses* define the metadata for the OPC UA *AddressSpace*. [UA Part 3] defines the OPC UA *NodeClasses*.

The *Base NodeClass* defines *Attributes* common to all *Nodes*, allowing identification, classification and naming. Each *NodeClass* inherits these *Attributes* and may additionally define its own *Attributes*.

To promote interoperability of *Clients* and *Servers*, the OPC UA *AddressSpace* is structured hierarchically with the top levels standardized for all *Servers*. Although *Nodes* in the *AddressSpace* are typically accessible via the hierarchy, they may have *References* to each other, allowing the *AddressSpace* to represent an interrelated network of *Nodes*. The model of the *AddressSpace* is defined in [UA Part 3].

OPC UA *Servers* may subset the *AddressSpace* into *Views* to simplify *Client* access. Clause 6.3.3.3 describes *AddressSpace Views* in more detail.

## 5.3.3 Integrated object model

The OPC UA Object Model provides a consistent, integrated set of *NodeClasses* for representing *Objects* in the *AddressSpace*. This model represents *Objects* in terms of their *Variables*, *Events* and *Methods*, and their relationships with other *Objects*. [UA Part 3] describes this model.

The OPC UA object model allows *Servers* to provide type definitions for *Objects* and their components. Type definitions may be subclassed. They also may be standardized or they may be system-specific. *ObjectTypes* may be defined by the OPC Foundation, other standards organizations, vendors or end-users.

This model allows data, *Alarms* and *Events*, and their history to be integrated into a single OPC UA Server. For example, OPC UA Servers are able to represent a temperature transmitter as an *Object* that is composed of a temperature value, a set of alarm parameters, and a corresponding set of alarm limits.

#### 5.3.4 Integrated services

The interface between OPC UA *Clients* and *Servers* is defined as a set of *Services*. These *Services* are organized into logical groupings called *Service Sets*. *Service Sets* are discussed in Clause 7 and specified in [UA Part 4].

OPC UA *Services* provide two capabilities to *Clients*. They allow *Clients* to issue requests to *Servers* and receive responses from them. They also allow *Clients* to subscribe to *Servers* for *Notifications*. *Notifications* are used by the *Server* to report occurrences such as *Alarms*, data value changes, *Events*, and *Program* execution results.

OPC UA *Messages* may be encoded as XML text or in binary format for efficiency purposes. They may be transferred using multiple underlying transports, for example TCP or web services over HTTP. *Servers* may provide different encodings and transports as defined by [UA Part 7].

### 5.4 Sessions

OPC UA requires a stateful model. The state information is maintained inside an application *Session*. Examples of state-information are *Subscriptions*, user credentials and continuation points for operations that span multiple requests.

*Sessions* are defined as logical connections between *Clients* and *Servers*. *Servers* may limit the number of concurrent *Sessions* based on resource availability, licensing restrictions, or other constraints. Each *Session* is independent of the underlying communications protocols. Failures of these protocols do not automatically cause the *Session* to terminate. *Sessions* terminate based on *Client* or *Server* request, or based on inactivity of the *Client*. The inactivity time interval is negotiated during *Session* establishment.

### 5.5 Redundancy

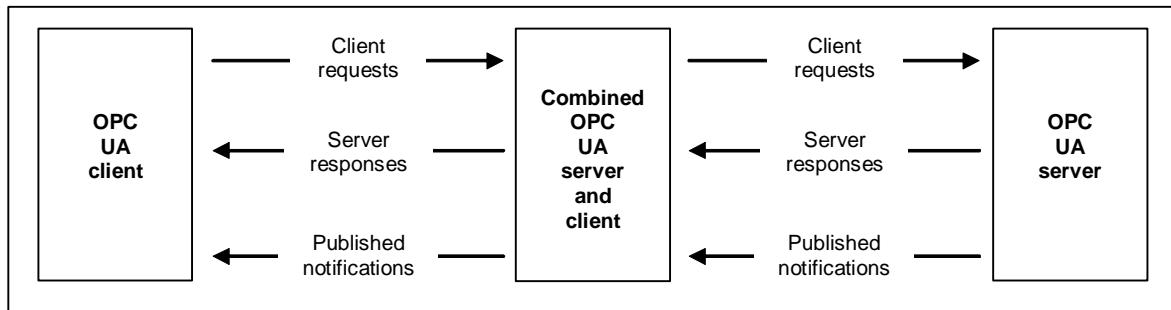
The design of OPC UA ensures that vendors can create redundant *Clients* and redundant *Servers* in a consistent manner. Redundancy may be used for high availability, fault tolerance and load balancing. The details for redundancy are found in [UA Part 4]. Only some *Profiles* [UA Part 7] will require redundancy support, but not the base *Profile*.

## 6 Systems concepts

### 6.1 Overview

The OPC UA systems architecture models OPC UA *Clients* and *Servers* as interacting partners. Each system may contain multiple *Clients* and *Servers*. Each *Client* may interact concurrently with one or more *Servers*, and each *Server* may interact concurrently with one or more *Clients*. An application may combine *Server* and *Client* components to allow interaction with other *Servers* and *Clients* as described in Clause 6.3.6.

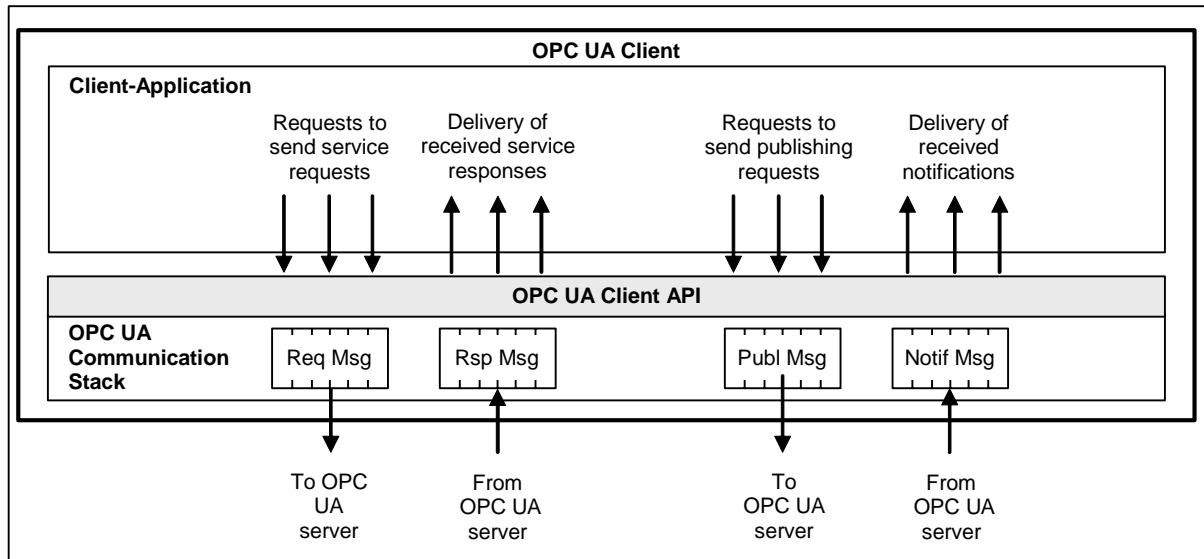
OPC UA *Clients* and *Servers* are described in the clauses that follow. Figure 3 illustrates the architecture that includes a combined *Server* and *Client*.



**Figure 3 – OPC UA System Architecture**

### 6.2 OPC UA Clients

The OPC UA *Client* architecture models the *Client* endpoint of client/server interactions. Figure 4 illustrates the major elements of a typical OPC UA *Client* and how they relate to each other.



**Figure 4 – OPC UA Client Architecture**

The *Client Application* is the code that implements the function of the *Client*. It uses the OPC UA *Client API* to send and receive OPC UA *Service* requests and responses to the OPC UA *Server*. The *Services* defined for OPC UA are described in Clause 7, and specified in [UA Part 4].

Note that the “OPC UA *Client API*” is an internal interface that isolates the *Client* application code from an OPC UA Communication Stack. The OPC UA Communication Stack converts OPC UA *Client API* calls into *Messages* and sends them through the underlying communications entity to the *Server* at the request of the *Client* application. The OPC UA Communication Stack also receives

response and *Notification Messages* from the underlying communications entity and delivers them to the *Client* application through the OPC UA *Client API*.

### 6.3 OPC UA Servers

The OPC UA Server architecture models the *Server* endpoint of client/server interactions. Figure 5 illustrates the major elements of the OPC UA Server and how they relate to each other.

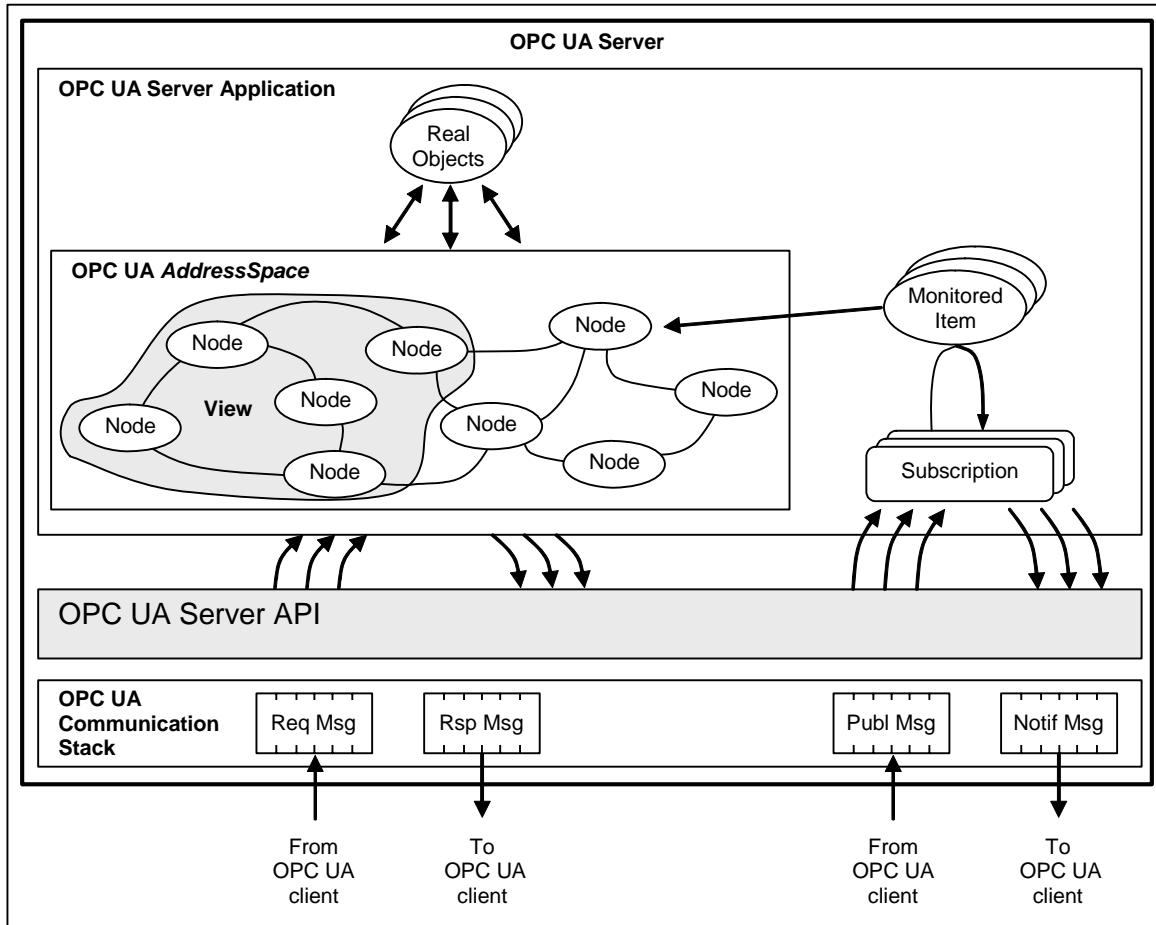


Figure 5 – OPC UA Server Architecture

#### 6.3.1 Real objects

Real objects are physical or software objects that are accessible by the OPC UA Server application or that it maintains internally. Examples include physical devices and diagnostics counters.

#### 6.3.2 OPC UA Server application

The OPC UA Server application is the code that implements the function of the *Server*. It uses the OPC UA Server API to send and receive OPC UA Messages from OPC UA *Clients*. Note that the “OPC UA Server API” is an internal interface that isolates the *Server* application code from an OPC UA Communication Stack. This may be a standard implementation provided by the OPC Foundation or it may be a vendor-specific implementation.

#### 6.3.3 OPC UA AddressSpace

##### 6.3.3.1 AddressSpace Nodes

The *AddressSpace* is modelled as a set of *Nodes* accessible by *Clients* using OPC UA Services (interfaces and methods). *Nodes* in the *AddressSpace* are used to represent real objects, their definitions and their *References* to each other.

### 6.3.3.2 **AddressSpace organization**

[UA Part 3] contains the details of the meta model “building blocks” used to create an *AddressSpace* out of interconnected *Nodes* in a standard, consistent manner. *Servers* are free to organize their *Nodes* within the *AddressSpace* as they choose. The use of *References* between *Nodes* permits *Servers* to organize the *AddressSpace* into hierarchies, a full mesh network of *Nodes*, or any possible mix.

[UA Part 5] defines standard OPC UA *Nodes* and *References* and their expected organization in the *AddressSpace*. Some *Profiles* will not require that all of the standard UA *Nodes* be implemented.

### 6.3.3.3 **AddressSpace Views**

A *View* is a subset of the *AddressSpace*. *Views* are used to restrict the *Nodes* that the *Server* makes visible to the *Client*, thus restricting the size of the *AddressSpace* for the *Service* requests submitted by the *Client*. The default *View* is the entire *AddressSpace*. *Servers* may optionally define other *Views*. *Views* hide some of the *Nodes* or *References* in the *AddressSpace*. *Views* are visible via the *AddressSpace* and *Clients* are able to browse *Views* to determine their structure. *Views* are often hierarchies, which are easier for *Clients* to navigate and represent in a tree.

### 6.3.3.4 **Support for information models**

The OPC UA *AddressSpace* supports information models. This support is provided through:

- a) *Node References* that allow *Objects* in the *AddressSpace* to be related to each other.
- b) *ObjectType Nodes* that provide semantic information for real *Objects* (type definitions).
- c) *ObjectType Nodes* to support subclassing of type definitions.
- d) Data type definitions exposed in the *AddressSpace* that allow industry specific data types to be used.
- e) OPC UA companion standards that permit industry groups to define how their specific information models are to be represented in OPC UA *Server AddressSpaces*.

### 6.3.4 **Publisher/subscriber entities**

#### 6.3.4.1 **MonitoredItems**

*MonitoredItems* are entities in the *Server* created by the *Client* that monitor *AddressSpace* *Nodes* and their real-world counterparts. When they detect a data change or an event/alarm occurrence, they generate a *Notification* that is transferred to the *Client* by a *Subscription*.

#### 6.3.4.2 **Subscriptions**

A *Subscription* is an endpoint in the *Server* that publishes *Notifications* to *Clients*. *Clients* control the rate at which publishing occurs by sending *Publish Messages*.

### 6.3.5 **OPC UA Service Interface**

#### 6.3.5.1 **General**

The Services defined for OPC UA are described in Clause 7, and specified in [UA Part 4].

#### 6.3.5.2 **Request/response Services**

Request/response *Services* are *Services* invoked by the *Client* through the OPC UA *Service Interface* to perform a specific task on one or more *Nodes* in the *AddressSpace* and to return a response.

### 6.3.5.3 Publisher Services

Publisher Services are Services invoked through the OPC UA Service Interface for the purpose of periodically sending *Notifications* to *Clients*. Notifications include *Events*, *Alarms*, data changes and *Program* outputs.

### 6.3.6 Server to Server interactions

Server to Server interactions are interactions in which one Server acts as a *Client* of another Server. Server to Server interactions allow for the development of servers that:

- a) exchange information with each other on a peer-to-peer basis, this could include redundancy or remote Servers that are used for maintaining system wide type definitions,
- b) are chained in a layered architecture of Servers to provide:
  - 1) aggregation of data from lower-layer Servers,
  - 2) higher-layer data constructs to *Clients*, and
  - 3) concentrator interfaces to *Clients* for single points of access to multiple underlying Servers.

Figure 6 illustrates interactions between Servers.

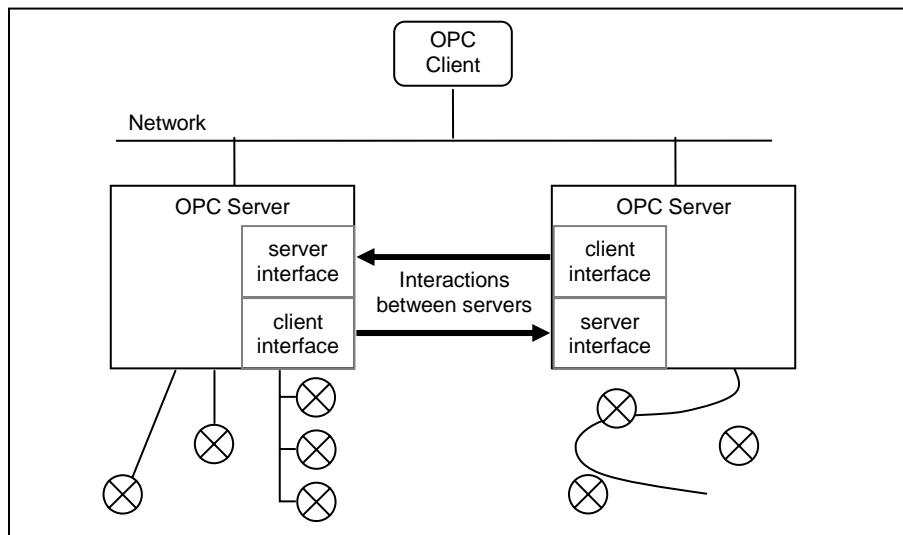
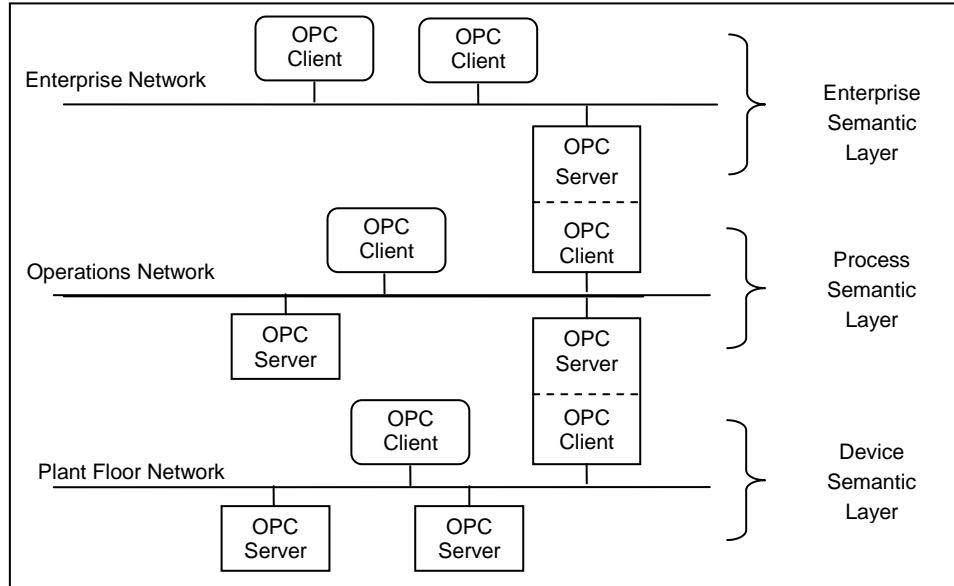


Figure 6 – Interactions between Servers

Figure 7 extends the previous example and illustrates the chaining of OPC UA Servers together for vertical access to data in an enterprise.



**Figure 7 – Chained Server Example**

## 7 Service Sets

### 7.1 General

OPC UA Services are divided into *Service Sets*, each defining a logical grouping of Services used to access a particular aspect of the Server. The *Service Sets* are described below. The *Service Sets* and their *Services* are specified in [UA Part 4]. Whether or not a Server supports a *Service Set*, or a specific *Service* within a *Service Set* is defined by its *Profile*. *Profiles* are described in [UA Part 7].

### 7.2 SecureChannel Service Set

This *Service Set* defines *Services* used to discover the security configuration of a Server and to open a communication channel that ensures the confidentiality and integrity of all *Messages* exchanged with the Server. The base concepts for UA security are defined in [UA Part 2].

The *SecureChannel Services* are unlike other *Services* because they are typically not implemented by the *UA application* directly. Instead, they are provided by the communication stack that the *UA application* is built on. For example, a UA Server may be built on a SOAP stack that allows applications to establish a *SecureChannel* using the WS-SecureConversation specification. In these cases, the *UA application* simply needs to verify that a WS-SecureConversation is active whenever it receives a *Message*. [UA Part 6] describes how the *SecureChannel Services* are implemented.

A *SecureChannel* is a long-running logical connection between a single *Client* and a single *Server*. This channel maintains a set of keys that are known only to the *Client* and *Server* and that are used to authenticate and encrypt *Messages* sent across the network. The *SecureChannel Services* allow the *Client* and *Server* to securely negotiate the keys to use.

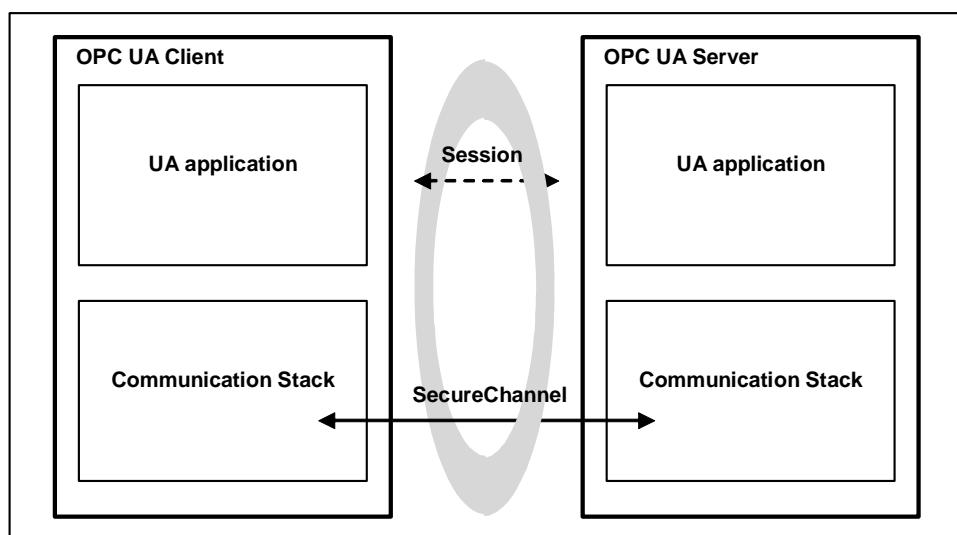
The exact algorithms used to authenticate and encrypt *Messages* are described in the security policies for a *Server*. A *Client* must select one of the security policies supported by the *Server* when it creates a *SecureChannel*.

When a *Client* and *Server* are communicating via a *SecureChannel* they must verify that all incoming *Messages* have been signed and/or encrypted according to the security policy. A UA

application must not process any *Message* that does not conform to the security policy for the channel.

A *SecureChannel* is separate from the *UA Application Session*; however, a single *UA Application Session* may only be accessed via a single *SecureChannel*. This implies that the *UA application* must be able to determine what *SecureChannel* is associated with each *Message*. A communication stack that provides a *SecureChannel* mechanism but that does not allow the application to know what *SecureChannel* was used for a given *Message* cannot be used to implement the *SecureChannel Service Set*.

The relationship between the *UA Application Session* and the *SecureChannel* is illustrated in Figure 8. The UA applications use the communication stack to exchange *Messages*. First, the *SecureChannel Services* are used to establish a *SecureChannel* between the two communication stacks, allowing them to exchange *Messages* in a secure way. Second, the UA applications use the *Session Service Set* to establish a *UA application Session*.



**Figure 8 – *SecureChannel* and *Session Services***

When a *Client* establishes a *SecureChannel* it may provide a user identity. This user identity may be different from the user identity provided when the *Client* opens the *UA Application Session*.

### 7.3 *Session Service Set*

This Service Set defines Services used to establish an application-layer connection in the context of a *Session* on behalf of a specific user.

### 7.4 *NodeManagement Service Set*

The *NodeManagement Service Set* allows *Clients* to add, modify, and delete *Nodes* in the *AddressSpace*. These Services provide a standard interface for the configuration of *Servers*.

### 7.5 *View Service Set*

Views are publicly defined, *Server-created* subsets of the *AddressSpace*. The entire *AddressSpace* is the default *View*, and therefore, the *View Services* are capable of operating on the entire *AddressSpace*. Future versions of this specification may also define Services to create *Client* defined *Views*.

The *View Service Set* allows *Clients* to discover *Nodes* in a *View* by browsing. Browsing allows *Clients* to navigate up and down the hierarchy, or to follow *References* between *Nodes* contained in the *View*. In this manner, browsing also allows *Clients* to discover the structure of the *View*.

## 7.6 Attribute Service Set

The *Attribute Service Set* is used to read and write *Attribute* values. *Attributes* are primitive characteristics of *Nodes* that are defined by OPC UA. They may not be defined by *Clients* or *Servers*. *Attributes* are the only elements in the *AddressSpace* permitted to have data values. A special *Attribute*, the *Value Attribute* is used to define the value of *Variables*.

## 7.7 Method Service Set

*Methods* represent the function calls of *Objects*. They are defined in [UA Part 3]. *Methods* are invoked and return after completion, whether successful or unsuccessful. Execution times for *Methods* may vary, depending on the function they are performing.

The *Method Service Set* defines the means to invoke *Methods*. A *Method* must be a component of an *Object*. Discovery is provided through the browse and query *Services*. *Clients* discover the *Methods* supported by a *Server* by browsing for the owning *Objects* that identify their supported *Methods*.

Because *Methods* may control some aspect of plant operations, method invocation may depend on environmental or other conditions. This may be especially true when attempting to re-invoke a *Method* immediately after it has completed execution. Conditions that are required to invoke the *Method* may not yet have returned to the state that permits the *Method* to start again. In addition, some *Methods* may be capable of supporting concurrent invocations, while others may have a single invocation executing at a given time.

## 7.8 MonitoredItem Service Set

The *MonitoredItem Service Set* is used by the *Client* to create and maintain *MonitoredItems*. *MonitoredItems* monitor *Variables*, *Attributes* and *EventNotifiers*. They generate *Notifications* when they detect certain conditions. They monitor *Variables* for a change in value, status or timestamp; *Attributes* for a change in value; and *EventNotifiers* for newly generated *Alarm* and *Event* reports.

Each *MonitoredItem* identifies the item to monitor and the *Subscription* to use to periodically publish *Notifications* to the *Client* (see Clause 7.9). Each *MonitoredItem* also specifies the rate at which the item is to be monitored (sampled) and, for *Variables* and *EventNotifiers*, the filter criteria used to determine when a *Notification* is to be generated. Filter criteria for *Attributes* are specified by their *Attribute* definitions in [UA Part 4].

The sample rate defined for a *MonitoredItem* may be faster than the publishing rate of the *Subscription*. For this reason, the *MonitoredItem* may be configured to either queue all *Notifications* or to queue only the latest *Notification* for transfer by the *Subscription*. In this latter case, the queue size is one.

*MonitoredItem Services* also define a monitoring mode. The monitoring mode is configured to disable sampling and reporting, to enable sampling only, or to enable both sampling and reporting. When sampling is enabled, the *Server* samples the item. In addition, each sample is evaluated to determine if a *Notification* should be generated. If so, the *Notification* is queued. If reporting is enabled, the queue is made available to the *Subscription* for transfer.

Finally, *MonitoredItems* can be configured to trigger the reporting of other *MonitoredItems*. In this case, the monitoring mode of the items to report is typically set to sampling only, and when the triggering item generates a *Notification*, any queued *Notifications* of the items to report are made available to the *Subscription* for transfer.

## 7.9 Subscription Service Set

The *Subscription Service Set* is used by the *Client* to create and maintain *Subscriptions*. *Subscriptions* are entities that periodically publish *Notification Messages* for the *MonitoredItem* assigned to them (see Clause 7.7). The *Notification Message* contains a common header followed

by a series of *Notifications*. The format of *Notifications* is specific to the type of item being monitored (i.e. *Variables*, *Attributes*, and *EventNotifiers*).

Once created, the existence of a *Subscription* is independent of the *Client's Session* with the *Server*. This allows one *Client* to create a *Subscription*, and a second, possibly a redundant *Client*, to receive *Notification Messages* from it.

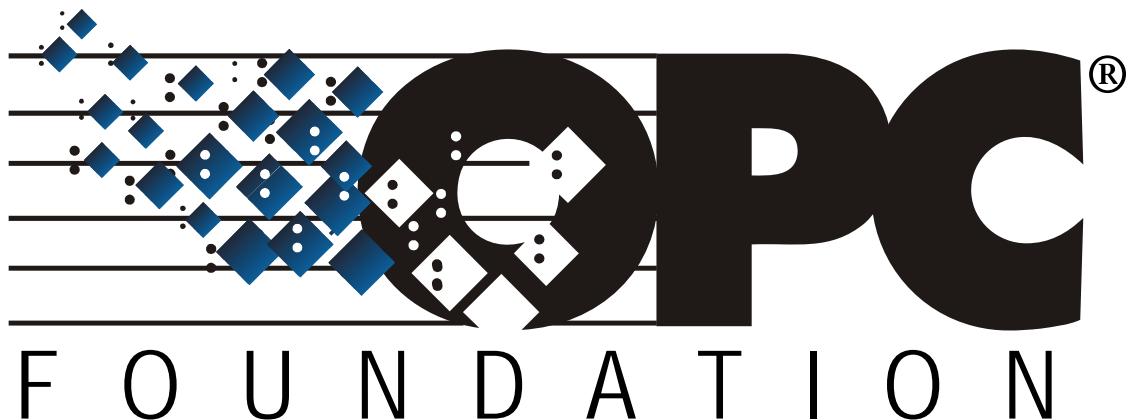
To protect against non-use by *Clients*, *Subscriptions* have a configured lifetime that *Clients* periodically renew. If any *Client* fails to renew the lifetime, the lifetime expires and the *Subscription* is closed by the *Server*. When a *Subscription* is closed, all *MonitoredItems* assigned to the *Subscription* are deleted.

*Subscriptions* include features that support detection and recovery of lost *Messages*. Each *Notification Message* contains a sequence number that allows *Clients* to detect missed *Messages*. When there are no *Notifications* to send within the keep-alive time interval, the *Server* sends a keep-alive *Message* that contains the sequence number of the last *Notification Message* sent. If a *Client* fails to receive a *Message* after the keep-alive interval has expired, or if it determines that it has missed a *Message*, it can request the *Server* to resend one or more *Messages*.

## 7.10 Query Service Set

The *Query Service Set* allows *Clients* to select a subset of the *Nodes* in the address space or in a *View* based on some *Client*-provided filter criteria. The *Nodes* and their *Attributes* selected by the *Query Service* are called a result set. The result set may be a “flat” set of *Nodes*, or it may preserve the structure by returning *References* connecting the *Nodes* in the result set.

*Servers* may find it difficult to process queries that require access to runtime data, such as device data, that involves resource intensive operations or significant delays. In these cases, the *Server* may find it necessary to reject the query.



# **OPC Unified Architecture**

## **Specification**

### **Part 2: Security Model**

**Version 1.00**

**July 28, 2006**

Specification Type	<u>Industry Standard Specification</u>		
Title:	OPC Unified Architecture	Date:	July 28, 2006
<u>Security Model</u>			
Version:	<u>Release 1.00</u>	Software Source:	MS-Word OPC UA Part 2 - Security Model 1.00 Specification.doc
Author:	<u>OPC Foundation</u>	Status:	<u>Release</u>

## CONTENTS

	Page
<u>FOREWORD</u> .....	vii
<u>AGREEMENT OF USE</u> .....	vii
1 Scope .....	1
2 Reference documents .....	1
2.1 References .....	1
3 Terms, definitions, and abbreviations .....	2
3.1 OPC UA Part 1 terms .....	2
3.2 OPC UA Security terms.....	2
3.2.1 Authentication .....	2
3.2.2 Authorization.....	3
3.2.3 Confidentiality .....	3
3.2.4 Integrity.....	3
3.2.5 Auditing .....	3
3.2.6 Non-Repudiation .....	3
3.2.7 Availability.....	3
3.2.8 Cyber Security Management System (CSMS).....	3
3.2.9 OPC UA Application .....	3
3.2.10 Application Instance .....	3
3.2.11 Application Instance Certificate.....	3
3.2.12 X.509 Certificate .....	3
3.2.13 Digital Certificate.....	4
3.2.14 Security Token .....	4
3.2.15 Secure Channel.....	4
3.2.16 Nonce .....	4
3.2.17 Asymmetric Cryptography .....	4
3.2.18 Hash Function .....	4
3.2.19 Public Key Infrastructure (PKI).....	4
3.2.20 Symmetric Cryptography.....	5
3.2.21 Message Authentication Code (MAC).....	5
3.2.22 Hashed Message Authentication Code (HMAC).....	5
3.2.23 PublicKey .....	5
3.2.24 PrivateKey .....	5
3.3 Abbreviations and symbols.....	5
3.4 Conventions.....	5
3.4.1 Conventions for security model figures.....	5
4 OPC UA Security architecture .....	6
4.1 OPC UA Security Environment .....	6
4.2 Security Objectives .....	7
4.2.1 Overview.....	7
4.2.2 Authentication .....	7
4.2.3 Authorization .....	7
4.2.4 Confidentiality .....	7
4.2.5 Integrity.....	7
4.2.6 Auditability .....	7
4.2.7 Availability.....	7
4.3 Security Threats to OPC UA Systems .....	8

4.3.1	Overview .....	8
4.3.2	Message Flooding .....	8
4.3.3	Eavesdropping .....	8
4.3.4	Message Spoofing.....	8
4.3.5	Message Alteration.....	9
4.3.6	Message Replay.....	9
4.3.7	Malformed Messages.....	9
4.3.8	Server Profiling .....	9
4.3.9	Session Hijacking .....	9
4.3.10	Rogue Server.....	9
4.3.11	Compromising User Credentials.....	10
4.4	OPC UA Relationship to Site Security.....	10
4.5	OPC UA Security Architecture .....	11
4.6	Policy .....	12
4.7	Profile.....	12
4.8	User Authorization .....	13
4.9	User Authentication.....	13
4.10	OPC UA Security Services .....	13
4.11	Auditing .....	13
4.11.1	Single client and server .....	14
4.11.2	Aggregating server .....	15
4.11.3	Aggregation through a non-auditing server.....	16
4.11.4	Aggregating server with service distribution .....	17
5	Security Reconciliation.....	18
5.1	Reconciliation of Threats with OPC UA Functions .....	18
5.1.1	Overview .....	18
5.1.2	Message Flooding .....	18
5.1.3	Eavesdropping .....	18
5.1.4	Message Spoofing.....	18
5.1.5	Message Alteration.....	19
5.1.6	Message Replay.....	19
5.1.7	Malformed Messages.....	19
5.1.8	Server Profiling .....	19
5.1.9	Session Hijacking .....	19
5.1.10	Rogue Server.....	19
5.1.11	Compromising User Credentials.....	19
5.2	Reconciliation of Objectives with OPC UA Functions.....	19
5.2.1	Overview .....	19
5.2.2	Authentication .....	20
5.2.3	Authorization .....	20
5.2.4	Confidentiality .....	20
5.2.5	Integrity.....	20
5.2.6	Auditability .....	20
5.2.7	Availability.....	21
6	Implementation considerations .....	21
6.1	Application Instance Certificates.....	21
6.2	Appropriate Timeouts:.....	21
6.3	Strict Message Processing .....	21
6.4	Robust Error recovery .....	22

6.5	Random Number Generation .....	22
6.6	Special and Reserved Packets .....	22
6.7	Rate Limiting and Flow Control.....	22
7	Site Recommendations.....	22

**FIGURES**

Figure 1 - OPC UA Network Model .....	6
Figure 2 – OPC UA Security Architecture.....	11
Figure 3 – Simple Servers .....	14
Figure 4 – Aggregating Servers .....	15
Figure 5 – Aggregation with a Non-Auditing Server .....	16
Figure 6 – Aggregate server with Service Distribution .....	17

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA Applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these

materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>

## 1 Scope

This specification describes the OPC Unified Architecture security model. It describes the security threats of the physical, hardware, and software environments in which UA is expected to run. It describes how UA relies upon other standards for security. It gives an overview of the security features that are specified in other parts of the OPC UA specification. It references services, mappings, and profiles that are specified normatively in other parts of this multi-part specification. This part of the specification is informative rather than normative. Any seeming ambiguity between this part and one of the normative parts does not remove or reduce the requirement specified in the normative part.

This Part 2 is directed to readers who will develop OPC UA client or server applications or implement the OPC UA services layer.

## 2 Reference documents

### 2.1 References

[UA Part 1] OPC UA Specification: Part 1 – Concepts

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 6] OPC UA Specification: Part 6 – Mappings

<http://www.opcfoundation.org/UA/Part6/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles

<http://www.opcfoundation.org/UA/Part7/>

[SOAP Part 1] SOAP Version 1.2 Part 1: Messaging Framework

<http://www.w3.org/TR/soap12-part1/>

[SOAP Part 2] SOAP Version 1.2 Part 2: Adjuncts

<http://www.w3.org/TR/soap12-part2/>

[XML Encryption] XML Encryption Syntax and Processing

<http://www.w3.org/TR/xmlenc-core/>

[XML Signature] XML-Signature Syntax and Processing

<http://www.w3.org/TR/xmldsig-core/>

[WS Security] SOAP Message Security 1.1

<http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>

[WS Addressing] Web Services Addressing (WS-Addressing)

<http://www.w3.org/Submission/ws-addressing/>

[WS Trust] Web Services Trust Language (WS-Trust)

<http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf>

[WS Secure Conversation] Web Services Secure Conversation Language (WS-SecureConversation)

<http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf>

[SSL/TLS] RFC 2246: The TLS Protocol Version 1.0

<http://www.ietf.org/rfc/rfc2246.txt>

[X509] X.509 Public Key Certificate Infrastructure

<http://www.itu.int/rec/T-REC-X.509-200003-I/e>

[HTTP] RFC 2616: Hypertext Transfer Protocol - HTTP/1.1

<http://www.ietf.org/rfc/rfc2616.txt>

[HTTPS] RFC 2818: HTTP Over TLS

<http://www.ietf.org/rfc/rfc2818.txt>

[IS Glossary] Internet Security Glossary

<http://www.ietf.org/rfc/rfc2828.txt>

[NIST 800-12] Introduction to Computer Security

<http://csrc.nist.gov/publications/nistpubs/800-12/>

[NERC CIP] CIP 002-1 through CIP 009-1, by North-American Electric Reliability Council

[IEC 62351] Data and Communications Security

[SPP-ICS] System Protection Profile – Industrial Control System, by Process Control Security Requirements Forum (PCSRF)

It is assumed that the reader is familiar with Web Services and XML/SOAP. Information on these technologies can be found in [SOAP Part 1] and [SOAP Part 2].

### 3 Terms, definitions, and abbreviations

#### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) certificate
- 2) command
- 3) event
- 4) message
- 5) notification
- 6) profile
- 7) subscription

#### 3.2 OPC UA Security terms

##### 3.2.1 Authentication

*Authentication* is the process of verifying the identity of an entity such as a client, server, or user.

### **3.2.2 Authorization**

An *authorization* is a right or a permission that is granted to a system entity to access a system resource.

### **3.2.3 Confidentiality**

*Confidentiality* is the protection of data from being read by unintended parties.

### **3.2.4 Integrity**

*Integrity* is the process of assuring that a message is received as sent, i.e. that it was not modified in transit.

### **3.2.5 Auditing**

*Auditing* is the tracking of actions and activities in the system, including security related activities. Audit records can be used to verify the operation of system security.

### **3.2.6 Non-Repudiation**

*Non-repudiation* prevents either the sender or the receiver from denying that it sent or received a transmitted message.

### **3.2.7 Availability**

*Availability* is the running of the system with unimpeded capacity.

### **3.2.8 Cyber Security Management System (CSMS)**

A CSMS is a program designed by an organization to maintain the security of the entire organization's assets to an established level of *confidentiality*, *integrity*, and *availability*, whether they are on the business side or the industrial automation and control systems side of the organization

### **3.2.9 OPC UA Application**

An *OPC UA Application* is an OPC UA *Client*, which calls OPC UA services, or an OPC UA *Server*, which performs those services.

### **3.2.10 Application Instance**

An *Application Instance* is an individual copy, such as an executable, of a program running on one computer. There can be several *Application Instances* of the same application product running at the same time on several computers or possibly the same computer.

### **3.2.11 Application Instance Certificate**

An *Application Instance Certificate* is a *digital certificate* of an individual instance of an application that has been installed in an individual host. Different installations of one software product would have different application certificates.

### **3.2.12 X.509 Certificate**

An X.509 public-key certificate is a public-key certificate in one of the formats defined by X.509 v1, 2, or 3. An [X509] public-key certificate contains a sequence of data items and has a digital signature computed on that sequence.

### 3.2.13 Digital Certificate

A *digital certificate* is a structure that associates an identity with an entity such as a user or *Application Instance*. The certificate has an associated asymmetric key pair which can be used to authenticate that the entity does, indeed, possess the *private key*.

### 3.2.14 Security Token

A *security token* is an identifier for a cryptographic key set. All *security tokens* belong to a security context.

### 3.2.15 Secure Channel

A *secure channel* in OPC UA is a communication path established between an OPC UA client and server that have authenticated each other using certain OPC UA services and for which security parameters have been negotiated and applied.

### 3.2.16 Nonce

A *nonce* is a random number that is used once and then assigned a different value before the next use. Nonces are often used in algorithms that generate security keys or in a challenge/response test.

### 3.2.17 Asymmetric Cryptography

*Asymmetric cryptography*, also known as "public-key cryptography", employs a pair of numeric keys. One of these keys, the *private key*, is known only by the owner of the key pair. The other key, the *public key*, is known by all correspondents of the *private key* owner.

For encryption: In an asymmetric encryption algorithm, such as RSA, when Alice wants to ensure *confidentiality* for data she sends to Bob, she encrypts the data with a *public key* provided by Bob. Only Bob has the matching *private key* that is needed to decrypt the data.

For signature: In an asymmetric digital signature algorithm, such as DSA, when Alice wants to ensure data *integrity* or provide *authentication* for data she sends to Bob, she uses her *private key* to sign the data. To verify the signature, Bob uses the matching *public key* that Alice has provided.

For key agreement: In an asymmetric key agreement algorithm, such as Diffie-Hellman, Alice and Bob each send their own *public key* to the other person. Then each uses their own *private key* and the other's *public key* to compute the new key value. [IS Glossary]

### 3.2.18 Hash Function

A cryptographic *hash function*, is an algorithm for which it is computationally infeasible to find either a data object that maps to a given hash result (the "one-way" property) or two data objects that map to the same hash result (the "collision-free" property). [IS Glossary]

### 3.2.19 Public Key Infrastructure (PKI)

A *PKI* is the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke *digital certificates* based on asymmetric cryptography. The core *PKI* functions are to register users and issue their public-key certificates, to revoke certificates when required, and to archive data needed to validate certificates at a much later time. Key pairs for data *confidentiality* may be generated by a certificate authority (CA), but requiring a *private key* owner to generate its own key pair improves security because the *private key* would never be transmitted. [IS Glossary]

### 3.2.20 Symmetric Cryptography

*Symmetric cryptography* employs a single “shared” or “secret” key to encrypt messages. *Symmetric cryptography* has a key management disadvantage compared to asymmetric cryptography because of the risk of the key being obtained by an unintended party when it is shared. [IS Glossary]

### 3.2.21 Message Authentication Code (MAC)

A *MAC* is a short piece of data that results from an algorithm that uses a secret key (see *Symmetric Cryptography*) to hash a message. The receiver of the message can check against alteration of the message by computing a *MAC* that should be identical using the same message and secret key.

### 3.2.22 Hashed Message Authentication Code (HMAC)

An *HMAC* is a *MAC* that has been generated using an iterative hash algorithm such as MD5 or SHA.

### 3.2.23 PublicKey

A *public key* is the publicly-disclosable component of a pair of cryptographic keys used for asymmetric cryptography. [IS Glossary]

### 3.2.24 PrivateKey

A *private key* is the secret component of a pair of cryptographic keys used for asymmetric cryptography.

## 3.3 Abbreviations and symbols

CSMS	Cyber Security Management System
DSA	Digital Signature Algorithm
DX	Data Exchange
PKI	Public Key Infrastructure
RSA	public key algorithm for signing or encryption, Rivest, Shamir, Adleman
SHA1	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UA	Unified Architecture
UML	Unified Modeling Language
WSDL	Web Services Definition Language
XML	Extensible Mark-up Language

## 3.4 Conventions

### 3.4.1 Conventions for security model figures

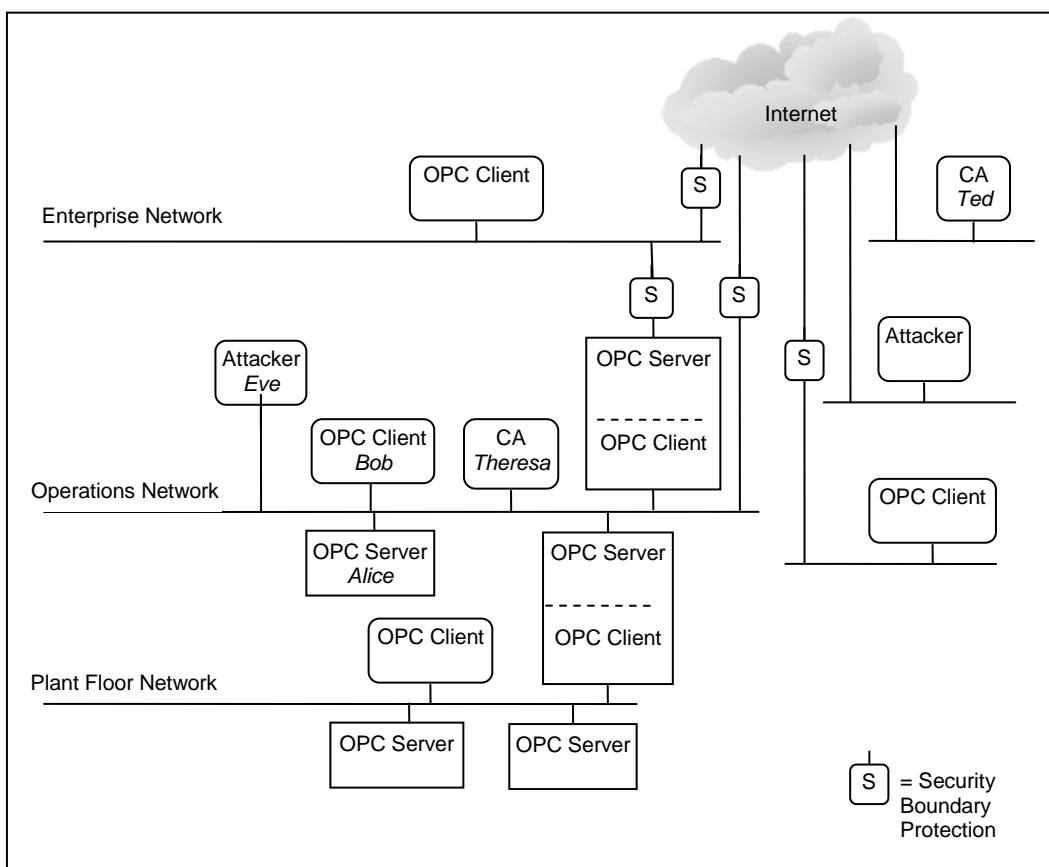
The figures in this document do not use any special common conventions. Any conventions used in a particular figure are explained for that figure.

## 4 OPC UA Security architecture

### 4.1 OPC UA Security Environment

OPC-UA is an interface used between components in the operation of an industrial facility at multiple levels: from high-level enterprise management to low-level direct process control of a device. The use of OPC-UA for enterprise management involves dealings with customers and suppliers. It may be an attractive target for industrial espionage or sabotage and may also be exposed to threats through untargeted malware, such as worms, circulating on public networks. Disruption of communications at the process control end causes at least an economic cost to the enterprise and can have employee and public safety consequences or cause environmental damage. This may be an attractive target for those who seek to harm the enterprise or society.

OPC-UA will be deployed in a diverse range of operational environments, with varying assumptions about threats and accessibility, and with a variety of security policies and enforcement regimes. It must, therefore, provide a flexible set of security mechanisms. Figure 1 is a composite that shows a combination of such environments. Some OPC UA clients and servers are on the same host and can be more easily protected from external attack. Some clients and servers are on different hosts in the same operations network and might be protected by the security boundary protections that separate the operations network from external connections. Some *OPC UA Applications* run in relatively open environments where users and applications might be difficult to control. Other applications are embedded in control systems that have no direct electronic connection to external systems.



**Figure 1 - OPC UA Network Model**

OPC UA may run at different places in this wide range of environments. Client and server may communicate within the same host or between hosts within the highly protected control network. Alternatively, OPC UA clients and servers may communicate in the much more open environment over the Internet. The OPC UA activities are protected by whatever security controls the site provides to the parts of the system within which OPC UA runs.

## 4.2 Security Objectives

### 4.2.1 Overview

Fundamentally, information system security reduces the damage from attacks. It does this by identifying the threats to the system, identifying the system's vulnerabilities to these threats, and providing countermeasures. The countermeasures reduce vulnerabilities directly, counteract threats, or recover from successful attacks.

Industrial automation system security is achieved by meeting a set of objectives. These objectives have been refined through many years of experience in providing security for information systems in general and they remain quite constant despite the ever-changing set of threats to systems. They are described in the following subsections of Section 4.2. Following the sections that describe the OPC UA security architecture and functions, Section 5.2 reconciles these objectives against the OPC UA functions.

### 4.2.2 Authentication

Entities such as clients, servers, and users should prove their identities. This *authentication* can be based on something the entity is, has, or knows.

### 4.2.3 Authorization

The access to read, write, or execute resources should be authorized for only those entities that have a need for that access within the requirements of the system. *Authorization* can be as coarse-grained as allowing or disallowing a client to call a server or it could be much finer grained, such as allowing specific actions on specific information items by specific users.

*Authorization* must be supported by identification and *authentication* of the parties.

### 4.2.4 Confidentiality

Data must be protected from passive attacks, such as eavesdropping, whether the data is being transmitted, in memory, or being stored. To provide *confidentiality* data encryption algorithms are used.

### 4.2.5 Integrity

Receivers must receive the same information that the sender sent, without the data being changed during transmission. Message *integrity* is threatened by message spoofing, message alteration, message replay. Session *integrity* is threatened by session hijacking.

### 4.2.6 Auditability

The use of the system must be checked to determine that the security measures are effective. Audits are done rigorously in order to provide evidence of secure operation to stakeholders. The system supports *auditing* by recording events that are evidence of security working both well and poorly. These events include new connections, configuration changes, and security error responses to service calls.

### 4.2.7 Availability

*Availability* is impaired when the execution of software that needs to run is turned off or when software or the communication system is overwhelmed processing input. Impaired *availability* in OPC UA can appear as slowing down of subscription performance or inability to add sessions for example.

## 4.3 Security Threats to OPC UA Systems

### 4.3.1 Overview

OPC UA provides countermeasures to resist the threats to the security of the information that is communicated. The following subsections of Section 4.3 list the currently known threats to environments in which OPC UA will be deployed. Following the sections that describe the OPC UA security architecture and functions, Section 5.1 reconciles these threats against the OPC UA functions.

### 4.3.2 Message Flooding

An attacker can send a large volume of messages, or a single message that contains a large number of requests, with the goal of overwhelming the OPC server or components on which the OPC server may depend for reliable operation such as CPU, TCP/IP stack, Operating System, or the file system. Flooding attacks can be conducted at multiple layers including OPC-UA, SOAP, [HTTP], or TCP.

Message flooding attacks can use both well-formed and malformed messages. In the first scenario the attacker could be a malicious person using a legitimate client to flood the server with requests. Two cases exist, one in which the client does not have a session with the server and one in which it does.

Message flooding may impair the ability to establish OPC-UA sessions, or terminate an existing session. More generally message flooding may impair the ability to communicate with an OPC-UA entity and result in denial of service.

Message flooding impacts *availability*.

### 4.3.3 Eavesdropping

Eavesdropping is the unauthorized disclosure of sensitive information that might result directly in a critical security breach or be used in follow-on attacks.

If an attacker has compromised the underlying operating system or the network infrastructure, the attacker might record and capture messages. It may be beyond the capability of a client or server to recover from a compromise of the operating system.

Eavesdropping impacts *confidentiality* directly and threatens all of the other security objectives indirectly.

### 4.3.4 Message Spoofing

An attacker may forge messages from the client or server. Spoofing may occur at multiple layers in the protocol stack. This threat does not include taking over a session, which is described in section 4.3.9 as "Session Hijacking".

By spoofing messages from the client or server, attackers may perform unauthorized operations and avoid detection of their activities.

Message spoofing impacts *integrity* and *authorization*.

#### 4.3.5 Message Alteration

Network traffic and application layer messages may be captured, modified, and the modified message sent forward to OPC clients and servers. Message alteration allows illegitimate access to a system.

Message alteration impacts *integrity* and *authorization*.

#### 4.3.6 Message Replay

Network traffic and valid application layer messages may be captured and resent to OPC clients and servers at a later stage without modification. An attacker could misinform the user or send in improper command such a command to open a valve but at an improper time.

Message replay impacts *integrity* and *authorization*.

#### 4.3.7 Malformed Messages

An attacker can craft a variety of messages with invalid message structure (malformed XML, SOAP, malicious binary encodings, etc.) or data values and send them to OPC-UA clients or servers.

The OPC client or server may incorrectly handle certain malformed messages by performing unauthorized operations or disclosing unnecessary information. It might result in a denial or degradation of service including termination of the application or, in the case of embedded devices, a complete crash.

Malformed messages impact *integrity* and *availability*.

#### 4.3.8 Server Profiling

An attacker tries to deduce the identity, type, software version, or vendor of the server or client in order to apply knowledge about specific vulnerabilities of that product to mount a more intrusive or damaging attack. The attacker might profile the target by sending valid or invalid formatted messages to the target and try to recognize the type of target by the pattern of its normal and error responses.

Server profiling impacts all of the objectives.

#### 4.3.9 Session Hijacking

An attacker injects valid formatted OPC-UA messages into an existing session by taking over a session.

An attacker may gain unauthorized access to data or perform unauthorized operations.

Session hijacking impacts all of the objectives.

#### 4.3.10 Rogue Server

An attacker builds a malicious OPC-UA server or installs an unauthorized instance of a genuine OPC-UA server.

The OPC client may disclose unnecessary information.

A rogue server impacts all of the security objectives.

#### 4.3.11 Compromising User Credentials

An attacker obtains user credentials such as usernames, passwords, certificates, or keys by observing them on papers, on screens, or in electronic communications; by cracking them through guessing or the use of automated tools such as password crackers.

An unauthorized user could launch and access the system to obtain all information and make control and data changes that harm plant operation or information. Once compromised credentials are used, subsequent activities may all appear legitimate.

Compromised user credentials impact *authorization*.

### 4.4 OPC UA Relationship to Site Security

OPC UA security works within the overall *cyber security management system* (CSMS) of the site. Sites often have a CSMS that addresses security policy and procedures, personnel, responsibilities, audits, and physical security. A CSMS typically addresses threats that include those that were described in Section 4.3. They also analyze the security risks and determine what security controls the site needs.

Resulting security controls commonly implement a “defense-in-depth” strategy that provides multiple layers of protection and recognizes that no single layer can protect against all attacks. Boundary protections, shown as abstract examples in Figure 1, may include firewalls, intrusion detection systems, controls on dial-in connections, and controls on media and computers that are brought into the system. Protections in components of the system may include hardened configuration of the operating systems, security patch management, anti-virus programs, and not allowing email in the control network. Standards that may be followed by a site include CIP 002-1 through CIP 009-1 and IEC 62351 which are referenced in Section 2.

The security requirements of a site CSMS apply to its OPC UA interfaces. That is, the security requirements of the OPC UA interfaces that are deployed at a site are specified by the site, not by the OPC UA specification. OPC UA specifies features that are intended so that conformant client and server products can meet the security requirements that are expected to be made by sites where they will be deployed. Those who are responsible for the security at the site should determine how to meet the site requirements with OPC UA conformant products.

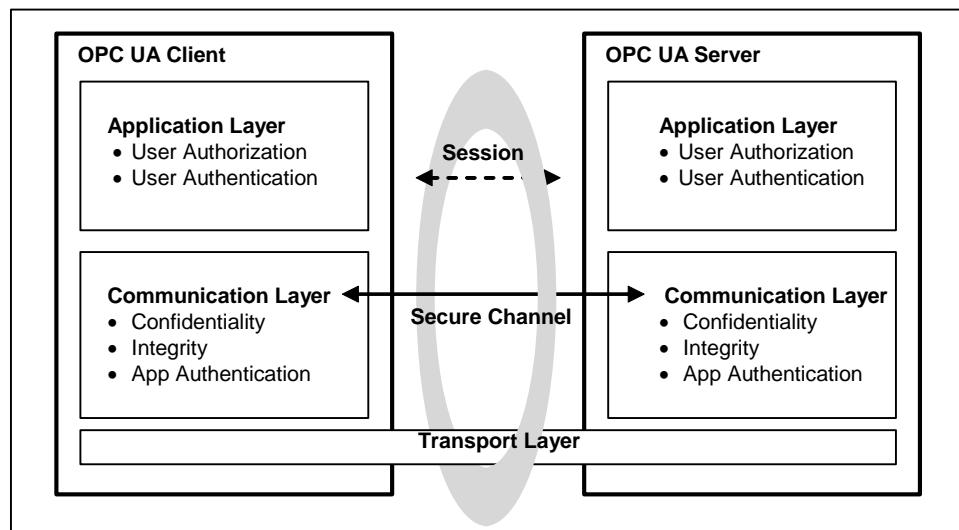
The system owner that installs OPC UA clients or servers must analyze its security risks and must provide appropriate mechanisms to mitigate those risks to achieve an acceptable level of security. OPC UA must meet the wide variety of security needs that might result from such individual analyses. OPC UA clients and servers are required to be implemented with certain security features, which are available for the system owner’s optional use. Each system owner should be able to tailor a security solution that meets its security and economic requirements using a combination of mechanisms available within the OPC UA specification and external to OPC UA.

The security requirements placed on the OPC UA clients and servers deployed at a site are specified by the site CSMS, not by the OPC UA specification. The OPC UA security specifications, however, are requirements placed upon OPC UA client and server products, and recommendations of how OPC UA should be deployed at a site in order to meet the security requirements that are anticipated to be specified at the site.

OPC UA addresses some threats as described in the section 4.3. The OPC foundation recommends that vendors address the remaining threats, as detailed in section 6. Threats to infrastructure components that might result in the compromise of client and server operating systems are not addressed by OPC-UA.

## 4.5 OPC UA Security Architecture

The OPC UA security architecture is a generic solution that allows implementation of the required security features at various places in the OPC UA architecture. Depending on the different mappings described in [UA Part 6], the security functionalities are addressed at different levels. The OPC UA Security Architecture is structured in an Application Layer and a Communication Layer atop the Transport Layer as shown in Figure 2.



**Figure 2 – OPC UA Security Architecture**

The routine work of a client application and a server application to transmit plant information, settings, and commands is done in a session in the application layer. The application layer also manages the security functions of user *authentication* and user *authorization*. The security functions that are managed by the Application Layer are provided by the Session Services that are specified in [UA Part 4]. A session in the Application Layer communicates over a *secure channel* that is created in the Communication Layer and relies upon it for secure communication. All of the session data is passed to the Communication Layer for further processing.

Although a session communicates over a *secure channel*, the binding of users, sessions, and *secure channels* is flexible. Impersonation allows the user of the session to change. A session can have a different user than the user that created the *secure channel*. To survive the loss of the original channel and resume with another, the implementation of the communication channel is responsible to re-establish the connection without interrupting the logical secure channel.

The Communication Layer provides security functionalities to meet *confidentiality*, *integrity* and application *authentication* as security objectives. The provided security functionalities together with negotiated and secret information are used to establish a *secure channel* between a client and a server. This logical channel provides encryption to maintain *confidentiality*, signatures to maintain *integrity* and certificates to provide application *authentication* for data that comes from the Application Layer and passes the “secured” data to the Transport Layer. The security functions that are managed by the Communication Layer are provided by the Secure Channel Services that are specified in [UA Part 4].

The security functions provided by the *secure channel* services are implemented by a protocol stack that is chosen for the implementation. Mappings of the services to some of the protocol stack options are specified in [UA Part 6] which details how the functions of the protocol stack are used to meet the OPC UA security objectives.

The Communication Layer can represent an OPC UA protocol stack. OPC UA specifies two alternative stack mappings that can be used as the Communication Layer. These mappings are UA Native mapping and Web Services mapping.

If the UA Native mapping is used, then functionalities for confidentiality, integrity, application authentication, and the secure channel are similar to the [TLS/SSL] specifications, as described in detail in [UA Part 6].

If the Web Services mapping is used, then [WS Security], [WS Secure Conversation] and [XML Encryption] are used to implement the functionalities for confidentiality, integrity, application authentication as well as for implementing a *secure channel*. For more specific information, see [UA Part 6].

The Transport Layer handles the transmission, reception and the transport of data that is provided by the Communication Layer.

#### 4.6 Policy

Security policies specify which security mechanisms are to be used. Security policies are used by the server to announce what mechanisms it supports and by the client to select one of those available policies to be used for the secure channel it wishes to open. The mechanisms specified include the following:

- which messages will be signed: choice of: all messages or no messages at all
- whether to encrypt all messages or no messages
- algorithms for signing and encryption

The choice of policy is normally made by the control system administrator, typically when the client and server products are installed.

If a server serves multiple clients, it maintains separate policy selections for the different clients. This allows a new client to select policies independent of the policy choices that other clients have selected for their secure channels.

[UA Part 7] specifies several policies. To improve interoperability among vendors' products, server products should implement these policies rather than define their own.

The security policy structure may be extended in later OPC UA versions, so the security policy structure allows additional fields so that it can be compatible with servers that conform to later versions of OPC UA.

#### 4.7 Profile

OPC UA client and server products are certified against profiles that are defined in [UA Part 7]. Some of the profiles specify security functions and others specify other functionality that is not related to security. The profiles impose requirements on the certified products but they do not impose requirements on how the products are used. A consistent minimum level of security is required by the various profiles. However, different profiles specify different details, such as which encryption algorithms are required for which UA functions. If a problem is found in one encryption algorithm, then the OPC foundation can define a new profile that is similar, but that specifies a different encryption algorithm that does not have a known problem. [UA Part 7], not this Part 2, is the normative specification of the profiles.

Policies refer to many of the same security choices as profiles, however the policy specifies which of those choices to use in the session. The policy does not specify the range of choices that the product must offer like the profile does.

Each security mechanism in OPC UA is provided in client and server products in accordance with the profiles with which the client or server complies. At the site, however, the security mechanisms

may be deployed optionally. In this way each individual site has all of the OPC UA security functions available and can choose which of them to use to meet its security requirements.

#### 4.8 User Authorization

UA provides a mechanism to exchange user credentials but does not specify how the applications use these credentials. Client and server applications may determine in their own way what data is accessible and what operations are authorized.

#### 4.9 User Authentication

User *authentication* is provided by the Session Services with which the client passes user credentials to the server as specified in [UA Part 4]. The client and server can each authenticate the user with these credentials.

The user who is communicating over a session can be changed using the ImpersonateUser service in order to meet needs of the application. This does not change the identity of the user who created the *secure channel* over which the session is currently communicating.

#### 4.10 OPC UA Security Services

A set of cryptographic keys is used to encrypt and sign messages in order to protect *confidentiality* and *integrity*. They are negotiated by services of the *secure channel* service set. Session keys are changed periodically so that attackers do not have unlimited time and unrestricted sequences of messages to use to determine what the keys are. The functions for managing these keys are specified in [UA Part 4].

#### 4.11 Auditing

Clients and servers generate audit records of successful and unsuccessful connection attempts, results of security option negotiations, configuration changes, system changes, and session rejections.

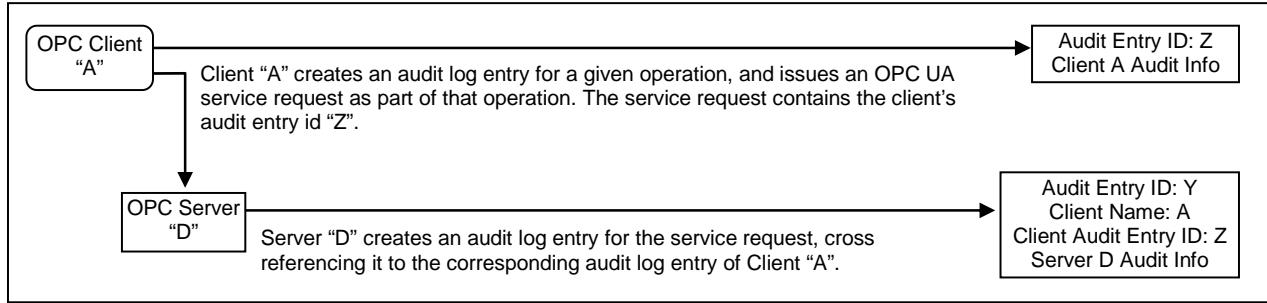
OPC UA provides support for security audit trails through two mechanisms. First, it provides for traceability between client and server audit logs. The client generates an audit log entry for an operation that includes a request. When the client issues a service request, it generates an audit log entry and includes the local identifier of the log entry in the request sent to the server. The server logs requests that it receives and includes the client's entry id in its audit log entry. In this fashion, if a security-related problem is detected at the server, the associated client audit log entry can be located and examined. OPC UA does not require the audit entries to be written to disk, but it does require that they be available. OPC UA provides the capability for servers to generate event notifications that report auditable events to clients capable of processing and logging them.

Second, OPC UA defines standard audit parameters to be included in audit records. This promotes consistency across audit logs and in *audit* events. [UA Part 5] defines the data types for these parameters. [UA Part 7] defines profiles, which include the ability to generate audit events and use these parameters, including the client audit record id.

The following clauses illustrate the behavior of OPC-UA servers and clients that support *auditing*.

#### 4.11.1 Single client and server

Figure 3 illustrates the simple case of a client communicating with a server.



**Figure 3 – Simple Servers**

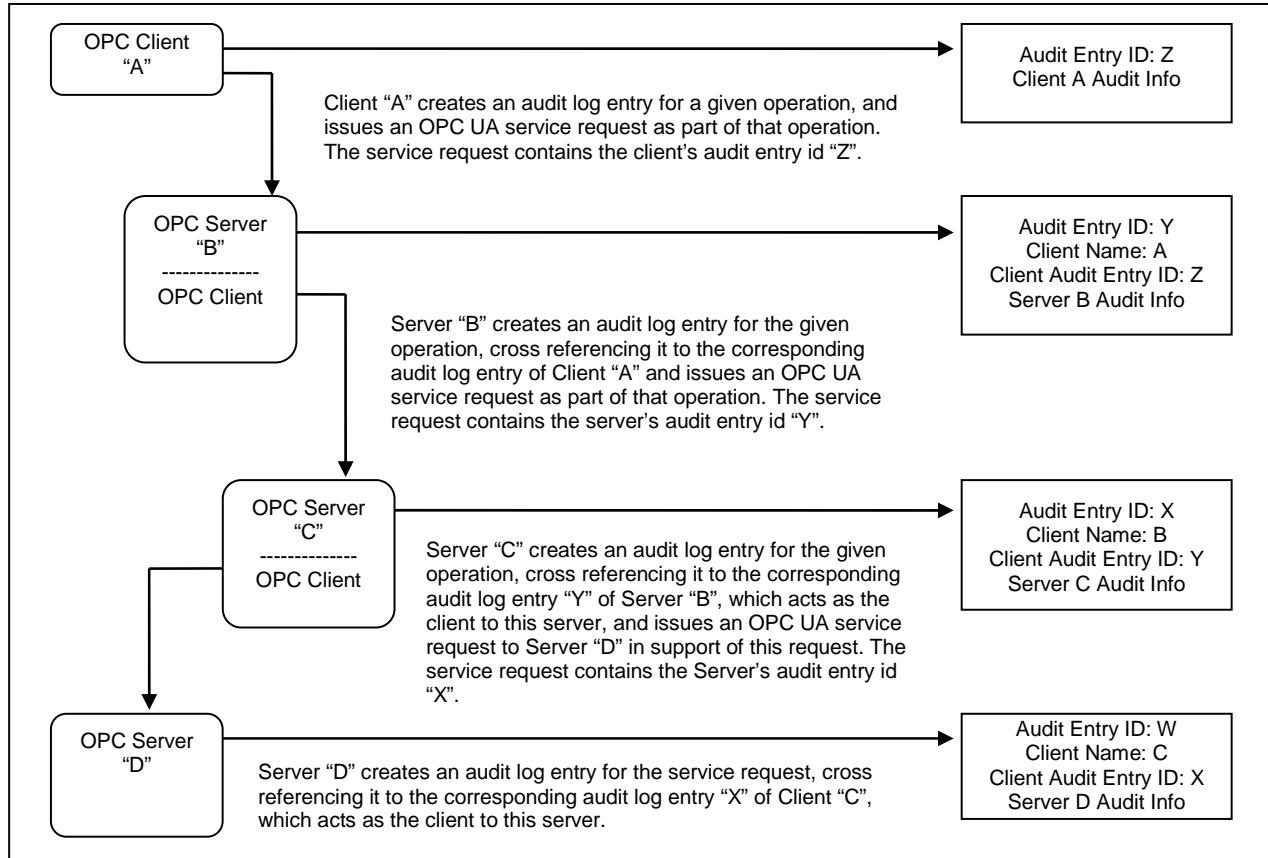
In this case, Client "A" executes some auditable operation that includes the invocation of an OPC UA service in Server "D". It writes its own audit log entry, and includes the identifier of that entry in the service request that it submits to the server.

The server receives the request and creates its own audit log entry for it. This entry is identified by its own audit id and contains its own *auditing* information. It also includes the name of the client that issued the service request and the client audit entry id received in the request.

Using this information, an auditor can inspect the collection of log entries of the server and relate them back to their associated client entries.

#### 4.11.2 Aggregating server

Figure 4 illustrates the case of a client accessing services from an aggregating server. An aggregating server is a server that provides its services by accessing services of other OPC UA servers, referred to as lower layer-servers.



**Figure 4 – Aggregating Servers**

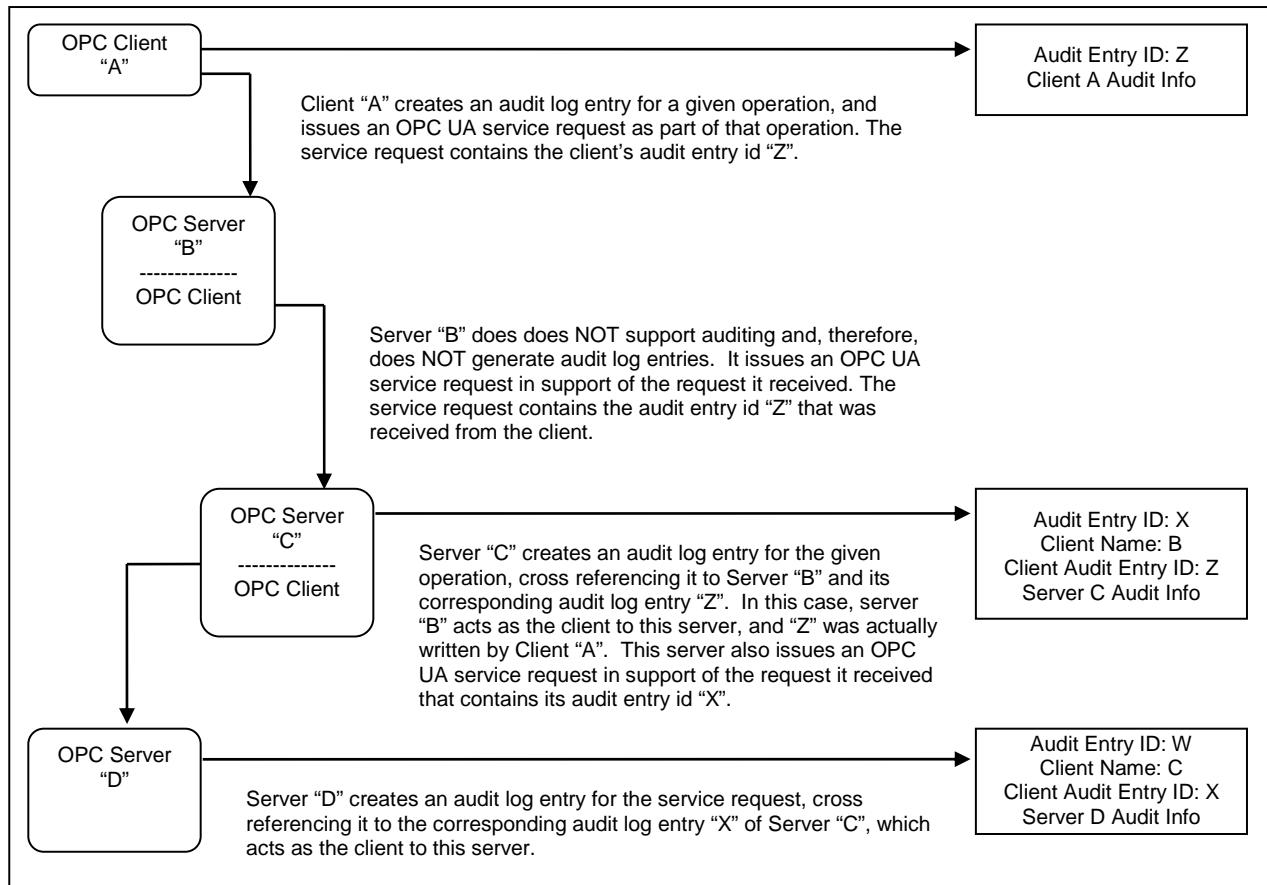
In this case, each of the servers receives requests and creates its own audit log entry for them. Each entry is identified by its own audit id and contains its own *auditing* information. It also includes the name of the client that issued the service request and the client audit entry id received in the request. The server then passes the audit id of the entry it just created to the next server in the chain.

Using this information, an auditor can inspect the server's log entries and relate them back to their associated client entries.

In most cases the servers will only generate Audit events, but these Audit events will still contain the same information as the Audit log records. In the case of aggregating servers a server would also be required to subscribe for audit events from the servers it is aggregating. In this manner, Server "B" would be able to provide all of the audit events to Client "A", including the event generated by Server "C" and server "D".

#### 4.11.3 Aggregation through a non-auditing server

Figure 5 illustrates the case of a client accessing services from an aggregating server that does not support *auditing*.



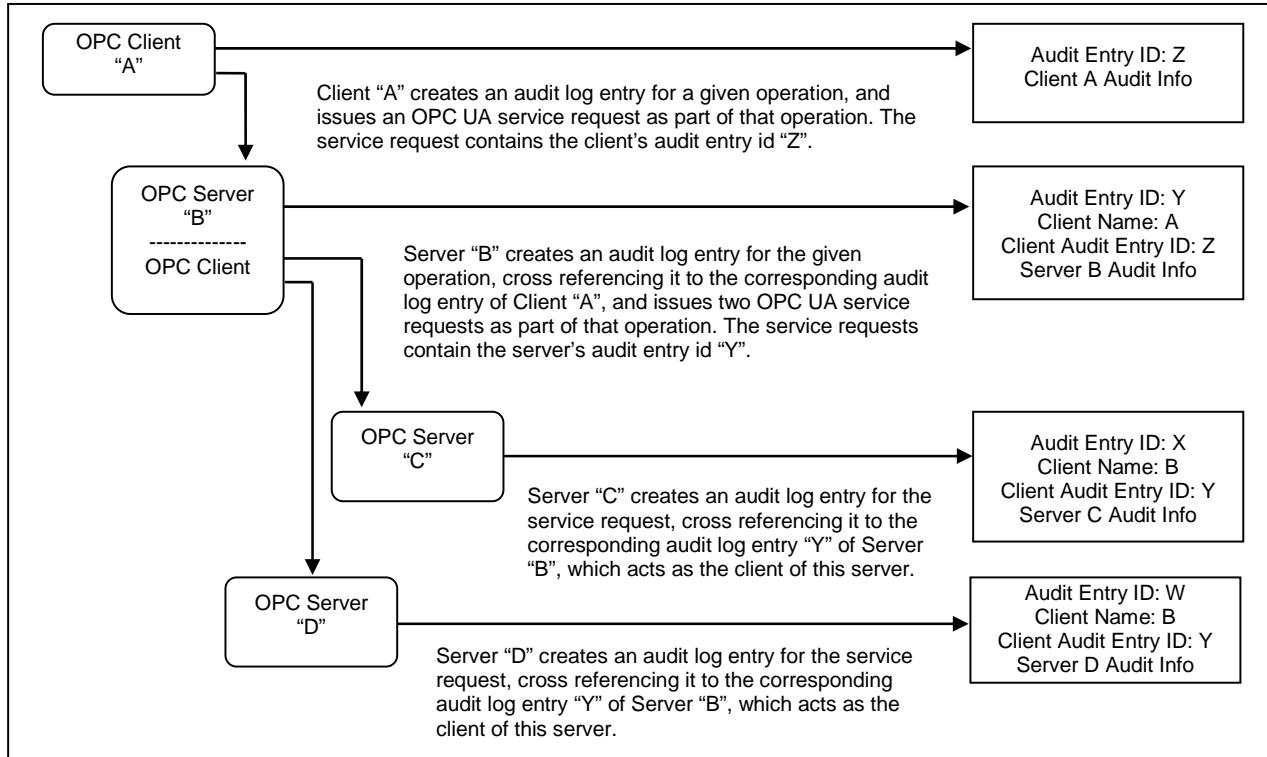
**Figure 5 – Aggregation with a Non-Auditing Server**

In this case, each of the servers receives their requests and creates their own audit log entry for them, with the exception of Server "B", which does not support *auditing*. In this case, Server "B" passes the audit id it receives from its Client "A" to the next server. This creates the required audit chain. Server "B" is not listed as supporting *auditing*. In a case where a server does not support writing audit entries, the entire system may be considered as not supporting *auditing*.

In the case of an aggregating server that does not support *auditing*, the server would still be required to subscribe for audit events from the servers it is aggregating. In this manner, Server "B" would be able to provide all of the audit events to Client "A", including the event generated by Server "C" and server "D", even though it did not generate an audit event.

#### 4.11.4 Aggregating server with service distribution

Figure 6 illustrates the case of a client that submits a service request to an aggregating server, and the aggregating service supports that service by submitting multiple service requests to its underlying servers.



**Figure 6 – Aggregate server with Service Distribution**

In the case of aggregating servers a server would also be required to subscribe for audit events from the servers it is aggregating. In this manner, Server "B" would be able to provide all of the audit events to Client "A", including the event generated by Server "C" and server "D"

## 5 Security Reconciliation

### 5.1 Reconciliation of Threats with OPC UA Functions

#### 5.1.1 Overview

The following subsections of Section 5.1 reconcile the threats that were described in Section 4.3 against the OPC UA functions. Each of the following subsections relates directly to the threat described in its corresponding subsection of Section 4.3.

Compared to the reconciliation with the objectives that will be given in section 5.2, this is a more specific reconciliation that relates OPC UA security functions to specific threats.

#### 5.1.2 Message Flooding

OPC UA minimizes the loss of *availability* caused by message flooding by minimizing the amount of processing that must be done with a message before the message is authenticated. This prevents an attacker from leveraging a small amount of effort to cause the legitimate *OPC UA Application* to spend a large amount of time responding, thus taking away processing resources from legitimate activities. GetSecurityPolicies (specified in [UA Part 4]) and OpenSecureChannel (specified in [UA Part 4]) are the only services that the server must handle before the client is recognized. The response to GetSecurityPolicies is only a small set of static information so the server does not need to do much processing. The response to OpenSecureChannel consumes significant server resources because of the signature and encryption processing. The server implementation could protect itself from floods of OpenSecureChannel messages in two ways: first, the server could intentionally delay its processing of OpenSecureChannel requests once it receives a bad one and issuing an alarm would alert plant personnel that an attack is underway that could still be blocking new legitimate OpenSecureChannel calls; second, when an OpenSecureChannel request attempts to exceed the server's specified maximum number of concurrent channels the server must give an error response without performing the signature and encryption processing. Certified OPC UA servers are required to specify their maximum number of concurrent channels in their product documentation as specified in [UA Part 7].

OPC UA user and client *authentication* reduce the risk of a legitimate client being used to mount a flooding attack. See the reconciliation of *authentication* in Section 5.2.2.

OPC UA *auditing* functionality provides the site with evidence that can help the site discover that flooding attacks are being mounted and find ways to prevent similar future attacks. See Section 4.11.

OPC UA relies upon the site CSMS to prevent attacks such as message flooding at protocol layers and systems that support OPC UA.

#### 5.1.3 Eavesdropping

OPC UA provides encryption to protect against eavesdropping as described in Section 5.2.4 - Confidentiality.

#### 5.1.4 Message Spoofing

As specified in the OpenSecureChannel service in [UA Part 4], OPC UA counters message spoofing threats by the possibility to sign messages. Additionally all messages must contain a valid session id and are assigned to a *secure channel*.

### 5.1.5 Message Alteration

OPC UA counters message alteration by the signing of messages that is specified in the OpenSecureChannel service specified in [UA Part 4]. If messages are altered, checking the signature will reveal any changes and allow the recipient to discard the message.

### 5.1.6 Message Replay

OPC UA uses session ID, timestamps, and sequence numbers for every message request and response. If messages are additionally signed then messages cannot be replayed without detection. Timestamps are specified in the RequestHeader section in [UA Part 4]. Signing is specified in the OpenSecureChannel service in [UA Part 4]. Sequence numbers are specified in the section that specifies the Write, Call, and Notification Message services in [UA Part 4].

### 5.1.7 Malformed Messages

Implementations of OPC UA client and server products counter threats of malformed messages by checking that messages have the proper form and that parameters of messages are within their legal range. This is specified in [UA Part 6] and a similar specification regarding range checking of parameters of services is in [UA Part 4].

### 5.1.8 Server Profiling

OPC UA limits the amount of information that servers provide to clients that have not yet been identified. This information is the response to the GetServerPolicies service specified in [UA Part 4].

Compliance testing might improve the consistency of error handling by OPC UA products from different vendors.

### 5.1.9 Session Hijacking

OPC UA counters session hijacking by assigning a security context with each session as specified in the CreateSession service in [UA Part 4]. Hijacking a session would thus first require compromising the security context.

### 5.1.10 Rogue Server

OPC UA counters the use of rogue servers to collect information from a client by requiring servers to authenticate to clients as specified in the OpenSecureChannel service in [UA Part 4].

### 5.1.11 Compromising User Credentials

OPC UA protects user credentials sent over the network by encryption as described in Section 5.2.4 - Confidentiality.

OPC UA depends upon the site CSMS to protect against other attacks to gain user credentials, such as password guessing or social engineering.

## 5.2 Reconciliation of Objectives with OPC UA Functions

### 5.2.1 Overview

The following subsections of Section 5.2 reconcile the objectives that were described in Section 4.2 with the OPC UA functions. Each of the following subsections relates directly to the objective described in its corresponding subsection of Section 4.2.

Compared to the reconciliation against the threats of section 5.1, this reconciliation justifies the completeness of the OPC UA security architecture.

### 5.2.2 Authentication

*OPC UA Applications* support *authentication* of the entities with which they are communicating as well as providing the necessary *authentication* credentials to the other entities.

#### Application Authentication

As specified in the OpenSecureChannel service in [UA Part 4], OPC UA client and server applications identify and authenticate themselves with [X509] certificates. Some choices of the Communication Stack require these certificates to represent the machine or user instead of the application.

#### User Authentication

As described in the OpenSecureChannel service in [UA Part 4], the OPC UA client accepts a user identity token from the user, authenticates the token, and passes it to the OPC UA server. The OPC UA server authenticates the user token. *OPC UA Applications* accept tokens in any of the following three forms: username/password, [X509] v3, or Web Services compliant. Clients and servers accept username/password tokens.

As specified in the sections of the CreateSession and ActivateSession services in [UA Part 4], the user identity token is validated with a challenge-response process using the ClientNonce and ServerNonce. The server provides a *nonce* and signing algorithm as the challenge in its CreateSession response. The client responds to the challenge by signing the server's *nonce* and providing it as an argument in its subsequent ActivateSession call.

### 5.2.3 Authorization

OPC UA does not specify how user or client *authorization* is to be provided. *OPC UA Applications* that are part of a larger industrial automation product may manage *authorizations* consistent with the *authorization* management of that product. Identification and *authentication* of users is specified in OPC UA so that client and server applications can recognize the user in order to determine the *authorizations* of the user.

OPC UA servers respond with the Bad\_UserAccessDenied error code to indicate an *authorization* error as specified in the status codes section of [UA Part 4].

### 5.2.4 Confidentiality

OPC UA provides encryption to protect *confidentiality*. Encryption is specified in [UA art 6] in the sections on encryption and decryption in the UA Native Mapping and in the section on encrypting messages of [WS Security] in the XML Web Services Mapping.

OPC UA relies upon the site CSMS to protect *confidentiality* on the network and system infrastructure. OPC UA relies upon the CSMS to manage keys.

### 5.2.5 Integrity

*Integrity* is threatened by message spoofing, message alteration, message replay, malformed messages, session hijacking which are reconciled with OPC UA countermeasures in their respective subsections of Section 5.1. The signing with changing keys of messages that contain session ID, timestamp, and sequence number provides a complete foundation to protect *integrity*.

### 5.2.6 Auditability

As specified in the UA Auditing section of [UA Part 4], OPC UA supports audit logging by providing traceability of activities through the log entries of the multiple clients and servers that initiate, forward, and handle the activity. OPC UA depends upon *OPC UA Application* products to provide an effective audit logging scheme or an efficient manner of collecting the Audit events of all nodes. This scheme may be part of a larger industrial automation product of which the *OPC UA Applications* are part.

### 5.2.7 Availability

OPC UA minimizes the impact of message flooding as described in Section 5.1.2.

Some attacks on *availability* involve opening more sessions than a server can handle thereby causing the server to fail or operate poorly. Servers reject sessions that exceed their specified maximum number.

## 6 Implementation considerations

This section provides guidance to vendors that implement OPC-UA applications. Since many of the countermeasures required to address the threats described above fall outside the scope of the OPC-UA specification, the advice in this section suggests how some of those countermeasures should be provided.

For each of the following areas, this section defines the problem space, identifies consequences if appropriate countermeasures are not implemented and recommends best practices.

### 6.1 Application Instance Certificates

It is recommended that all UA applications create a unique, self-signed root certificate which is defined as the *Application Instance Certificate* whenever a new instance of the application is installed. The *public key* for this auto-generated certificate should be stored in a location accessible to the administrator. This will allow the administrator to configure other UA applications that need to communicate with the new UA *Application Instance*. The UA application should also allow the administrator to replace the auto-generated certificate with one issued by a CA selected by the administrator. The trusted certificate list should be protected at the site from unauthorized alteration.

It also recommended that UA applications simplify the process of adding new self-signed certificates to its list of trusted certificates. If possible, the UA application should generate some sort of notification whenever UA applications exchange certificates using UA services for the first time. The best process depends on the application. However, it could include displaying a dialog to the user and asking the user to confirm that the certificate is acceptable. Server applications or applications without a user interface could conditionally accept the certificate and report it to an administrator via a security log or e-mail. UA applications should never silently accept certificates that are either self-signed or signed by an unknown certification authority.

### 6.2 Appropriate Timeouts:

Time outs, the time that the implementation must wait (usually for an event such as message arrival), play a very significant role in influencing the security of an implementation. Potential consequences include

- Denial of service: Denial of service conditions may exist when a client does not reset a session, if the timeouts are very large.
- Resource consumption: When a client is idle for long periods of time, the server must keep the client information for that period, leading to resource exhaustion.

The implementer should use reasonable timeouts for each connection stage.

### 6.3 Strict Message Processing

The specifications often specify the format of the right messages and are silent on what the implementation should do for messages that deviate from the specification. Typically, the implementations continue to parse such packets, leading to vulnerabilities.

- The implementer should do strict checking of the message format and should either drop the packets or send an error message as described below.
- Error handling shall use the error code, defined in [UA Part 4], that most precisely fits the condition.

#### **6.4 Robust Error recovery**

An error (hang, entering wrong protocol state, etc.) in the protocol may occur due to various reasons such as improper specification of the protocol or the client sending messages that are out of the scope of the specification. Graceful error recovery practices should be implemented so that the implementation recovers from the error. Sending the right type of error messages and reclaiming the memory are examples of such practices.

#### **6.5 Random Number Generation**

Random numbers that meet security needs can be generated by suitable functions that are provided by cryptography libraries. Other means are too weak, such as using CRT rand().

#### **6.6 Special and Reserved Packets**

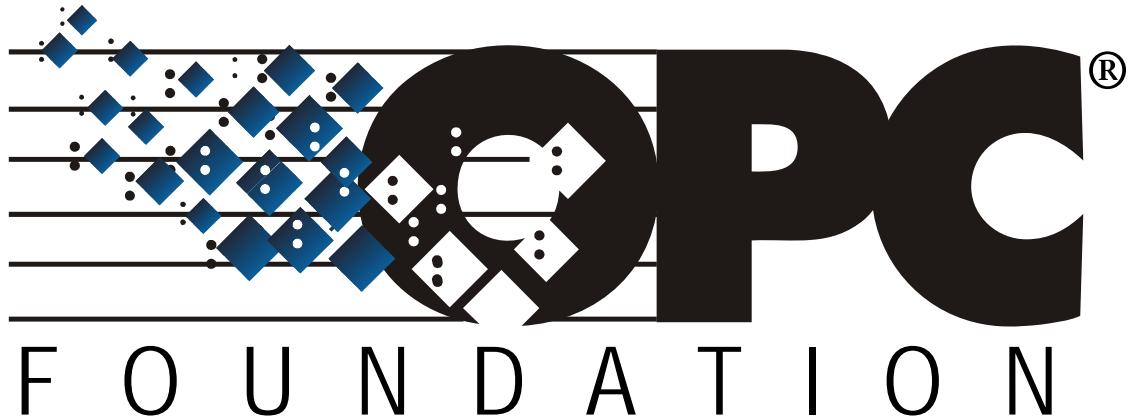
The implementation must understand and correctly interpret any message types that are reserved as special (such as broadcast and multicast addresses in IP specification). Failing to understand and interpret those special packets may lead to vulnerabilities.

#### **6.7 Rate Limiting and Flow Control**

OPC-UA does not provide rate control mechanisms, however an implementation can incorporate rate control.

### **7 Site Recommendations**

In later releases of this specification, this section will contain recommendations of actions that should be taken at the site in order to provide security of OPC UA.



# **OPC Unified Architecture**

## **Specification**

### **Part 3: Address Space Model**

**Version 1.00**

**July 28, 2006**



Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture Part 3: Address Space Model	Date:	June 28, 2006
Version:	Release 1.00	Software Source:	MS-Word OPC UA Part 3 - Address Space Model 1.00 Specification.doc
Author:	OPC Foundation	Status:	Release

## CONTENTS

	Page
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, abbreviations, and conventions .....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 2 terms .....	2
3.3 OPC UA Address Space Model terms .....	2
3.3.1 DataVariable .....	2
3.3.2 EventType .....	2
3.3.3 Hierarchical Reference .....	2
3.3.4 InstanceDeclaration .....	2
3.3.5 ModellingRule .....	2
3.3.6 Property .....	2
3.3.7 SourceNode .....	2
3.3.8 TargetNode .....	2
3.3.9 TypeDefinitionNode .....	2
3.3.10 VariableType .....	2
3.4 Abbreviations and symbols .....	3
3.5 Conventions .....	3
3.5.1 Conventions for AddressSpace figures .....	3
3.5.2 Conventions for defining NodeClasses .....	3
4 AddressSpace concepts .....	5
4.1 Overview .....	5
4.2 Object Model .....	5
4.3 Node Model .....	5
4.3.1 General .....	5
4.3.2 NodeClasses .....	6
4.3.3 Attributes .....	6
4.3.4 References .....	6
4.4 Variables .....	7
4.4.1 General .....	7
4.4.2 Properties .....	7
4.4.3 DataVariables .....	7
4.5 TypeDefinitionNodes .....	8
4.5.1 Overview .....	8
4.5.2 Complex TypeDefinitionNodes and their InstanceDeclarations .....	8
4.5.3 Subtyping .....	9
4.5.4 Instantiation of complex TypeDefinitionNodes .....	10
4.6 Event Model .....	11
4.6.1 Overview .....	11
4.6.2 EventTypes .....	11
4.6.3 Event Categorization .....	11
4.7 Methods .....	12
5 Standard NodeClasses .....	12
5.1 Overview .....	12
5.2 Base NodeClass .....	13
5.2.1 General .....	13

5.2.2	NodeId .....	13
5.2.3	NodeClass .....	13
5.2.4	BrowseName .....	13
5.2.5	DisplayName .....	14
5.2.6	Description .....	14
5.3	ReferenceType NodeClass .....	14
5.3.1	General .....	14
5.3.2	Attributes .....	14
5.3.3	References .....	16
5.4	View NodeClass .....	17
5.5	Objects .....	19
5.5.1	Object NodeClass .....	19
5.5.2	ObjectType NodeClass .....	20
5.5.3	Standard ObjectType FolderType .....	21
5.6	Variables .....	21
5.6.1	General .....	21
5.6.2	Variable NodeClass .....	21
5.6.3	Property .....	24
5.6.4	DataVariable .....	24
5.6.5	VariableType NodeClass .....	26
5.7	Method NodeClass .....	28
5.8	DataTypes .....	29
5.8.1	DataType Model .....	29
5.8.2	DataType NodeClass .....	31
5.8.3	DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem .....	32
5.9	Summary of Attributes of the NodeClasses .....	34
5.10	Subtyping of ObjectTypes and VariableTypes .....	34
5.11	Instantiation of ObjectTypes and VariableTypes .....	35
5.11.1	General .....	35
5.11.2	Ownership due to ModellingRules .....	35
5.11.3	Standard ModellingRules .....	36
6	Standard ReferenceTypes .....	39
6.1	General .....	39
6.2	References ReferenceType .....	39
6.3	HierarchicalReferences ReferenceType .....	39
6.4	NonHierarchicalReferences ReferenceType .....	40
6.5	Aggregates ReferenceType .....	40
6.6	HasComponent ReferenceType .....	40
6.7	HasProperty ReferenceType .....	40
6.8	HasOrderedComponent ReferenceType .....	41
6.9	HasSubtype ReferenceType .....	41
6.10	Organizes ReferenceType .....	41
6.11	HasModellingRule ReferenceType .....	41
6.12	HasTypeDefinition ReferenceType .....	42
6.13	HasEncoding ReferenceType .....	42
6.14	HasDescription ReferenceType .....	42
6.15	GeneratesEvent .....	42
6.16	HasEventSource .....	43

6.17	HasNotifier.....	43
7	Standard DataTypes .....	45
7.1	General .....	45
7.2	NodeId.....	45
7.2.1	General.....	45
7.2.2	NamespaceIndex.....	45
7.2.3	IdType.....	45
7.2.4	Identifier value .....	46
7.3	QualifiedName .....	46
7.4	LocaleId.....	47
7.5	LocalizedText .....	47
7.6	Argument.....	48
7.7	BaseDataType .....	48
7.8	Boolean .....	48
7.9	Byte.....	48
7.10	ByteString.....	48
7.11	Date .....	48
7.12	Double.....	48
7.13	Float.....	48
7.14	Guid .....	48
7.15	SByte .....	48
7.16	IdType .....	49
7.17	Integer.....	49
7.18	Int16.....	49
7.19	Int32.....	49
7.20	Int64.....	49
7.21	NodeClass .....	49
7.22	Number.....	49
7.23	String.....	49
7.24	Time .....	49
7.25	UInteger .....	49
7.26	UInt16 .....	50
7.27	UInt32 .....	50
7.28	UInt64 .....	50
7.29	UtcTime .....	50
7.30	XmlElement .....	50
8	Standard EventTypes.....	51
8.1	General .....	51
8.2	BaseEventType.....	51
8.3	SystemEventType .....	51
8.4	AuditEventType.....	51
8.5	AuditSecurityEventType .....	53
8.6	AuditChannelEventType .....	53
8.7	AuditOpenSecureChannelEventType .....	53
8.8	AuditCloseSecureChannelEventType .....	53
8.9	AuditSessionEventType .....	53
8.10	AuditCreateSessionEventType .....	54
8.11	AuditActivateSessionEventType .....	54
8.12	AuditImpersonateUserEventType .....	54

8.13	AuditNodeManagementEventType .....	54
8.14	AuditAddNodesEventType .....	54
8.15	AuditDeleteNodesEventType .....	54
8.16	AuditAddReferencesEventType .....	54
8.17	AuditDeleteReferencesEventType .....	54
8.18	AuditUpdateEventType .....	54
8.19	DeviceFailureEventType .....	54
8.20	ModelChangeEvents .....	54
8.20.1	General .....	54
8.20.2	NodeVersion Property .....	55
8.20.3	Views .....	55
8.20.4	Event Compression .....	55
8.20.5	BaseModelChangeEventType .....	55
8.20.6	GeneralModelChangeEventType .....	55
8.20.7	Guidelines for ModelChangeEvents .....	55
8.21	PropertyChangeEvent-Type .....	56
8.21.1	General .....	56
8.21.2	ViewVersion and NodeVersion Properties .....	56
8.21.3	Views .....	56
8.21.4	Event Compression .....	56
Appendix A	: How to use the Address Space Model .....	57
A.1	Overview .....	57
A.2	Type definitions .....	57
A.3	ObjectTypes .....	57
A.4	VariableTypes .....	57
A.4.1	General .....	57
A.4.2	Properties or DataVariables .....	58
A.4.3	Many Variables and / or complex DataTypes .....	58
A.5	Views .....	59
A.6	Methods .....	59
A.7	Defining ReferenceTypes .....	59
A.8	Defining ModellingRules .....	59
Appendix B	: OPC UA Meta Model in UML .....	60
B.1	Background .....	60
B.2	Notation .....	60
B.3	Meta Model .....	61
B.3.1	BaseNode .....	62
B.3.2	ReferenceType .....	62
B.3.3	Predefined ReferenceTypes .....	63
B.3.4	Attributes .....	64
B.3.5	Object and ObjectType .....	65
B.3.6	Variable and VariableType .....	66
B.3.7	Method .....	67
B.3.8	EventNotifier .....	67
B.3.9	DataType .....	68
B.3.10	View .....	68
Appendix C	: OPC Binary Type Description System .....	69
C.1	Concepts .....	69
C.2	Schema Description .....	70

C.2.1	TypeDictionary .....	70
C.2.2	TypeDescription .....	71
C.2.3	OpaqueTypeDescription .....	71
C.2.4	EnumeratedTypeDescription .....	72
C.2.5	StructuredTypeDescription .....	72
C.2.6	FieldDescription .....	72
C.2.7	EnumeratedValueDescription.....	74
C.2.8	ByteOrder .....	74
C.2.9	ImportDirective .....	74
C.3	Standard Type Descriptions .....	74
C.4	Type Description Examples .....	75
C.5	OPC Binary XML Schema .....	76
C.6	OPC Binary Standard TypeDictionary .....	78

## FIGURES

Figure 1 – AddressSpace Node diagrams .....	3
Figure 2 – OPC UA Object Model .....	5
Figure 3 – AddressSpace Node Model .....	6
Figure 4 – Reference Model .....	7
Figure 5 – Example of a Variable Defined By a VariableType .....	8
Figure 6 – Example of a Complex TypeDefinition .....	9
Figure 7 – Object and its Components defined by an ObjectType .....	10
Figure 8 – Symmetric and Non-Symmetric References .....	15
Figure 9 – Variables, VariableTypes and their DataTypes.....	29
Figure 10 – DataType Model.....	30
Figure 11 – Example of DataType Modelling .....	33
Figure 12 – Use of the Standard ModellingRule New .....	37
Figure 13 – Example of usage of Standard ModellingRules .....	38
Figure 14 – Standard ReferenceType Hierarchy .....	39
Figure 15 – Event Reference Example .....	44
Figure 16 – Complex Event Reference Example.....	44
Figure 17 – Standard EventType Hierarchy .....	51
Figure 18 – Audit Behaviour of a Server .....	52
Figure 19 – Audit Behaviour of an Aggregating Server .....	53
Figure 20 – Background of OPC UA Meta Model .....	60
Figure 21 – Notation (I) .....	60
Figure 22 – Notation (II) .....	61
Figure 23 – BaseNode.....	62
Figure 24 – Reference and ReferenceType .....	62
Figure 25 – Predefined ReferenceTypes .....	63
Figure 26 – Attributes.....	64
Figure 27 – Object and ObjectType.....	65
Figure 28 – Variable and VariableType .....	66
Figure 29 – Method .....	67
Figure 30 – EventNotifier.....	67
Figure 31 – DataType .....	68
Figure 32 – View .....	68
Figure 33 – OPC Binary Dictionary Structure .....	69

**TABLES**

Table 1 – NodeClass Table Conventions.....	3
Table 2 - Base NodeClass .....	13
Table 3 – ReferenceType NodeClass.....	14
Table 4 – View NodeClass.....	17
Table 5 – Object NodeClass .....	19
Table 6 – ObjectType NodeClass.....	20
Table 7 – Variable NodeClass .....	22
Table 8 – VariableType NodeClass.....	26
Table 9 – Method NodeClass.....	28
Table 10 – Data Type NodeClass .....	31
Table 11 – Overview about Attributes .....	34
Table 12 – Nodeld Definition .....	45
Table 13 – IdType Values.....	46
Table 14 – Nodeld Null Values .....	46
Table 15 – QualifiedName Definition.....	46
Table 16 –LocaleId Examples.....	47
Table 17 – LocalizedText Definition .....	47
Table 18 – Argument Definition.....	48
Table 19 – NodeClass Values.....	49
Table 20 – TypeDictionary Components.....	70
Table 21 – TypeDescription Components.....	71
Table 22 – OpaqueTypeDescription Components.....	71
Table 23 – EnumeratedTypeDescription Components .....	72
Table 24 – StructuredTypeDescription Components .....	72
Table 25 – FieldDescription Components .....	73
Table 26 – EnumeratedValueDescription Components .....	74
Table 27 – ImportDirective Components .....	74
Table 28 – Standard Type Descriptions .....	75

## OPC FOUNDATION

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these

materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>

## 1 Scope

This specification describes the OPC UA *AddressSpace* and its *Objects*.

## 2 Reference documents

- [UA Part 1]    OPC UA Specification: Part 1 – Concepts, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part1/>
- [UA Part 2]    OPC UA Specification: Part 2 – Security Model, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part2/>
- [UA Part 4]    OPC UA Specification: Part 4 – Services, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part4/>
- [UA Part 5]    OPC UA Specification: Part 5 – Information Model, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part5/>
- [UA Part 6]    OPC UA Specification: Part 6 – Mapping, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part6/>
- [UA Part 8]    OPC UA Specification: Part 8 – Data Access, Version 1.0 or later  
<http://www.opcfoundation.org/UA/Part8/>
- [XML Schema Part 1] <http://www.w3.org/TR/xmlschema-1/>
- [XML Schema Part 2] <http://www.w3.org/TR/xmlschema-2/>
- [EDDL]        IEC 61804-3 Ed. 1.0 – Function blocks (FB) for process control - Part 3: Electronic Device Description Language (EDDL)
- [XPATH]        <http://www.w3.org/TR/xpath/>

## 3 Terms, definitions, abbreviations, and conventions

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Attribute
- 3) Complex Data
- 4) Event
- 5) Information Model
- 6) Message
- 7) Method
- 8) Node
- 9) NodeClass
- 10) Notification
- 11) Object
- 12) ObjectType
- 13) Reference
- 14) ReferenceType

- 15) Service
- 16) Subscription
- 17) Variable
- 18) View

### 3.2 OPC UA Part 2 terms

There are no [UA Part 2] terms used in this part.

### 3.3 OPC UA Address Space Model terms

#### 3.3.1 DataVariable

*DataVariables* are *Variables* that represent values of *Objects*, either directly or indirectly for complex *Variables*. They are always the *TargetNode* for a *HasComponent Reference*.

#### 3.3.2 EventType

An *EventType* is an *ObjectType Node* that represents the type definition of an *Event*.

#### 3.3.3 Hierarchical Reference

*Hierarchical References* are *References* that are used to construct hierarchies in the *AddressSpace*. All hierarchical *ReferenceTypes* are derived from the *HierarchicalReferences*.

#### 3.3.4 InstanceDeclaration

An *InstanceDeclaration* is a *Node* that is used by a complex *TypeDefinitionNode* to expose its complex structure. It is an instance used by a type definition.

#### 3.3.5 ModellingRule

The *ModellingRule* of a *Node* defines how the *Node* will be used for instantiation or which rule was used to create the *Node*. It also defines subtyping rules for *InstanceDeclaration* and can specify the ownership of a *Node*.

#### 3.3.6 Property

*Properties* are *Variables* that are the *TargetNode* for a *HasProperty Reference*. *Properties* describe the characteristics of a *Node*.

#### 3.3.7 SourceNode

A *SourceNode* is a *Node* having a *Reference* to another *Node*. For example, in the *Reference* “A contains B”, “A” is the *SourceNode*.

#### 3.3.8 TargetNode

A *TargetNode* is a *Node* that is referenced by another *Node*. For example, in the *Reference* “A Contains B”, “B” is the *TargetNode*.

#### 3.3.9 TypeDefinitionNode

A *TypeDefinitionNode* is a *Node* that is used to define the type of another *Node*. *ObjectType*, *VariableType*, *ReferenceType*, and *data type Nodes* are *TypeDefinitionNodes*.

#### 3.3.10 VariableType

A *VariableType* is a *Node* that represents the type definition for a *Variable*.

### 3.4 Abbreviations and symbols

EDDL	Electronic Device Description Language
UA	Unified Architecture
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language

### 3.5 Conventions

#### 3.5.1 Conventions for AddressSpace figures

Nodes and their *References* to each other are illustrated using figures. Figure 1 illustrates the conventions used in these figures.

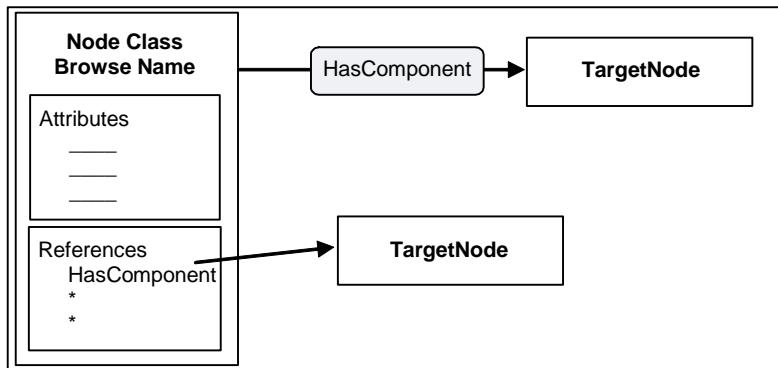


Figure 1 – AddressSpace Node diagrams

In these figures, rectangles represent *Nodes*. *Node* rectangles may be titled with one or two lines of text. When two lines are used, the first text line in the rectangle identifies the *NodeClass* and the second line contains the *BrowseName*. When one line is used, it contains the *BrowseName*.

*Node* rectangles may contain boxes used to define their *Attributes* and *References*. Specific names in these boxes identify specific *Attributes* and *References*.

Shaded rectangles with rounded corners and with arrows passing through them represent *References*. The arrow that passes through them begins at the *SourceNode* and points to the *TargetNode*. *References* may also be shown by drawing an arrow that starts at the *Reference name* in the “References” box and ends at the *TargetNode*.

#### 3.5.2 Conventions for defining NodeClasses

Clause 5 defines standard *AddressSpace NodeClasses*. Table 1 describes the format of the tables used to define *NodeClasses*.

Table 1 – NodeClass Table Conventions

Name	Use	Data Type	Description
<b>Attributes</b>			
“Attribute name”	“M” or “O”	Data type of the <i>Attribute</i>	Defines the <i>Attribute</i> .
<b>References</b>			
“Reference name”	“1”, “0..1” or “0..**”	Not used	Describes the use of the <i>Reference</i> by the <i>NodeClass</i> .
<b>Standard Properties</b>			
“Property name”	“M” or “O”	Data type of the <i>Property</i>	Defines the <i>Property</i> .

The Name column contains the name of the *Attribute*, the name of the *ReferenceType* used to create a *Reference* or the name of a standard *Property* referenced using the *HasProperty Reference*.

The Use column defines whether the *Attribute* or *Property* is mandatory (M) or optional (O). When mandatory the *Attribute* or *Property* must exist for every *Node* of the *NodeClass*. For *References* it specifies the cardinality. The following values may apply:

- “0..\*” identifies that there are no restrictions, that is, the *Reference* does not have to be provided but there is no limitation how often it can be provided;
- “0..1” identifies that the *Reference* is provided at most once;
- “1” identifies that the *Reference* must be provided exactly once.

The Data Type column contains the name of the *DataType* of the *Attribute* or *Property*. It is not used for *References*.

The Description column contains the description of the *Attribute*, the *Reference* or the *Property*.

Only OPC UA may define *Attributes*. Thus, all *Attributes* of the *NodeClass* are specified in the table and can only be extended by other parts of this multi-part specification.

OPC UA also defines standard *ReferenceTypes*, but *ReferenceTypes* can also be specified by a server or by a client using the *NodeManagement Services* specified in [UA Part 4]. Thus, the *NodeClass* tables contained in this specification can contain the base *ReferenceType* called *References* identifying that any *ReferenceType* may be used for the *NodeClass*, including system specific *ReferenceTypes*. The *NodeClass* tables only specify how the *NodeClasses* can be used as *SourceNodes* of *References*, not as *TargetNodes*. If a *NodeClass* table allows a *ReferenceType* for its *NodeClass* to be used as *SourceNode*, this is also true for subtypes of the *ReferenceType*. However, subclasses of the *ReferenceType* may restrict its *SourceNodes*.

OPC UA defines standard *Properties*, but standard *Properties* can be defined by other standard organizations or vendors and *Nodes* can have *Properties* that are not standardised. *Properties* defined in this document are defined by their name, which is mapped to the *BrowseName* having the *NamespaceIndex* 0, which represents the *Namespace* for OPC UA.

## 4 AddressSpace concepts

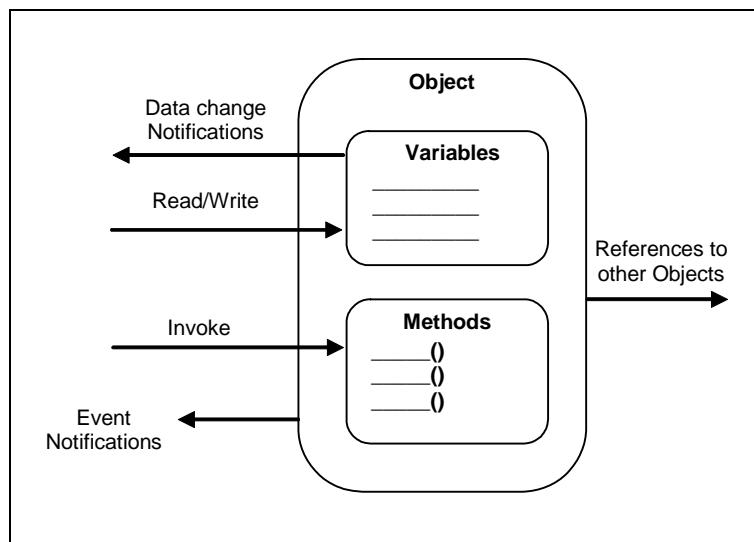
### 4.1 Overview

The following subclauses define the concepts of the *AddressSpace*. Clause 5 defines the *NodeClasses* of the *AddressSpace* representing the *AddressSpace* concepts. Standard *ReferenceTypes*, *DataTypes* and *EventTypes* are defined in Clauses 6-8.

The informative Appendix A describes general considerations how to use the Address Space Model and the informative Appendix B provides a UML Model of the Address Space Model.

### 4.2 Object Model

The primary objective of the OPC UA *AddressSpace* is to provide a standard way for servers to represent *Objects* to clients. The OPC UA Object Model has been designed to meet this objective. It defines *Objects* in terms of *Variables* and *Methods*. It also allows relationships to other *Objects* to be expressed. Figure 2 illustrates the model.



**Figure 2 – OPC UA Object Model**

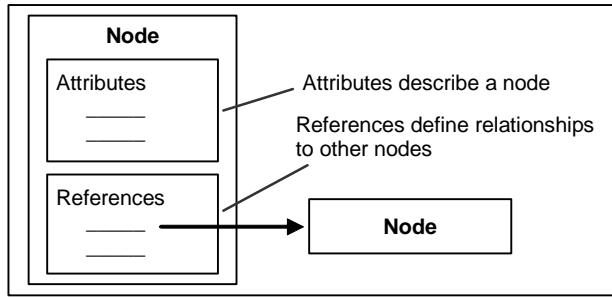
The elements of this model are represented in the *AddressSpace* as *Nodes*. Each *Node* is assigned to a *NodeClass* and each *NodeClass* represents a different element of the Object Model. Clause 5 defines the *NodeClasses* used to represent this model.

### 4.3 Node Model

#### 4.3.1 General

The set of *Objects* and related information that the OPC UA server makes available to clients is referred to as its *AddressSpace*. The model for *Objects* is defined by the OPC UA Object Model (see Clause 4.2).

Objects and their components are represented in the *AddressSpace* as a set of *Nodes* described by *Attributes* and interconnected by *References*. Figure 3 illustrates the model of a *Node* and the remainder of this Clause discusses the details of the Node Model.

**Figure 3 – AddressSpace Node Model**

#### 4.3.2 NodeClasses

*NodeClasses* are defined in terms of the *Attributes* and *References* that must be instantiated (given values) when a *Node* is defined in the *AddressSpace*. *Attributes* are discussed in Clause 4.3.3 and *References* in Clause 4.3.4.

Clause 5 defines the standard *NodeClasses* for the OPC UA *AddressSpace*. These *NodeClasses* are referred to collectively as the metadata for the *AddressSpace*. Each *Node* in the *AddressSpace* is an instance of one of these *NodeClasses*. No other *NodeClasses* may be used to define *Nodes*, and as a result, clients and servers are not allowed to define *NodeClasses* or extend the definitions of these *NodeClasses*.

#### 4.3.3 Attributes

*Attributes* are data elements that describe *Nodes*. Clients can access *Attribute* values using Read, Write, Query, and Subscription/MonitoredItem Services. These Services are defined in [UA Part 4].

*Attributes* are elementary components of *NodeClasses*. *Attribute* definitions are included as part of the *NodeClass* definitions in Clause 5 and, therefore, are not included in the *AddressSpace*.

Each *Attribute* definition consists of an integer id<sup>1</sup>, a name, a description, a data type and a mandatory/optional indicator. The set of *Attributes* defined for each *NodeClass* may not be extended by clients or servers.

When a *Node* is instantiated in the *AddressSpace*, the values of the *NodeClass Attributes* are provided. The mandatory/optional indicator for the *Attribute* indicates whether the *Attribute* has to be instantiated.

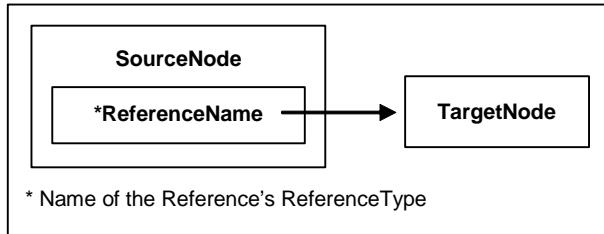
#### 4.3.4 References

*References* are used to relate *Nodes* to each other. They can be accessed using the browsing and querying Services defined in [UA Part 4].

Like *Attributes*, they are defined as fundamental components of *Nodes*. Unlike *Attributes*, *References* are defined as instances of *ReferenceType Nodes*. *ReferenceType Nodes* are visible in the *AddressSpace* and are defined using the *ReferenceType NodeClass* (see Clause 5.3).

The *Node* that contains the *Reference* is referred to as the *SourceNode* and the *Node* that is referenced is referred to as the *TargetNode*. The combination of the *SourceNode*, the *ReferenceType* and the *TargetNode* are used in OPC UA Services to uniquely identify *References*. Thus, each *Node* can reference another *Node* with the same *ReferenceType* only once. Figure 4 illustrates this model of a *Reference*.

<sup>1</sup> The integer ids of *Attributes* are defined in [UA Part 6].



**Figure 4 – Reference Model**

The *TargetNode* of a *Reference* may be in the same *AddressSpace* or in the *AddressSpace* of another OPC UA server. *TargetNodes* located in other servers are identified in OPC UA Services using a combination of the remote server name and the identifier assigned to the *Node* by the remote server.

Server names are URIs that identify OPC servers on the network. Each server name is unique within the scope of the network in which it is installed. If the network has Internet scope, then it is globally unique.

OPC UA does not require that the *TargetNode* exists, thus *References* may point to a *Node* that does not exist.

## 4.4 Variables

### 4.4.1 General

*Variables* are used to represent *values*. Two types of *Variables* are defined, *Properties* and *DataVariables*. They differ in the kind of data they represent and whether they can contain other *Variables*.

### 4.4.2 Properties

*Properties* are server-defined characteristics of *Objects*, *DataVariables* and other *Nodes*. *Properties* differ from *Attributes* in that they characterise *what* the *Node* represents, such as a device or a purchase order, instead of defining additional metadata that is used to instantiate all *Nodes* from a *NodeClass*.

For example, if a *DataVariable* is defined by a data structure that contains two fields, "startTime" and "endTime", it might have a *Property* specific to that data structure, such as "earliestStartTime".

To prevent recursion, *Properties* are not allowed to have *Properties* defined for them. To easily identify *Properties*, the *BrowseName* of a *Property* must be unique in the context of the *Node* containing the *Properties* (see Clause 5.6.3 for details).

A *Node* and its *Properties* must always reside in the same server.

### 4.4.3 DataVariables

*DataVariables* represent the content of an *Object*. For example, a file *Object* may be defined that contains a stream of bytes. The stream of bytes may be defined as a *DataVariable* that is an array of bytes. *Properties* may be used to expose the creation time and owner of the file *Object*.

As another example, function blocks in control systems might be represented as *Objects*. The parameters of the function block, such as its setpoints, may be represented as *DataVariables*. The function block *Object* might also have *Properties* that describe its execution time and its type.

*DataVariables* may have additional *DataVariables*, but only if they are complex. In this case, their *DataVariables* must always be elements of their complex definitions. Following the example

introduced by the description of *Properties* in Clause 4.4.2, the server could expose “start Time” and “end Time” as separate components of the data structure.

As another example, a complex *DataVariable* may define an aggregate of temperature values generated by three separate temperature transmitters that are also visible in the *AddressSpace*. In this case, this complex *DataVariable* could define *HasComponent References* from it to the individual temperature values that it is composed of.

## 4.5 TypeDefinitionNodes

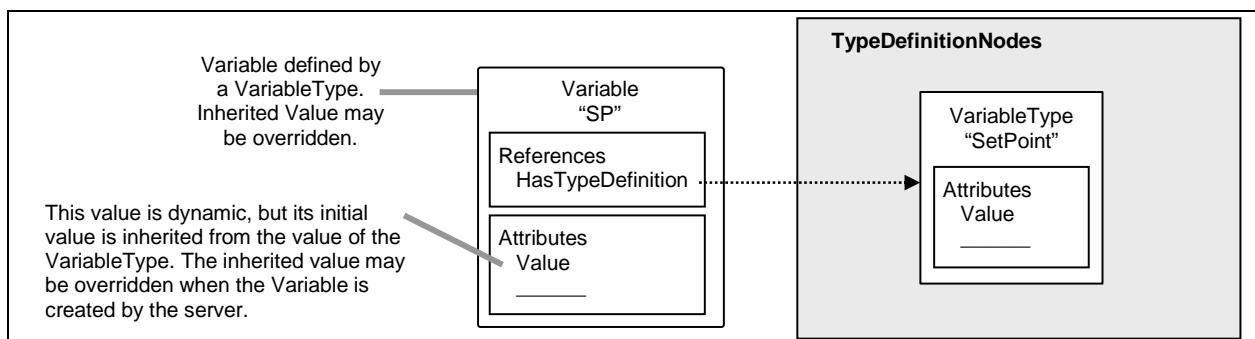
### 4.5.1 Overview

OPC UA requires servers to provide type definitions for *Objects* and *Variables*. The *HasTypeDefinition Reference* is used to link an instance with its type definition represented by a *TypeDefinitionNode*. This *Reference* is mandatory; OPC UA requires type definitions. However, [UA Part 5] defines a *BaseObjectType*, a *.PropertyType* and a *BaseDataVariableType* so a server can use such a base type if no more specialised type information is available.

In some cases, the *NodeId* used by the *HasTypeDefinition Reference* will be well-known to clients and servers. Organizations may define *TypeDefinitionNodes* that are well-known in the industry. Well-known *Nodelds* of *TypeDefinitionNodes* provide for commonality across UA servers and allow clients to interpret the *TypeDefinitionNode* without having to read it from the server. Therefore, servers may use well-known *Nodelds* without representing the corresponding *TypeDefinitionNodes* in their *AddressSpace*. However, the *TypeDefinitionNodes* must be provided for generic clients. These *TypeDefinitionNodes* may exist in another server.

The following example, illustrated in Figure 5, describes the use of the *HasTypeDefinition Reference*. In this example, a setpoint parameter “SP” is represented as a *DataVariable* in the *AddressSpace*. This *DataVariable* is part of an *Object* not shown in the figure.

To provide for a common setpoint definition that can be used by other *Objects*, a specialised *VariableType* is used. Each setpoint *DataVariable* that uses this common definition will have a *HasTypeDefinition Reference* that identifies the common “SetPoint” *VariableType*.



**Figure 5 – Example of a Variable Defined By a VariableType**

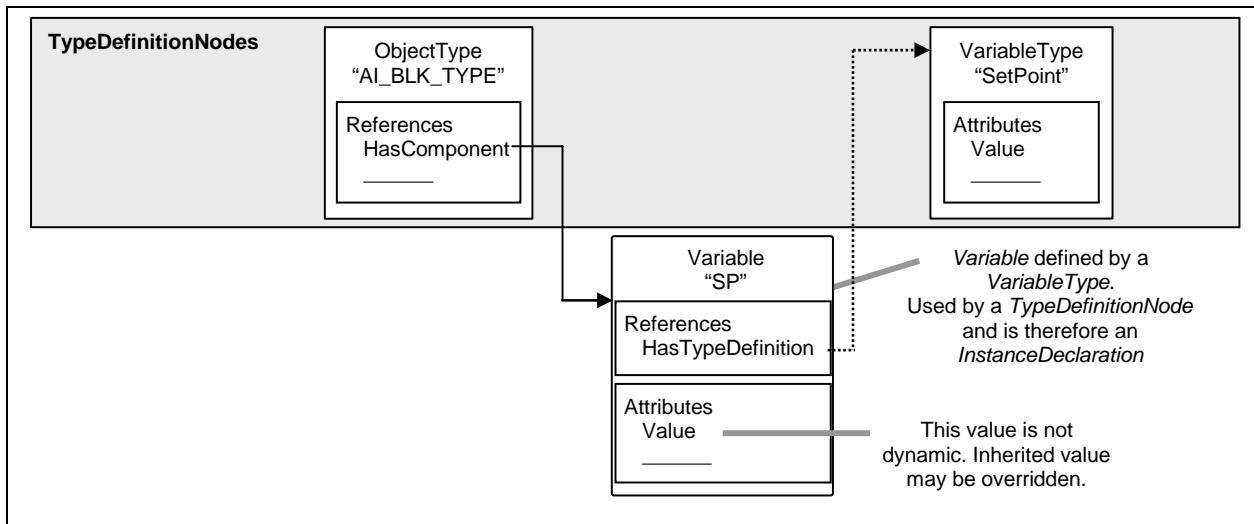
*Objects* and *Variables* inherit the *Attributes* specified by their *TypeDefinitionNode*.

### 4.5.2 Complex TypeDefinitionNodes and their InstanceDeclarations

*TypeDefinitionNodes* can be complex. A complex *TypeDefinitionNode* also defines *References* to other *Nodes* as part of the type definition. The *ModellingRules* defined in Clause 5.11 specify how those *Nodes* are handled when creating an instance of the type definition.

A *TypeDefinitionNode* typically references instances instead of other *TypeDefinitionNodes* to allow unique names for several instances of the same type, to define default values and to add *References* for those instances that are specific to this complex *TypeDefinitionNode* and not to the *TypeDefinitionNode* of the instance. For example, in Figure 6 the *ObjectType* “AI\_BLK\_TYPE”,

representing a function block, has a *HasComponent Reference* to a *Variable* "SP" of the *VariableType* "SetPoint". "AI\_BLK\_TYPE" could have an additional setpoint *Variable* of the same type using a different name. It could add a *Property* to the *Variable* that was not defined by its *TypeDefinitionNode* "SetPoint". And it could define a default value for "SP", that is, each instance of "AI\_BLK\_TYPE" would have a *Variable* "SP" initially set to this value.



**Figure 6 – Example of a Complex TypeDefinition**

This approach is commonly used in object-oriented programming languages in which the variables of a class are defined as instances of other classes. When the class is instantiated, each variable is also instantiated, but with the default values (constructor values) defined for the containing class. That is, typically, the constructor for the component class runs first, followed by the constructor for the containing class. The constructor for the containing class may override component values set by the component class.

To distinguish instances used for the type definitions from instances that represent real data, those instances are called *InstanceDeclarations*. However, this term is used to simplify this specification, if an instance is an *InstanceDeclaration* or not is only visible in the *AddressSpace* by following its *References*. Some instances may be shared and therefore referenced by *TypeDefinitionNodes*, *InstanceDeclarations* and instances. This is similar to class variables in object-oriented programming languages.

#### 4.5.3 Subtyping

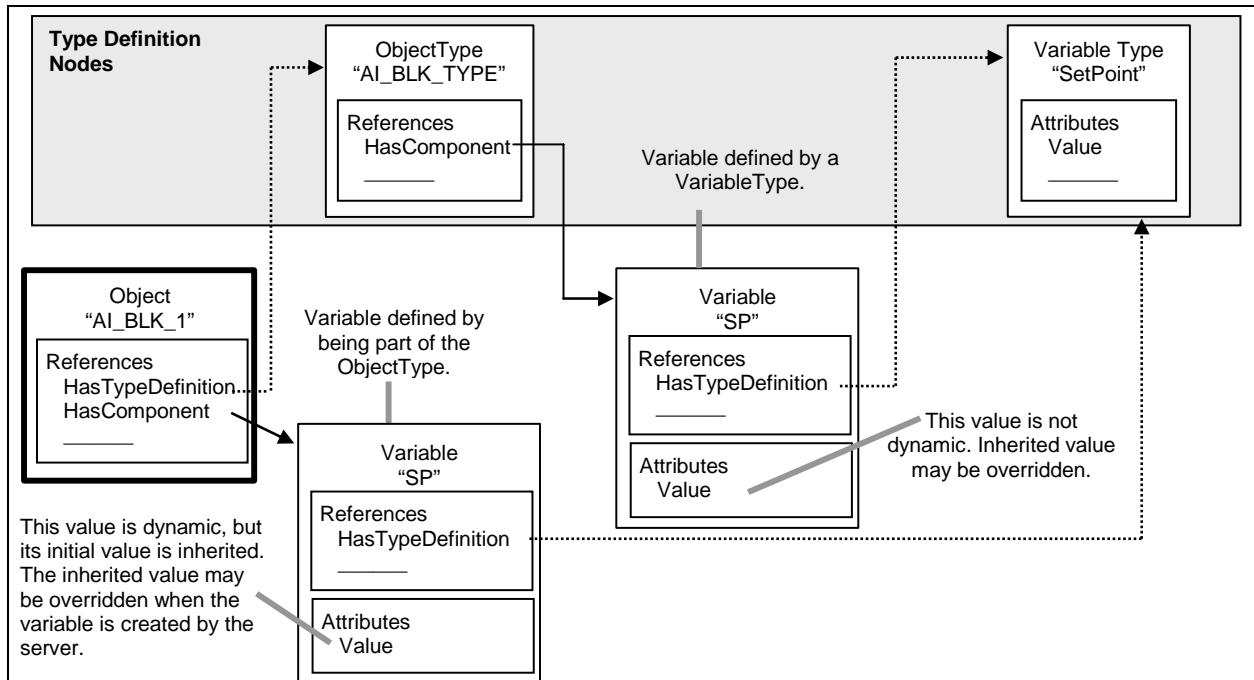
OPC UA allows subtyping of type definitions. The subtyping rules are defined in Clause 5.10. Subtyping of *ObjectTypes* and *VariableTypes* allows:

- clients that only know the supertype are able to handle an instance of the subtype as if it is an instance of the supertype;
- instances of the supertype can be replaced by instances of the subtype;
- the creation of specialised types that inherit common characteristics of the base type.

In other words, subtypes reflect the structure defined by their supertype but may add additional characteristics. For example, a vendor may wish to extend a general "TemperatureSensor" *VariableType* by adding a *Property* providing the next maintenance interval. The vendor would do this by creating a new *VariableType* which is a *TargetNode* for a *HasSubtype* reference from the original *VariableType* and adding the new *Property* to it.

#### 4.5.4 Instantiation of complex TypeDefinitionNodes

The instantiation of complex *TypeDefinitionNodes* depends on the *ModellingRules* defined in Clause 5.11. However, the intention is that instances of a type definition will reflect the structure defined by the *TypeDefinitionNode*. Figure 7 shows an instance of the *TypeDefinitionNode* “AI\_BLK\_TYPE”, where the *ModellingRule New*, defined in Clause 5.11.3.3, was applied for its containing *Variable*. Thus, an instance of “AI\_BLK\_TYPE”, called AI\_BLK\_1”, has a *HasTypeDefinition Reference* to “AI\_BLK\_TYPE”. It also contains a *Variable* “SP” having the same *BrowseName* as the *Variable* “SP” used by the *TypeDefinitionNode* and thereby reflects the structure defined by the *TypeDefinitionNode*.



**Figure 7 – Object and its Components defined by an ObjectType**

A client knowing the *ObjectType* “AI\_BLK\_TYPE” can use this knowledge to directly browse to the containing *Nodes* for each instance of this type. This allows programming against the *TypeDefinitionNode*. For example, a graphical element may be programmed in the client that handles all instances of “AI\_BLK\_TYPE” in the same way by showing the value of “SP”.

To allow this simple addressing, a *TypeDefinitionNode* or an *InstanceDeclaration* must never reference two *Nodes* having the same *BrowseName* using *hierarchical References*. Instances based on *InstanceDeclarations* must always keep the same *BrowseName* as the *InstanceDeclaration* they are derived from. A special Service defined in [UA Part 4] called *TranslateBrowsePathToNodelds* may be used to identify the instances based on the *InstanceDeclarations*. Using the simple *Browse Service* may not be sufficient since the uniqueness of the *BrowseName* is only required for *TypeDefinitionNodes* and *InstanceDeclarations*, not for other instances. Thus, “AI\_BLK\_1” may have another *Variable* with the *BrowseName* “SP”, although this one would not be derived from an *InstanceDeclaration* of the *TypeDefinitionNode*.

Instances derived from *InstanceDeclaration* must be of the same *TypeDefinitionNode* or a subtype of this *TypeDefinitionNode*.

A *TypeDefinitionNode* and its *InstanceDeclarations* must always reside in the same server. However, instances may point with their *HasTypeDefinition Reference* to a *TypeDefinitionNode* in a different server.

## 4.6 Event Model

### 4.6.1 Overview

The Event Model defines a general purpose eventing system that can be used in many diverse vertical markets.

*Events* represent specific transient occurrences. System configuration changes and system errors are examples of *Events*. *Event Notifications* report the occurrence of an *Event*. *Events* defined in this document are not directly visible in the OPC UA *AddressSpace*. *Objects* and *Views* can be used to subscribe to *Events*. The *EventNotifier Attribute* of those *Nodes* identifies if the *Node* allows subscribing to *Events*. Clients subscribe to such *Nodes* to receive *Notifications* of *Event* occurrences.

*Event Subscriptions* use the standard Monitoring and Subscription Services defined in [UA Part 4] to subscribe to *Event Notifications* of a *Node*.

Any UA server that supports eventing must expose at least one *Node* as *EventNotifier*. The server *Object* defined in [UA Part 5] is used for this purpose. All *Events* generated by the server are available via this standard server *Object*.

*Events* may also be exposed through other *Nodes* anywhere in the *AddressSpace*. These *Nodes* (identified via the *EventNotifier Attribute*) provide some subset of the *Events* generated by the server. The position in the *AddressSpace* dictates what this subset will be. For example, a process area *Object* representing a functional area of the process would provide *Events* originating from that area of the process only. It should be noted that this is only an example and it is fully up to the server to determine what *Events* should be provided by which *Node*. The only exception is the server *Object* that must be capable of providing all *Events* to a subscribing client.

### 4.6.2 EventTypes

Each *Event* is of a specific *EventType*. A server may support many types. This part defines the *BaseEventType* that all other *EventTypes* derive from. It is expected that other companion specifications will define additional *EventTypes* deriving from the base types defined in this part.

The *EventTypes* supported by a server are exposed in the *AddressSpace* of a server. *EventTypes* are represented as *ObjectTypes* in the *AddressSpace* and do not have a special *NodeClass* associated to them. [UA Part 5] defines how a server exposes the *EventTypes* in detail.

*EventTypes* defined in this document are specified as abstract and therefore never instantiated in the *AddressSpace*. Event occurrences of those *EventTypes* are only exposed via a *Subscription*. *EventTypes* exist in the *AddressSpace* to allow clients to discover the *EventType*. This information is used by a client when establishing and working with *Event Subscriptions*. *EventTypes* defined by other parts of this multi-part specification or companion specifications as well as server specific *EventTypes* may be defined as not abstract and therefore instances of those *EventTypes* may be visible in the *AddressSpace* although *Events* of those *EventTypes* are also accessible via the *Event Notification* mechanisms.

Standard *EventTypes* are described in Clause 8. Their representation in the *AddressSpace* is specified in [UA Part 5].

### 4.6.3 Event Categorization

*Events* can be categorised by using two mechanisms in combination or individually. New *EventTypes* can be defined and *Events* can be organised by using the *Event ReferenceTypes* described in Clause 6.16 and 6.17.

New *EventTypes* can be defined as subtypes of existing *EventTypes* that do not extend an existing type. They are used only to identify an event as being of the new *EventType*. For example, the

*EventType* DeviceFailureEventType could be subtyped into TransmitterFailureEventType and ComputerFailureEventType. These new subtypes would not add new *Properties* or change the semantic inherited from the DeviceFailureEventType other than purely for categorization of the *Events*.

*Event* sources can also be organised into groups by using the *Event ReferenceTypes* described in Clause 6.16 and 6.17. For example, a server may define *Objects* in the *AddressSpace* representing *Events* related to physical devices, or *Event* areas of a plant or functionality contained in the server. *Event References* would be used to indicate which *Event* sources represent physical devices and which ones represent some server-based functionality. In addition, *References* can be used to group the physical devices or server-based functionality into hierarchical *Event* areas. In some cases, an *Event* source may be categorised as being both a device and a server function. In this case, two relationships would be established. Refer to the description of the *Event ReferenceTypes* for additional examples.

Clients can select a category or categories of *Events* by defining content filters that include terms specifying the *EventType* of the *Event* or a grouping of *Event* sources. The two mechanisms allow for a single *Event* to be categorised in multiple manners. A client could obtain all *Events* related to a physical device or all failures of a particular device.

#### 4.7 Methods

*Methods* are “lightweight” functions, whose scope is bounded by an owning<sup>2</sup> *Object*, similar to the methods of a class in object-oriented programming. *Methods* are invoked by a client, proceed to completion on the server and return the result to the client. The lifetime of the *Method*’s invocation instance begins when the client calls the *Method* and ends when the result is returned.

While *Methods* may affect the state of the owning *Object*, they have no explicit state of their own. In this sense, they are stateless. *Methods* can have a varying number of input arguments and return resultant arguments. Each *Method* is described by a *Node* of the *Method NodeClass*. This *Node* contains the metadata that identifies the *Method*’s arguments and describes its behaviour.

*Methods* are invoked by using the Call Service defined in [UA Part 4]. Each *Method* is invoked within the context of an existing session. If the session is terminated during *Method* execution, the results of the *Method*’s execution cannot be returned to the client and are discarded. In that case, the *Method* execution is undefined, that is, the *Method* may be executed until it is finished or it may be aborted.

Clients discover the *Methods* supported by a server by browsing for the owning *Objects References* that identify their supported *Methods*.

### 5 Standard NodeClasses

#### 5.1 Overview

This clause defines the *NodeClasses* used to define *Nodes* in the OPC UA *AddressSpace*. *NodeClasses* are derived from a common, *Base NodeClass*. This *NodeClass* is defined first, followed by those used to organise the *AddressSpace* and then by the *NodeClasses* used to represent *Objects*.

The *NodeClasses* defined to represent *Objects* fall into three categories: those used to define instances, those used to define types for those instances and those used to define data types. Clause 5.10 describes the rules for subtyping and Clause 5.11 the rules for instantiation of the type definitions.

---

<sup>2</sup> The owning *Object* is specified in the service call when invoking the *Method*.

## 5.2 Base NodeClass

### 5.2.1 General

The OPC UA Address Space Model defines a *Base NodeClass* from which all other *NodeClasses* are derived. The derived *NodeClasses* represent the various components of the OPC UA Object Model (see Clause 4.2). The *Attributes* of the *Base NodeClass* are specified in Table 2. There are no *References* specified for the *Base NodeClass*.

**Table 2 - Base NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
NodeId	M	NodeId	See Clause 5.2.2
NodeClass	M	NodeClass	See Clause 5.2.3
BrowseName	M	QualifiedName	See Clause 5.2.4
DisplayName	M	LocalizedText	See Clause 5.2.5
Description	O	LocalizedText	See Clause 5.2.6
<b>References</b>			
No <i>References</i> specified for this <i>NodeClass</i>			

### 5.2.2 NodId

Nodes are unambiguously identified using a constructed identifier called the *NodId*. Some servers may accept alternative *NodIds* in addition to the canonical *NodId* represented in this *Attribute*. The structure of the *NodId* is defined in Clause 7.2.

### 5.2.3 NodeClass

The *NodeClass Attribute* identifies the *NodeClass* of a *Node*. Its data type is defined in Clause 7.21.

### 5.2.4 BrowseName

Nodes have a *BrowseName Attribute* that is used as a non-localised human-readable name when browsing the *AddressSpace* to create paths out of *BrowseNames*. The *TranslateBrowsePathToNodId Service* defined in [UA Part 4] can be used to follow a path constructed of *BrowseNames*.

A *BrowseName* should never be used to display the name of a *Node*. The *DisplayName* should be used instead for this purpose.

Unlike *NodIds*, the *BrowseName* cannot be used to unambiguously identify a *Node*. Different *Nodes* may have the same *BrowseName*.

Clause 7.3 defines the structure of the *BrowseName*. It contains a namespace and a string. The namespace is provided to make the *BrowseName* unique in some cases in the context of a *Node* (e.g. *Properties* of a *Node*) although not unique in the context of the server. If different organizations define standard *BrowseNames* for *Properties*, the namespace of the *BrowseName* provided by the organization makes the *BrowseName* unique, although different organizations may use the same string having a slightly different meaning.

Servers may often choose to use the same namespace for the *NodId* and the *BrowseName*. However, if they want to provide a standard *Property*, its *BrowseName* must have the namespace of the standards body although the namespace of the *NodId* reflects something else, for example the local server.

It is recommended that standard bodies defining standard type definitions use their namespace for the *NodId* of the *TypeDefinitionNode* as well as for the *BrowseName* of the *TypeDefinitionNode*.

## 5.2.5 DisplayName

The *DisplayName Attribute* contains the localised name of the *Node*. Clients should use this *Attribute* if they want to display the name of the *Node* to the user. They should not use the *BrowseName* for this purpose. The server may maintain one or more localised representations for each *DisplayName*. Clients negotiate the locale to be returned when they open a session with the server. Refer to [UA Part 4] for a description of session establishment and locales. Clause 7.5 defines the structure of the *DisplayName*.

## 5.2.6 Description

The optional *Description Attribute* must explain the meaning of the *Node* in a localised text using the same mechanisms as described for the *DisplayName* in Clause 5.2.5.

# 5.3 ReferenceType NodeClass

## 5.3.1 General

*References* are defined as instances of *ReferenceType Nodes*. *ReferenceType Nodes* are visible in the *AddressSpace* and are defined using the *ReferenceType NodeClass* as specified in Table 3. In contrast, a *Reference* is an inherent part of a *Node* and no *NodeClass* is used to represent *References*.

OPC UA defines a set of standard *ReferenceTypes* provided as an inherent part of the OPC UA Address Space Model. These *ReferenceTypes* are defined in Clause 6 and their representation in the *AddressSpace* is defined in [UA Part 5]. Servers may also define *ReferenceTypes*. In addition, [UA Part 4] defines *NodeManagement Services* that allow clients to add *ReferenceTypes* to the *AddressSpace*.

**Table 3 – ReferenceType NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE it is an abstract <i>ReferenceType</i> , i.e. no <i>References</i> of this type must exist, only of its subtypes. FALSE it is not an abstract <i>ReferenceType</i> , i.e. <i>References</i> of this type can exist.
Symmetric	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE the meaning of the <i>ReferenceType</i> is the same as seen from both the <i>SourceNode</i> and the <i>TargetNode</i> . FALSE the meaning of the <i>ReferenceType</i> as seen from the <i>TargetNode</i> is the inverse of that as seen from the <i>SourceNode</i> .
InverseName	O	LocalizedText	The inverse name of the <i>Reference</i> , i.e. the meaning of the <i>ReferenceType</i> as seen from the <i>TargetNode</i> .
<b>References</b>			
HasProperty	0..*		Used to identify the Properties (See Clause 5.3.3.2)
HasSubtype	0..*		Used to identify subtypes (See Clause 5.3.3.3)
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.

## 5.3.2 Attributes

The *ReferenceType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. The inherited *BrowseName Attribute* is used to specify the meaning of the *ReferenceType* as seen from the *SourceNode*. For example, the *ReferenceType* with the

*BrowseName* “Contains” is used in *References* that specify that the *SourceNode* contains the *TargetNode*. The inherited *DisplayName* *Attribute* contains a translation of the *BrowseName*.

The *BrowseName* of a *ReferenceType* must be unique in a server. It is not allowed that two different *ReferenceTypes* have the same *BrowseName*.

The *IsAbstract* *Attribute* indicates if the *ReferenceType* is abstract. Abstract *ReferenceTypes* can not be instantiated and are used only for organizational reasons, e.g. to specify some general semantics or constraints that are inherited to its subtypes.

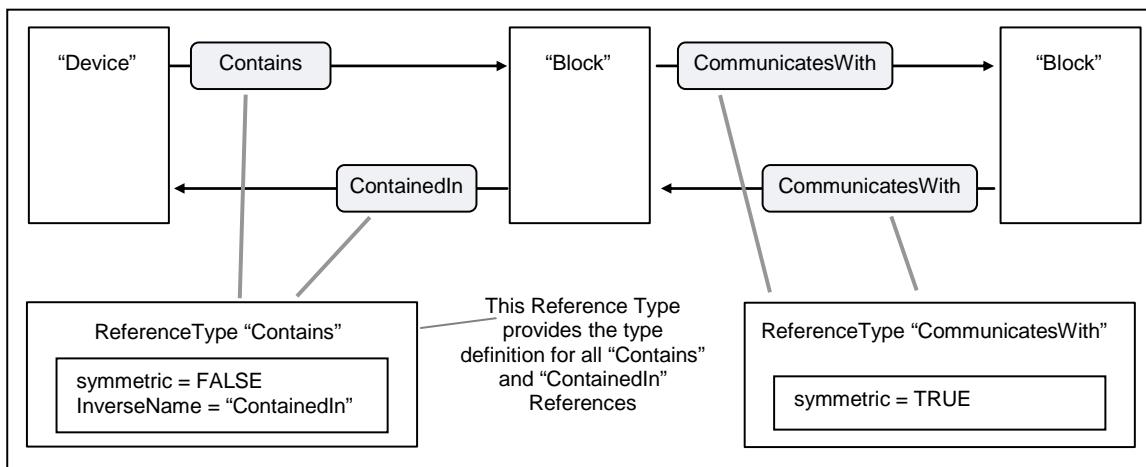
The *Symmetric* *Attribute* is used to indicate whether or not the meaning of the *ReferenceType* is the same for both the *SourceNode* and *TargetNode*.

If a *ReferenceType* is symmetric, the *InverseName* *Attribute* must be omitted. Examples of symmetric *ReferenceTypes* are “Connects To” and “Communicates With”. Both imply the same semantic coming from the *SourceNode* or the *TargetNode*.

If the *ReferenceType* is non-symmetric and not abstract, the *InverseName* *Attribute* must be set. The *InverseName* *Attribute* specifies the meaning of the *ReferenceType* as seen from the *TargetNode*. Examples of non-symmetric *ReferenceTypes* include “Contains” and “Contained In”, and “Receives From” and “Sends To”.

*References* that use the *InverseName*, such as “Contained In” *References*, are referred to as inverse *References*.

Figure 8 provides examples of symmetric and non-symmetric *References* and the use of the *BrowseName* and the *InverseName*.



**Figure 8 – Symmetric and Non-Symmetric References**

It may not always be possible for servers to instantiate both forward and inverse *References* for non-symmetric *ReferenceTypes* as shown in this figure. When they do, the *References* are referred to as *bidirectional*. Although not required, it is recommended that all *hierarchical References* be instantiated as bidirectional to ensure browse connectivity. A bidirectional *Reference* is modelled as two separate *References*.

As an example of a *unidirectional Reference*, it is often the case that a subscriber knows its publisher, but its publisher does not know its subscribers. The subscriber would have a “Subscribes To” *Reference* to the publisher, without the publisher having the corresponding “Publishes To” inverse *References* to its subscribers.

The *DisplayName* and the *InverseName* are the only standardised places to indicate the semantic of a *ReferenceType*. There may be more complex semantics associated with a *ReferenceType* than

can be expressed in those *Attributes* (e.g. the semantic of *HasSubtype*). OPC UA does not specify how this semantic should be exposed. However, the *Description Attribute* can be used for this purpose. OPC UA does provide a semantic for the standard *ReferenceTypes* specified in Clause 6.

A *ReferenceType* can have constraints restricting its use. For example, it can specify that starting from *Node A* and only following *References* of this *ReferenceType* or one of its subtypes must never be able to return to A, that is, a “No Loop” constraint.

OPC UA does not specify how those constraints could or should be made available in the *AddressSpace*. Nevertheless, for the standard *ReferenceTypes*, some constraints are specified in Clause 6. OPC UA does not restrict the kind of constraints valid for a *ReferenceType*. It can, for example, also affect an *ObjectType*. The restriction that a *ReferenceType* can only be used relating *Nodes* of some *NodeClasses* with a defined cardinality is a special constraint of a *ReferenceType*.

### 5.3.3 References

#### 5.3.3.1 General

*HasSubtype References* and *HasProperty References* are the only *ReferenceTypes* that may be used with *ReferenceType Nodes* as *SourceNode*. *ReferenceType Nodes* must not be the *SourceNode* of other types of *References*.

#### 5.3.3.2 HasProperty References

*HasProperty References* are used to identify the *Properties* of a *ReferenceType* and must only refer to *Nodes* of the *Variable NodeClass*.

The standard *Property NodeVersion* is used to indicate the version of the *ReferenceType*.

There are no additional standard *Properties* defined for *ReferenceTypes* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *ReferenceTypes*.

#### 5.3.3.3 HasSubtype References

*HasSubtype References* are used to define subtypes of *ReferenceTypes*. It is not required to provide the *HasSubtype Reference* for the supertype, but it is required that the subtype provides the inverse *Reference* to its supertype. The following rules for subtyping apply:

1. The semantic of a *ReferenceType* (e.g. “spans a hierarchy”) is inherited to its subtypes and can be refined there (e.g. “spans a special hierarchy”). The *DisplayName*, and also the *InverseName* for non-symmetric *ReferenceTypes*, reflect the specialization.
2. If a *ReferenceType* specifies some constraints (e.g. “allow no loops”) this is inherited and can only be refined (e.g. inheriting “no loops” could be refined as “must be a tree – only one parent”) but not lowered (e.g. “allow loops”).
3. The constraints concerning which *NodeClasses* can be referenced are also inherited and can only be further restricted. That is, if a *ReferenceType* “A” is not allowed to relate an *Object* with an *ObjectType*, this is also true for its subtypes.
4. A *ReferenceType* can have only zero or one super type. The *ReferenceType* hierarchy does not support multiple inheritance.

## 5.4 View NodeClass

Underlying systems are often large and clients often have an interest in only a specific subset of the data. They do not need, or want, to be burdened with viewing *Nodes* in the *AddressSpace* for which they have no interest.

To address this problem, OPC UA defines the concept of a *View*. Each *View* defines a subset of the *Nodes* in the *AddressSpace*. The entire *AddressSpace* is the default *View*. Each *Node* in a *View* may contain only a subset of its *References*, as defined by the creator of the *View*. The *View Node* acts as the root for the *Nodes* in the *View*. *Views* are defined using the *View NodeClass*, which is specified in Table 4.

**Table 4 – View NodeClass**

Name	Use	Data Type	Description																		
<b>Attributes</b>																					
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2																		
ContainsNoLoops	M	Boolean	If set to “true” this <i>Attribute</i> indicates that following <i>References</i> in the context of the <i>View</i> contains no loops, i.e. starting from a <i>Node</i> “A” contained in the <i>View</i> and following the <i>References</i> in the context of the <i>View Node</i> “A” will not be reached again. It does not specify that there is only one path starting from the <i>View Node</i> to reach a <i>Node</i> contained in the <i>View</i> . If set to “false” this <i>Attribute</i> indicates that following <i>References</i> in the context of the <i>View</i> may lead to loops.																		
EventNotifier	M	Byte	The <i>EventNotifier Attribute</i> is used to indicate if the <i>Node</i> can be used to subscribe to <i>Events</i> or the read / write historic <i>Events</i> . The <i>EventNotifier</i> is an 8-bit unsigned integer with the structure defined in the following table:																		
			<table border="1"> <thead> <tr> <th>Field</th><th>Bit</th><th>Description</th></tr> </thead> <tbody> <tr> <td>SubscribeTo Events</td><td>0</td><td>Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i>, 1 means can be used to subscribe to <i>Events</i>).</td></tr> <tr> <td>Reserved</td><td>1</td><td>Reserved for future use. Must always be zero.</td></tr> <tr> <td>HistoryRead</td><td>2</td><td>Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>HistoryWrite</td><td>3</td><td>Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr> </tbody> </table> <p>The second two bits also indicate if the history of the <i>Events</i> is available via the OPC UA server.</p>	Field	Bit	Description	SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i> ).	Reserved	1	Reserved for future use. Must always be zero.	HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Must always be zero.
Field	Bit	Description																			
SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i> ).																			
Reserved	1	Reserved for future use. Must always be zero.																			
HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Must always be zero.																			
<b>References</b>																					
HierarchicalReferences	0..*		Top level <i>Nodes</i> in a <i>View</i> are referenced by <i>hierarchical References</i> (see Clause 6.3).																		
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>View</i> .																		
<b>Standard Properties</b>																					
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.																		
ViewVersion	O	UInt32	The version number for the <i>View</i> . When <i>Nodes</i> are added to or removed from a <i>View</i> , the value of the <i>ViewVersion Property</i> is updated. Clients may detect changes to the composition of a <i>View</i> using this <i>Property</i> . The value of the <i>ViewVersion</i> must always be greater than 0.																		

The *View NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. It also defines two additional *Attributes*.

The mandatory *ContainsNoLoops Attribute* is set to false if the server is not able to identify if the *View* contains loops or not.

The mandatory *EventNotifier Attribute* identifies if the *View* can be used to subscribe to *Events* that either occur in the content of the *View* or as *ModelChangeEvents* of the content of the *View* or to read / write the history of the *Events*.

Views are defined by the server. The browsing and querying Services defined in [UA Part 4] expect the *NodeId* of a *View Node* to provide these Services in the context of the *View*.

*HasProperty References* are used to identify the *Properties* of a *View*. The standard *Property NodeVersion* is used to indicate the version of the *View Node*. The *ViewVersion Property* indicates the version of the content of the *View*. In contrast to the *NodeVersion*, the *ViewVersion Property* is updated even if *Nodes* not directly referenced by the *View Node* are added to or deleted from the *View*. This *Property* is optional because it may not be possible for servers to detect changes in the *View* contents. Servers may also generate a *ModelChangeEvent* described in Clause 8.20 if *Nodes* are added to or deleted from the *View*. There are no additional standard *Properties* defined for *Views* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *Views*.

*Views* can be the *SourceNode* of any *hierarchical Reference*. They must not be the *SourceNode* of any *non-hierarchical Reference*.

## 5.5 Objects

### 5.5.1 Object NodeClass

*Objects* are used to represent systems, system components, real-world objects and software objects. *Objects* are defined using the *Object NodeClass*, specified in Table 5.

**Table 5 – Object NodeClass**

Name	Use	Data Type	Description																		
<b>Attributes</b>																					
Base NodeClass Attributes																					
EventNotifier	M	Byte	<p>The <i>EventNotifier Attribute</i> is used to indicate if the <i>Node</i> can be used to subscribe to <i>Events</i> or the read / write historic <i>Events</i>.  The <i>EventNotifier</i> is an 8-bit unsigned integer with the structure defined in the following table:</p> <table border="1"> <thead> <tr> <th>Field</th><th>Bit</th><th>Description</th></tr> </thead> <tbody> <tr> <td>SubscribeTo Events</td><td>0</td><td>Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i>, 1 means can be used to subscribe to <i>Events</i>).</td></tr> <tr> <td>Reserved</td><td>1</td><td>Reserved for future use. Must always be zero.</td></tr> <tr> <td>HistoryRead</td><td>2</td><td>Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>HistoryWrite</td><td>3</td><td>Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr> </tbody> </table> <p>The second two bits also indicate if the history of the <i>Events</i> is available via the OPC UA server.</p>	Field	Bit	Description	SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i> ).	Reserved	1	Reserved for future use. Must always be zero.	HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Must always be zero.
Field	Bit	Description																			
SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i> ).																			
Reserved	1	Reserved for future use. Must always be zero.																			
HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Must always be zero.																			
<b>References</b>																					
HasComponent	0..*		<i>HasComponent References</i> identify the <i>DataVariables</i> , the <i>Methods</i> and <i>Objects</i> contained in the <i>Object</i> .																		
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>Object</i> .																		
HasModellingRule	0..1		<i>Objects</i> can point to at most one <i>ModellingRule Object</i> using a <i>HasModellingRule Reference</i> (see Clause 5.11 for details on <i>ModellingRules</i> ). If no <i>ModellingRule</i> is specified, the default <i>ModellingRule None</i> is used.																		
HasTypeDefinition	1		The <i>HasTypeDefinition Reference</i> points to the type definition of the <i>Object</i> . Each <i>Object</i> must have exactly one type definition and therefore be the <i>SourceNode</i> of exactly one <i>HasTypeDefinition Reference</i> pointing to an <i>ObjectType</i> . See Clause 4.5 for a description of type definitions.																		
Organizes	0..*		This <i>Reference</i> should be used only for <i>Objects</i> of the <i>ObjectType FolderType</i> (see Clause 5.5.3).																		
HasDescription	0..1		This <i>Reference</i> should be used only for <i>Objects</i> of the <i>ObjectType DataTypeEncodingType</i> (see Clause 5.8.3).																		
References	0..*		<i>Objects</i> may contain other <i>References</i> .																		
<b>Standard Properties</b>																					
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute value changes</i> do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.																		

The *Object NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2.

The mandatory *EventNotifier Attribute* identifies whether the *Object* can be used to subscribe to *Events* or to read and write the history of the *Events*.

The *Object NodeClass* uses the *HasComponent Reference* to define the *DataVariables*, *Objects* and *Methods* of an *Object*.

It uses the *HasProperty Reference* to define the *Properties* of an *Object*. The standard *Property NodeVersion* is used to indicate the version of the *Object*. There are no additional standard

*Properties* defined for *Objects* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *Objects*.

To specify its *ModellingRule*, an *Object* can use at most one *HasModellingRule Reference* pointing to a *ModellingRule Object*. *ModellingRules* are defined in Clause 5.11.

The *HasTypeDefintion Reference* points to the *ObjectType* used as type definition of the *Object*.

*Objects* may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *Objects*. Standard *ReferenceTypes* are described in Clause 6.

If the *Object* is used as *InstanceDeclaration* (see Clause 4.5) all *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of this *Object*.

If the *Object* is created based on an *InstanceDeclaration*, it must have the same *BrowseName* as its *InstanceDeclaration*.

### 5.5.2 ObjectType NodeClass

*ObjectTypes* provide definitions for *Objects*. *ObjectTypes* are defined using the *ObjectType NodeClass*, which is specified in Table 6.

**Table 6 – ObjectType NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE      it is an abstract <i>ObjectType</i> , i.e. no <i>Objects</i> of this type must exist, only of its subtypes. FALSE     it is not an abstract <i>ObjectType</i> , i.e. <i>Objects</i> of this type can exist.
<b>References</b>			
HasComponent	0..*		<i>HasComponent References</i> identify the <i>DataVariables</i> , the <i>Methods</i> , and <i>Objects</i> contained in the <i>ObjectType</i> . If and how the referenced <i>Nodes</i> are instantiated when an <i>Object</i> of this type is instantiated, is specified in Clause 5.11.
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>ObjectType</i> . If and how the <i>Properties</i> are instantiated when an <i>Object</i> of this type is instantiated, is specified in Clause 5.11.
HasSubtype	0..*		<i>HasSubtype References</i> identify <i>ObjectTypes</i> that are subtypes of this type. The inverse <i>SubtypeOf Reference</i> identifies the parent type of this type.
GeneratesEvent	0..*		<i>GeneratesEvent References</i> identify the type of <i>Events</i> instances of this type may generate.
References	0..*		<i>ObjectTypes</i> may contain other <i>References</i> that can be instantiated by <i>Objects</i> defined by this <i>ObjectType</i> .
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.

The *ObjectType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. The additional *IsAbstract Attribute* indicates if the *ObjectType* is abstract or not.

The *ObjectType NodeClass* uses the *HasComponent References* to define the *DataVariables*, *Objects*, and *Methods* for it.

The *HasProperty Reference* is used to identify the *Properties*. The standard *Property NodeVersion* is used to indicate the version of the *ObjectType*. There are no additional standard *Properties* defined for *ObjectTypes* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *ObjectTypes*.

*HasSubtype References* are used to subtype *ObjectTypes*. *ObjectType* subtypes inherit the general semantics from the parent type. The general rules for subtyping apply as defined in Clause 5.10. It is not required to provide the *HasSubtype Reference* for the supertype, but it is required that the subtype provides the inverse *Reference* to its supertype.

*GeneratesEvent References* identify the type of *Events* that instances of the *ObjectType* may generate. These *Objects* may be the source of an *Event* of the specified type or one of its subtypes. Servers should make *GeneratesEvent References* bidirectional *References*. However, it is allowed to be unidirectional when the server is not able to expose the inverse direction pointing from the *EventType* to each *ObjectType* supporting the *EventType*. Note that the *EventNotifier Attribute* of an *Object* and the *GeneratesEvent References* of its *ObjectType* are completely unrelated. *Objects* that can generate *Events* may not be used as *Objects* to which clients subscribe to get the corresponding *Event* notifications.

*GeneratesEvent References* are optional, i.e. *Objects* may generate *Events* of an *EventType* that is not exposed by its *ObjectType*.

*ObjectTypes* may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *ObjectTypes*. Standard *ReferenceTypes* are described in Clause 6.

All *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of an *ObjectType* (see Clause 4.5).

### 5.5.3 Standard ObjectType FolderType

The *ObjectType FolderType* is formally defined in [UA Part 5]. Its purpose is to provide *Objects* that have no other semantic than organizing of the *AddressSpace*. A special *ReferenceType* is introduced for those *Folder Objects*, the *Organizes ReferenceType*. The *SourceNode* of such a *Reference* should always be an *Object* of the *ObjectType FolderType*; the *TargetNode* can be of any *NodeClass*. *Organizes References* can be used in any combination with *Aggregates References* (*HasComponent*, *HasProperty*, etc.; see Clause 6.5) and do not prevent loops. Thus, they can be used to span multiple hierarchies.

## 5.6 Variables

### 5.6.1 General

Two types of *Variables* are defined, *Properties* and *DataVariables*. Although they differ in the way they are used as described in Clause 4.4 and have different constraints described in the following subclauses, they use the same *NodeClass* described in Clause 5.6.2. The constraints of *Properties* based on this *NodeClass* are defined in Clause 5.6.3, the constraints of *DataVariables* in Clause 5.6.4.

### 5.6.2 Variable NodeClass

*Variables* are used to represent values which may be simple or complex. *Variables* are defined by *VariableTypes*, specified in Clause 5.6.5.

*Variables* are always defined as *Properties* or *DataVariables* of other *Nodes* in the *AddressSpace*. They are never defined by themselves. A *Variable* is always part of at least one other *Node*, but may be related to any number of other *Nodes*. *Variables* are defined using the *Variable NodeClass*, specified in Table 7.

**Table 7 – Variable NodeClass**

Name	Use	Data Type	Description																		
<b>Attributes</b>																					
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2																		
Value	M	Defined by the <i>DataType Attribute</i>	The most recent value of the <i>Variable</i> that the server has. Its data type is defined by the <i>DataType Attribute</i> . It is the only <i>Attribute</i> that does not have a data type associated with it. This allows all <i>Variables</i> to have a value defined by the same <i>Value Attribute</i> .																		
DataType	M	NodeId	<i>NodeId</i> of the <i>DataType</i> definition for the <i>Value Attribute</i> . Standard <i>DataTypes</i> are defined in Clause 7.																		
ArraySize	M	Int32	<p>This <i>Attribute</i> indicates whether the <i>Value Attribute</i> of the <i>Variable</i> is an array.</p> <p>If it is not an array, the <i>ArraySize</i> is set to -1.</p> <p>If it is an array, the <i>ArraySize</i> specifies the number of elements in the array. The value 0 is used to indicate an array whose size has not been allocated or is not known.</p> <p>For example, if a <i>Variable</i> is defined by the following C array:</p> <pre>Int32 myArray[346];</pre> <p>then this <i>Variable's</i> <i>DataType</i> would point to an Int32 and the <i>Variable's</i> <i>ArraySize</i> has the value 346.</p>																		
AccessLevel	M	Byte	<p>The <i>AccessLevel Attribute</i> is used to indicate how the <i>Value</i> of a <i>Variable</i> can be accessed (read/write) and if it contains current and/or historic data. The <i>AccessLevel</i> does not take any user access rights into account, i.e. although the <i>Variable</i> is writeable this may be restricted to a certain user / user group.</p> <p>The <i>AccessLevel</i> is an 8-bit unsigned integer with the structure defined in the following table:</p> <table border="1"> <thead> <tr> <th>Field</th><th>Bit</th><th>Description</th></tr> </thead> <tbody> <tr> <td>CurrentRead</td><td>0</td><td>Indicates if the current value is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>CurrentWrite</td><td>1</td><td>Indicates if the current value is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>HistoryRead</td><td>2</td><td>Indicates if the history of the value is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>HistoryWrite</td><td>3</td><td>Indicates if the history of the value is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr> </tbody> </table> <p>The first two bits also indicate if a current value of this <i>Variable</i> is available and the second two bits indicates if the history of the <i>Variable</i> is available via the OPC UA server.</p>	Field	Bit	Description	CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).	CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).	HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Must always be zero.
Field	Bit	Description																			
CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).																			
CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).																			
HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Must always be zero.																			
UserAccessLevel	M	Byte	<p>The <i>UserAccessLevel Attribute</i> is used to indicate how the <i>Value</i> of a <i>Variable</i> can be accessed (read/write) and if it contains current or historic data taking user access rights into account.</p> <p>The <i>UserAccessLevel</i> is an 8-bit unsigned integer with the structure defined in the following table:</p> <table border="1"> <thead> <tr> <th>Field</th><th>Bit</th><th>Description</th></tr> </thead> <tbody> <tr> <td>CurrentRead</td><td>0</td><td>Indicates if the current value is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>CurrentWrite</td><td>1</td><td>Indicates if the current value is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>HistoryRead</td><td>2</td><td>Indicates if the history of the value is readable (0 means not readable, 1 means readable).</td></tr> <tr> <td>HistoryWrite</td><td>3</td><td>Indicates if the history of the value is writable (0 means not writable, 1 means writable).</td></tr> <tr> <td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr> </tbody> </table> <p>The first two bits also indicate if a current value of this <i>Variable</i> is available and the second two bits indicate if the history of the <i>Variable</i> is available via the OPC UA server.</p>	Field	Bit	Description	CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).	CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).	HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Must always be zero.
Field	Bit	Description																			
CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).																			
CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).																			
HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Must always be zero.																			
MinimumSamplingInterval	O	Int32	<p>The <i>MinimumSamplingInterval Attribute</i> indicates how “current” the <i>Value</i> of the <i>Variable</i> will be kept. It specifies (in milliseconds) how fast the server can reasonably sample the value for changes (see [UA Part 4] for a detailed description of sampling interval).</p> <p>A <i>MinimumSamplingInterval</i> of 0 indicates that the server is to monitor the item continuously. A <i>MinimumSamplingInterval</i> of -1 means indeterminate.</p>																		

<b>References</b>			
HasModellingRule	0..1		<i>Variables</i> can point to at most one <i>ModellingRule Object</i> using a <i>HasModellingRule Reference</i> (see Clause 5.11 for details on <i>ModellingRules</i> ). If no <i>ModellingRule</i> is specified, the default <i>ModellingRule None</i> is used.
HasProperty	0..*		<i>HasProperty References</i> are used to identify the <i>Properties</i> of a <i>DataVariable</i> . <i>Properties</i> are not allowed to be the <i>SourceNode</i> of <i>HasProperty References</i> .
HasComponent	0..*		<i>HasComponent References</i> are used by complex <i>DataVariables</i> to identify their composed <i>DataVariables</i> . <i>Properties</i> are not allowed to use this <i>Reference</i> .
HasTypeDefinition	1		The <i>HasTypeDefinition Reference</i> points to the type definition of the <i>Variable</i> . Each <i>Variable</i> must have exactly one type definition and therefore be the <i>SourceNode</i> of exactly one <i>HasTypeDefinition Reference</i> pointing to a <i>VariableType</i> . See Clause 4.5 for a description of type definitions.
References	0..*		<i>Data Variables</i> may be the <i>SourceNode</i> of any other <i>References</i> . <i>Properties</i> may only be the <i>SourceNode</i> of any <i>non-hierarchical Reference</i> .
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>DataVariable</i> . It does not apply for <i>Properties</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes except for the <i>DataType Attribute</i> do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed. Although the relationship of a <i>Variable</i> to its <i>DataType</i> is not modelled using <i>References</i> , changes to the <i>DataType Attribute</i> of a <i>Variable</i> lead to an update of the <i>NodeVersion Property</i> .
VariableTimeZone	O	Int32	The <i>VariableTimeZone Property</i> is only used for <i>DataVariables</i> . It does not apply for <i>Properties</i> . This <i>Property</i> specifies the time difference (in minutes) between the <i>SourceTimestamp (UTC)</i> associated with the value and the standard time at the location in which the value was obtained. The <i>SourceTimestamp</i> is defined in [UA Part 4]. <i>VariableTimeZone</i> must not be dependent on Standard/Daylight savings time at the originating location, because this would add ambiguities.
DataTypeVersion	O	String	Only used for <i>Variables</i> of the <i>VariableType DataTypeDictionaryType</i> and <i>DataTypeDescriptionType</i> as described in Clause 5.8.
DictionaryFragment	O	String	Only used for <i>Variables</i> of the <i>VariableType DataTypeDescriptionType</i> as described in Clause 5.8.

The *Variable NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2.

The *Variable NodeClass* also defines a set of *Attributes* that describe the *Variable*'s Runtime value. The *Value Attribute* represents the *Variable* value. The *DataType* and *ArraySize Attributes* provide the capability to describe simple and complex values.

The *AccessLevel Attribute* indicates the accessibility of the *Value* of a *Variable* not taking user access rights into account. If the OPC UA server does not have the ability to get the *AccessLevel* information from the underlying system, it should state that it is read and writable. If a read or write operation is called on the *Variable*, the server should transfer this request and return the corresponding *StatusCodes* if such a request is rejected. *StatusCodes* are defined in [UA Part 4].

The *UserAccessLevel Attribute* indicates the accessibility of the *Value* of a *Variable* taking user access rights into account. If the OPC UA server does not have the ability to get any user access rights related information from the underlying system, it should use the same bit mask as used in the *AccessLevel Attribute*. The *UserAccessLevel Attribute* can restrict the accessibility indicated by the *AccessLevel Attribute*, but not exceed it.

The *MinimumSamplingInterval Attribute* specifies how fast the server can reasonably sample the *value* for changes. The accuracy of this value (the ability of the server to attain "best case" performance) can be greatly affected by system load and other factors.

Clients may read or write *Variable* values, or monitor them for value changes, as specified in [UA Part 4]. [UA Part 8] defines additional rules when using the Services for automation data.

To specify its *ModellingRule*, a *Variable* can use at most one *HasModellingRule Reference* pointing to a *ModellingRule Object*. *ModellingRules* are defined in Clause 5.11.

If the *Variable* is created based on an *InstanceDeclaration* (see Clause 4.5) it must have the same *BrowseName* as its *InstanceDeclaration*.

The other *References* are described separately for *Properties* and *DataVariables* in the following subclauses.

### 5.6.3 Property

*Properties* are used to define the characteristics of *Nodes*. *Properties* are defined using the *Variable NodeClass*, specified in Table 7. However, they restrict their use.

*Properties* are the leaf of any hierarchy, therefore they must not be the *SourceNode* of any *hierarchical References*. This includes the *HasComponent* or *HasProperty Reference*, that is, *Properties* do not contain *Properties* and cannot expose their complex structure. However, they may be the *SourceNode* of any *non-hierarchical References*.

The *HasTypeDefinition Reference* points to the *VariableType* of the *Property*. Since *Properties* are uniquely identified by their *BrowseName*, all *Properties* must point to the *PropertyType* defined in [UA Part 5].

*Properties* must always be defined in the context of another *Node* and must be the *TargetNode* of at least one *HasProperty Reference*. To distinguish them from *DataVariables*, they must not be the *TargetNode* of any *HasComponent Reference*. Thus, a *HasProperty Reference* pointing to a *Variable Node* defines this *Node* as a *Property*.

The *BrowseName* of a *Property* is always unique in the context of a *Node*. It is not permitted for a *Node* to refer to two *Variables* using *HasProperty References* having the same *BrowseName*.

### 5.6.4 DataVariable

*DataVariables* represent the content of an *Object*. *DataVariables* are defined using the *Variable NodeClass*, specified in Table 7.

*DataVariables* identify their *Properties* using *HasProperty References*. Complex *DataVariables* use *HasComponent References* to expose their component *DataVariables*.

The standard *Property NodeVersion* indicates the version of the *DataVariable*. The standard *Property VariableTimeZone* indicates the difference between the *SourceTimestamp* of the value and the standard time at the location in which the value was obtained. The standard *Property DataTypeVersion* is used only for *DataTypeDictionaries* and *DataTypeDescriptions* as defined in Clause 5.8. The Standard *Property DictionaryFragment* is used only for *DataTypeDescriptions* as defined in Clause 5.8. There are no additional standard *Properties* defined for *DataVariables* in this part of this document. Additional parts of this multi-part specification may define additional standard *Properties* for *DataVariables*. [UA Part 8] defines a standard set of *Properties* that can be used for *DataVariables*.

*DataVariables* may use additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *DataVariables*. Standard *ReferenceTypes* are described in Clause 6.

A *DataVariable* is intended to be defined in the context of an *Object*. However, complex *DataVariables* may expose other *DataVariables*, and *ObjectTypes* and complex *VariableTypes* may also contain *DataVariables*. Therefore each *DataVariable* must be the *TargetNode* of at least one *HasComponent Reference* coming from an *Object*, an *ObjectType*, a *DataVariable* or a *VariableType*. *DataVariables* must not be the *TargetNode* of any *HasProperty References*. Therefore, a *HasComponent Reference* pointing to a *Variable Node* identifies it as a *DataVariable*.

The *HasTypeDefinition Reference* points to the *VariableType* used as type definition of the *DataVariable*.

If the *DataVariable* is used as *InstanceDeclaration* (see Clause 4.5) all *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of this *DataVariable*.

## 5.6.5 VariableType NodeClass

*VariableTypes* are used to provide type definitions for *Variables*. *VariableTypes* are defined using the *VariableType NodeClass*, specified in Table 8.

**Table 8 – VariableType NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2
Value	O	Defined by the <i>DataType attribute</i>	The default <i>Value</i> for instances of this type.
DataType	M	NodeId	<i>NodeId</i> of the data type definition for instances of this type.
ArraySize	M	Int32	This <i>Attribute</i> indicates whether the <i>Value Attribute</i> of the <i>VariableType</i> is an array. If it is not an array, the <i>ArraySize</i> is set to -1. If it is an array, the <i>ArraySize</i> specifies the number of elements in the array. The value 0 is used to indicate an array whose size has not been allocated or is not known. For example, if a <i>VariableType</i> is defined by the following C array: Int32 myArray[346]; then this <i>VariableType</i> 's <i>DataType</i> would point to an <i>Int32</i> and the <i>VariableType</i> 's <i>ArraySize</i> has the value 346.
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE     it is an abstract <i>VariableType</i> , i.e. no <i>Variables</i> of this type must exist, only of its subtypes. FALSE    it is not an abstract <i>VariableType</i> , i.e. <i>Variables</i> of this type can exist.
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> are used to identify the <i>Properties</i> of the <i>VariableType</i> . The referenced <i>Nodes</i> may be instantiated by the instances of this type, depending on the <i>ModellingRules</i> defined in Clause 5.11.
HasComponent	0..*		<i>HasComponent References</i> are used for complex <i>VariableTypes</i> to identify their containing <i>DataVariables</i> . Complex <i>VariableTypes</i> can only be used for <i>DataVariables</i> . The referenced <i>Nodes</i> may be instantiated by the instances of this type, depending on the <i>ModellingRules</i> defined in Clause 5.11.
HasSubtype	0..*		<i>HasSubtype References</i> identify <i>VariableTypes</i> that are subtypes of this type. The inverse <i>subtype</i> of <i>Reference</i> identifies the parent type of this type.
GeneratesEvent	0..*		<i>GeneratesEvent References</i> identify the type of <i>Events</i> instances of this type may generate.
References	0..*		<i>VariableTypes</i> may contain other <i>References</i> that can be instantiated by <i>Variables</i> defined by this <i>VariableType</i> . <i>ModellingRules</i> are defined in Clause 5.11.
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes except for the <i>DataType Attribute</i> do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed. Although the relationship of a <i>VariableType</i> to its <i>DataType</i> is not modelled using <i>References</i> , changes to the <i>DataType Attribute</i> of a <i>VariableType</i> lead to an update of the <i>NodeVersion Property</i> .

The *VariableType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. The *VariableType NodeClass* also defines a set of *Attributes* that describe the default or initial value of its instance *Variables*. The *Value Attribute* represents the default value. The *DataType* and *ArraySize Attributes* provide the capability to describe simple and complex values. The *IsAbstract Attribute* defines if the type can be directly instantiated.

The *VariableType NodeClass* uses *HasProperty References* to define the *Properties* and *HasComponent References* to define *DataVariables*. Whether they are instantiated depends on the *ModellingRules* defined in Clause 5.11.

The standard *Property NodeVersion* indicates the version of the *VariableType*. There are no additional standard *Properties* defined for *VariableTypes* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *VariableTypes*. [UA Part 8] defines a standard set of *Properties* that can be used for *VariableTypes*.

*HasSubtype References* are used to subtype *VariableTypes*. *VariableType* subtypes inherit the general semantics from the parent type. The general rules for subtyping are defined in Clause 5.10. It is not required to provide the *HasSubtype Reference* for the supertype, but it is required that the subtype provides the inverse *Reference* to its supertype.

*GeneratesEvent References* identify that *Variables* of the *VariableType* may be the source of an *Event* of the specified *EventType* or one of its subtypes. Servers should make *GeneratesEvent References* bidirectional *References*. However, it is allowed to be unidirectional when the server is not able to expose the inverse direction pointing from the *EventType* to each *VariableType* supporting the *EventType*.

*GeneratesEvent References* are optional, i.e. *Variables* may generate *Events* of an *EventType* that is not exposed by its *VariableType*.

*VariableTypes* may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *VariableTypes*. Standard *ReferenceTypes* are described in Clause 6.

All *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of the *VariableType* (see Clause 4.5).

## 5.7 Method NodeClass

*Methods* define callable functions. *Methods* are invoked using the *Call Service* defined in [UA Part 4]. Method invocations are not represented in the *AddressSpace*. Method invocations always run to completion and always return responses when complete. *Methods* are defined using the *Method NodeClass*, specified in Table 9.

**Table 9 – Method NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2
Executable	M	Boolean	The <i>Executable Attribute</i> indicates if the <i>Method</i> is currently executable (“False” means not executable, “True” means executable). The <i>Executable Attribute</i> does not take any user access rights into account, i.e. although the <i>Method</i> is executable this may be restricted to a certain user / user group.
UserExecutable	M	Boolean	The <i>UserExecutable Attribute</i> indicates if the <i>Method</i> is currently executable (“False” means not executable, “True” means executable). The <i>Executable Attribute</i> does not take any user access rights into account, i.e. although the <i>Method</i> is executable this may be restricted to a certain user / user group.
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> for the <i>Method</i> .
HasModellingRule	0..1		<i>Methods</i> can point to at most one <i>ModellingRule Object</i> using a <i>HasModellingRule Reference</i> (see Clause 5.11 for details on <i>ModellingRules</i> ). If no <i>ModellingRule</i> is specified, the default <i>ModellingRule None</i> is used.
References	0..*		<i>Methods</i> may contain other <i>References</i> .
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.
InputArguments	O	Argument[]	The <i>InputArguments Property</i> is used to specify the arguments that must be used by a client when calling the <i>Method</i> .
OutputArguments	O	Argument[]	The <i>OutputArguments Property</i> specifies the result returned from the <i>Method</i> call.

The *Method NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. The *Method NodeClass* defines no additional *Attributes*.

The *Executable Attribute* indicates whether the *Method* is executable, not taking user access rights into account. If the OPC UA server cannot get the *Executable* information from the underlying system, it should state that it is executable. If a *Method* is called, the server should transfer this request and return the corresponding *StatusCodes* if such a request is rejected. *StatusCodes* are defined in [UA Part 4].

The *UserExecutable Attribute* indicates whether the *Method* is executable, taking user access rights into account. If the OPC UA server cannot get any user rights related information from the underlying system, it should use the same value as used in the *Executable Attribute*. The *UserExecutable Attribute* can be set to “False”, even if the *Executable Attribute* is set to “True”, but it must be set to “False” if the *Executable Attribute* is set to “False”.

*Properties* may be defined for *Methods* using *HasProperty References*. The standard *Properties InputArguments* and *OutputArguments* specify the input arguments and output arguments of the *Method*. Both contain an array of the *Data Type Argument* as specified in Clause 7.6. An empty array a *Property* that is not provided indicates that there are no input arguments or output arguments for the *Method*. The standard *Property NodeVersion* indicates the version of the *Method*. There are no additional standard *Properties* defined for *Methods* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *Methods*.

To specify its *ModellingRule*, a *Method* can use at most one *HasModellingRule* Reference pointing to a *ModellingRule* Object. *ModellingRules* are defined in Clause 5.11.

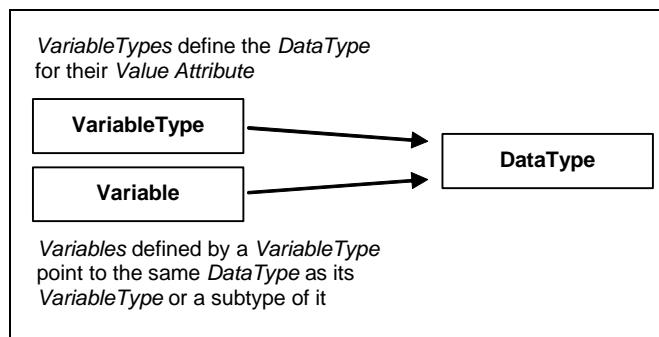
*Methods* may use additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *Methods*. Standard *ReferenceTypes* are described in Clause 6.

A *Method* must always be the *TargetNode* of at least one *HasComponent* Reference. The *SourceNode* of these *HasComponent* References must be an *Object* or an *ObjectType*. If a *Method* is called, the *NodeId* of one of those *Nodes* must be put into the Call Service defined in [UA Part 4] as parameter to detect the context of the *Method* operation.

## 5.8 DataTypes

### 5.8.1 DataType Model

The *DataType* Model is used to define simple and complex data types. Data types are used to describe the structure of the *Value Attribute* of *Variables* and their *VariableTypes*. Therefore each *Variable* and *VariableType* is pointing with its *DataType* *Attribute* to a *Node* of the *DataType* *NodeClass* as shown Figure 9.

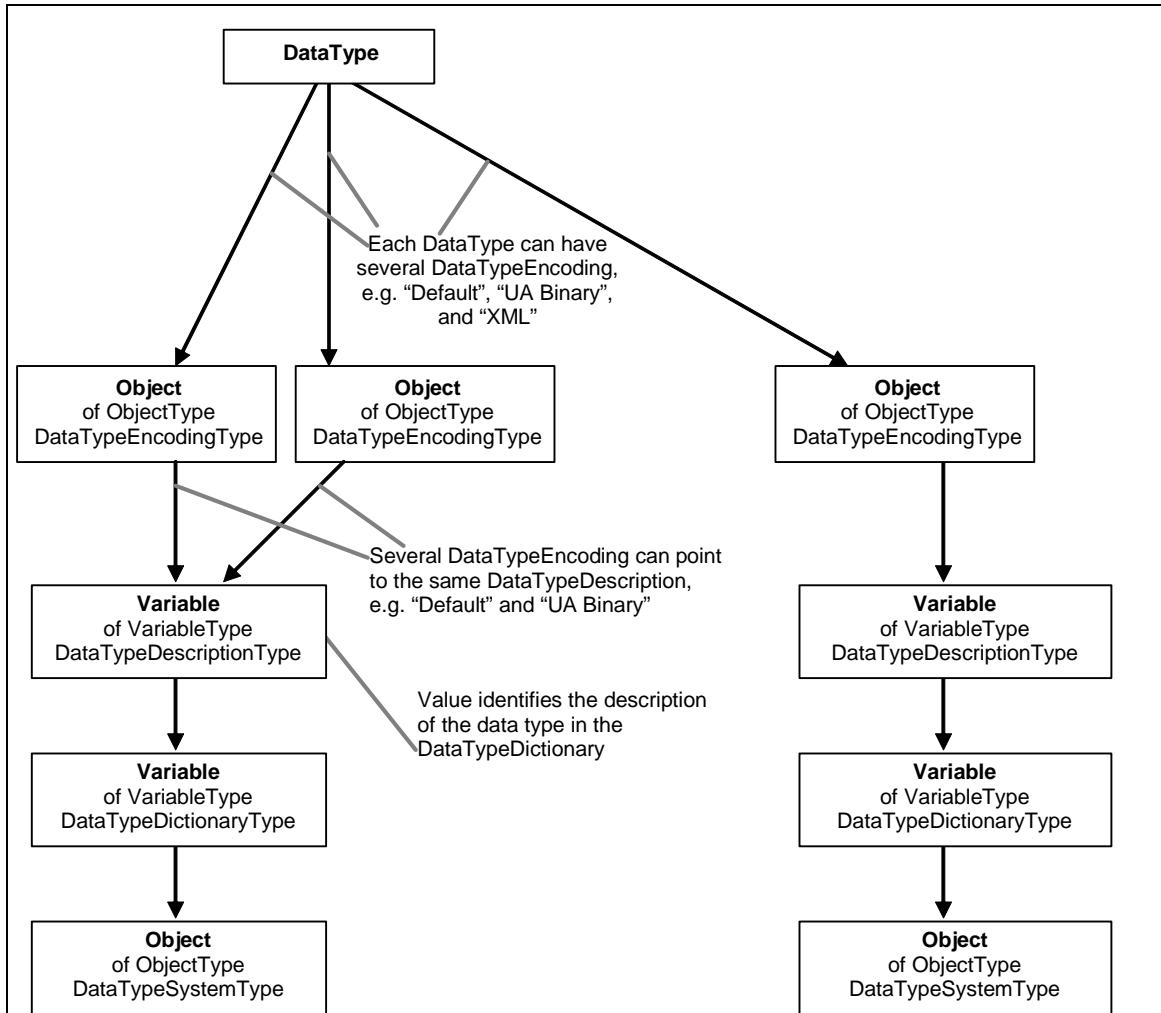


**Figure 9 – Variables, VariableTypes and their DataTypes**

In many cases, the *NodeId* of the *DataType Node* – the *DataTypeId* – will be well-known to clients and servers. Clause 7 defines standard *DataTypes* and [UA Part 5] defines their *DataTypeIds*. In addition, other organizations may define *DataTypes* that are well-known in the industry. Well-known *DataTypeIds* provide for commonality across UA servers and allow clients to interpret values without having to read the type description from the server. Therefore, servers may use well-known *DataTypeIds* without representing the corresponding *DataType Nodes* in their *AddressSpaces*.

In other cases, *DataTypes* and their corresponding *DataTypeIds* may be vendor-defined. Servers should attempt to expose the *DataType Nodes* and the information about the structure of those *DataTypes* for clients to read, although this information may not always be available to the server.

Figure 10 illustrates the *Nodes* used in the *AddressSpace* to describe the structure of a *DataType*. The *DataType* points to an *Object* of type *DataTypeEncodingType*. Each *DataType* can have several *DataTypeEncoding*, for example “Default”, “UA Binary” and “XML” encoding. Services in [UA Part 4] allow clients to request an encoding or choosing the “Default” encoding. Each *DataTypeEncoding* is used by exactly one *DataType*, that is, it is not permitted for two *DataTypes* to point to the same *DataTypeEncoding*. The *DataTypeEncoding Object* points to exactly one *Variable* of type *DataTypeDescriptionType*. The *DataTypeDescription Variable* belongs to a *DataTypeDictionary Variable*.

**Figure 10 – DataType Model**

Since the *NodeId* of the *DataTypeEncoding* will be used in some Mappings to identify the *DataType* and its encoding as defined in [UA Part 6], those *NodIds* may also be well-known for well-known *DataTypeIds*.

The *DataTypeDictionary* describes a set of *DataTypes* in sufficient detail to allow clients to parse/interpret *Variable Values* that they receive and to construct *Values* that they send. The *DataTypeDictionary* is represented as a *Variable* of type *DataTypeDictionaryType* in the *AddressSpace*, the description about the *DataTypes* is contained in its *Value Attribute*. All containing *DataTypes* exposed in the *AddressSpace* are represented as *Variables* of type *DataTypeDescriptionType*. The *Value* of one of these *Variables* identifies the description of a *DataType* in the *Value Attribute* of the *DataTypeDictionary*.

The *DataType* of a *DataTypeDictionary Variable* is always a *ByteString*. The format and conventions for defining *DataTypes* in this *ByteString* are defined by *DataTypeSystems*. *DataTypeSystems* are identified by *NodeIds*. They are represented in the *AddressSpace* as *Objects* of the *ObjectType* *DataTypeSystemType*. Each *Variable* representing a *DataTypeDictionary* references a *DataTypeSystem Object* to identify their *DataTypeSystem*.

A client must recognise the *DataTypeSystem* to parse any of the type description information. UA clients that do not recognise a *DataTypeSystem* will not be able to interpret its type descriptions, and consequently, the values described by them. In these cases; clients interpret these values as opaque *ByteStrings*.

OPC Binary, W3C XML Schema and Electronic Device Description Language (EDDL) are examples of *Data Type Systems*. The OPC Binary *Data Type System* is defined in Appendix C. OPC Binary uses XML to describe binary data values. W3C XML Schema is specified in [XML Schema Part 1] and [XML Schema Part 2], EDDL in [EDDL].

### 5.8.2 DataType NodeClass

The *DataType NodeClass* describes the syntax of a *Variable Value*. The *Data Types* may be simple or complex, depending on the *DataTypeSystem*. *Data Types* are defined using the *DataType NodeClass*, specified in Table 10.

**Table 10 – Data Type NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See Clause 5.2
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> for the <i>data type</i> .
HasSubtype	0..*		<i>HasSubtype References</i> may be used to span a <i>data type hierarchy</i> .
HasEncoding	0..*		<i>HasEncoding References</i> identify the encodings of the <i>DataType</i> represented as <i>Objects</i> of type <i>DataTypeEncodingType</i> . Each concrete <i>DataType</i> must point to at least one <i>DataTypeEncoding Object</i> with the <i>BrowseName</i> “Default Binary” or “Default XML” having the <i>NamespaceIndex</i> 0. The <i>BrowseName</i> of the <i>DataTypeEncoding Objects</i> must be unique in the context of a <i>DataType</i> , i.e. a <i>DataType</i> must not point to two <i>DataTypeEncodings</i> having the same <i>BrowseName</i> . An abstract <i>DataType</i> does not point to a <i>DataTypeEncoding Object</i> . This is the way to identify if a <i>DataType</i> is abstract.
<b>Standard Properties</b>			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.

The *DataType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 5.2. The *DataType NodeClass* defines no additional *Attributes*.

*HasProperty References* are used to identify the *Properties* of a *DataType*.

The standard *Property NodeVersion* is used to indicate the version of the *DataType*. This Version is not affected by the *DataTypeVersion Property* of *DataTypeDictionaries* and *DataTypeDescriptions*.

There are no additional standard *Properties* defined for *Data Types* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *Data Types*.

*HasSubtype References* may be used to expose a *data type hierarchy* in the *AddressSpace*. This hierarchy must reflect the hierarchy specified in the *DataTypeDictionary*. The semantic of subtyping depends on the *DataTypeSystem*. Servers need not provide *HasSubtype References*, even if their *Data Types* span a type hierarchy. Clients should not make any assumptions about any other semantic with that information than provided by the *DataTypeDictionary*. For example, it might not be possible to cast a value of one *data type* to its base *data type*.

*HasEncoding References* point from the *DataType* to its *DataTypeEncodings*. Following such a *Reference*, the client can browse to the *DataTypeDictionary* describing the structure of the *DataType* for the used encoding. Each concrete *DataType* can point to many *DataTypeEncodings*, but each *DataTypeEncoding* must belong to one *DataType*, that is, it is not permitted for two *DataType Nodes* to point to the same *DataTypeEncoding Object* using *HasEncoding References*.

An abstract *DataType* is not the *SourceNode* of a *HasEncoding* Reference. The *DataTypeEncoding* of an abstract *DataType* is provided by its concrete subtypes.

*DataType* Nodes must not be the *SourceNode* of other types of References. However, they may be the *TargetNode* of other References.

### 5.8.3 **DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem**

A *DataTypeDictionary* is an entity that contains a set of type descriptions, such as an XML schema or an EDDL Device Description. *DataTypeDictionaries* are defined as *Variables* of the *VariableType* *DataTypeDictionaryType*.

A *DataTypeSystem* specifies the format and conventions for defining *DataTypes* in *DataTypeDictionaries*. *DataTypeSystems* are defined as *Objects* of the *ObjectType* *DataTypeSystemType*.

The *ReferenceType* used to relate *Objects* of the *ObjectType* *DataTypeSystemType* to *Variables* of the *VariableType* *DataTypeDictionaryType* is the *HasComponent* *ReferenceType*. Thus, the *Variable* is always the *TargetNode* of a *HasComponent* Reference – a requirement for *Variables*. However, for *DataTypeDictionaries* the server must always provide the inverse reference, since it is necessary to know the *DataTypeSystem* when processing the *DataTypeDictionary*.

An example of a *DataTypeDictionary* is an XML document containing an XML schema. In this case, the *DataTypeSystem* is the W3C XML Schema and the top level element declarations in the schema document are the data type descriptions. Each of these descriptions is defined in different versions of an XML schema using the same XML target namespace. This target namespace is used as the namespace component of the *DataTypeId* in the server's *AddressSpace*. Since the same target namespace can be used in other XML schemas, clients must be aware that two *DataTypeIds* with the same namespace are not necessarily defined in the same *DataTypeDictionary*.

Changes may be a result of a change to a type description, but it is more likely that dictionary changes are a result of the addition or deletion of type descriptions. This includes changes made while the server is offline so that the new version is available when the server restarts. Clients may subscribe to the *DataTypeVersion* Property to determine if the *DataTypeDictionary* has changed since it was last read.

The server may – but is not required to – make the *DataTypeDictionary* contents available to clients through the *Value Attribute*. Clients should assume that *DataTypeDictionary* contents are relatively large and that they will encounter performance problems if they automatically read the *DataTypeDictionary* contents each time they encounter an instance of a specific *DataType*. The client should use the *DataTypeVersion* Property to determine whether the locally cached copy is still valid. If the client detects a change to the *DataTypeVersion*, then it must re-read the *DataTypeDictionary*. This implies that the *DataTypeVersion* must be updated by a server even after restart since clients may persistently store the locally cached copy.

The *Value Attribute* of the *DataTypeDictionary* containing the type descriptions is a *ByteString* whose formatting is defined by the *DataTypeSystem*. For the “XML Schema” *DataTypeSystem*, the *ByteString* contains a valid XML Schema document. For the “OPC Binary” *DataTypeSystem*, the *ByteString* contains a string that is a valid XML document. The server must ensure that any change to the contents of the *ByteString* is matched with a corresponding change to the *DataTypeVersion* Property. In other words, the client may safely use a cached copy of the *DataTypeDictionary*, as long as the *DataTypeVersion* remains the same.

*DataTypeDictionaries* are complex *Variables* which expose their *DataTypeDescriptions* as *Variables* using *HasComponent* References. A *DataTypeDescription* provides the information necessary to find the formal description of a *DataType* within the *DataTypeDictionary*. The *Value* of a *DataTypeDescription* depends on the *DataTypeSystem* of the *DataTypeDictionary*. When using “OPC Binary” dictionaries the *Value* must be the name of the *TypeDescription*. When using “XML

Schema" dictionaries the Value must be an XPath expression [XPATH] which points to an XML element in the schema document.

Like *DataTypeDictionaries* each *DataTypeDescription* provides the standard *Property DataTypeVersion* indicating whether the type description of the *DataType* has changed. Changes to the *DataTypeVersion* may impact the operation of *Subscriptions*. If the *DataTypeVersion* changes for a *Variable* that is being monitored for a *Subscription* and that uses this *DataTypeDescription*, then the next data change *Notification* sent for the *Variable* will contain a status that indicates the change in the *DataTypeDescription*.

*DataTypeEncoding Objects* of the *DataTypes* reference their *DataTypeDescriptions* of the *DataTypeDictionaries* using *HasDescription* References. However, servers are not required to provide the inverse References that relate the *DataTypeDescriptions* back to the *DataTypeEncoding Objects*. If a *DataType Node* is exposed in the *AddressSpace*, it must provide its *DataTypeEncodings* and if a *DataTypeDictionary* is exposed, it should expose all its *DataTypeDescriptions*. Both of these References must be bi-directional.

The *VariableTypes* *DataTypeDictionaryType* and *DataTypeDescriptionType* and the *ObjectTypes* *DataTypeSystemType* and *DataTypeEncodingType* are formally defined in [UA Part 5].

Figure 11 gives an example how *DataTypes* are modelled in the *AddressSpace*.

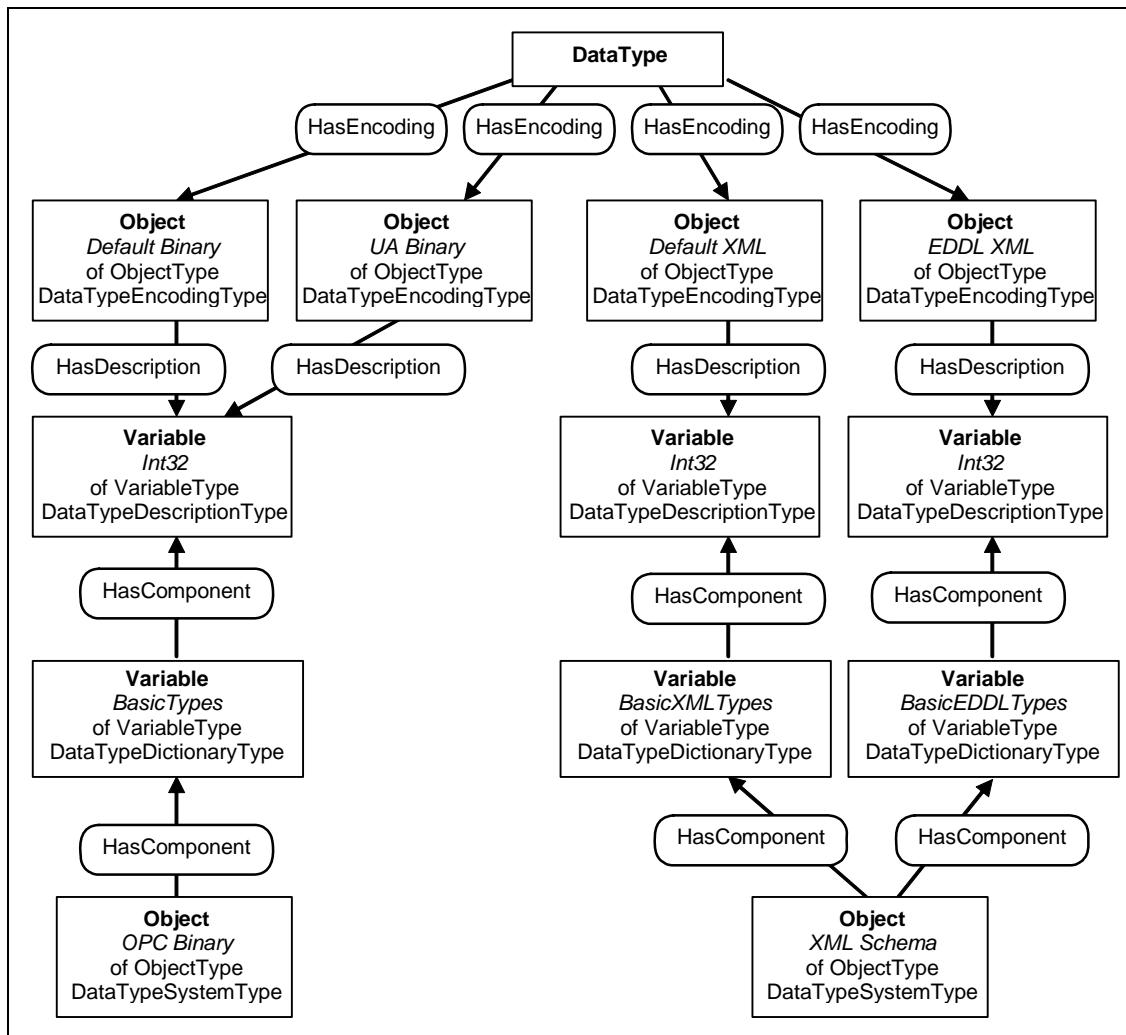


Figure 11 – Example of DataType Modelling

In some scenarios an OPC UA server may have resource limitations which make it impractical to expose large *DataTypeDictionaries*. In these scenarios the server may be able to provide access to descriptions for individual DataTypes even if the entire dictionary cannot be read. For this reason, UA defines a standard *Property* for the *DataTypeDescription* called *DictionaryFragment* (see Clause 5.6.2). This *Property* is a ByteString that contains a subset of the *DataTypeDictionary* which describes the format of the *DataType* associated with the *DataTypeDescription*. Thus the server splits the large *DataTypeDictionary* into several small parts clients can access without affecting the overall system performance.

However, servers should provide the whole *DataTypeDictionary* at once and if this is possible. Clients can typically act more effective reading the whole *DataTypeDictionary* at once instead of reading several parts and building their own *DataTypeDictionary* over a period of time.

All *DataTypeDictionaries* must be uniquely identified by a URI that is usually assigned by the organization that created the dictionary and may be used by multiple servers. For this reason the canonical *NodeId* for each *DataTypeDictionary* must be this URI. As result, clients may use the URI to read the dictionary from its own configuration or even fetch it from a website (if the URI is a URL). Clients must use the URI to locate the dictionary if the server does not provide either the *Value* of a *DataTypeDictionary* or *DictionaryFragment Properties*.

## 5.9 Summary of Attributes of the NodeClasses

Table 11 summarises all *Attributes* defined in this document and points out which *NodeClasses* use them either optional (O) or mandatory (M).

**Table 11 – Overview about Attributes**

Attribute	Variable	Variable Type	Object	Object Type	Reference Type	DataType	Method	View
AccessLevel	M							
ArraySize	M	M						
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
Description	O	O	O	O	O	O	O	O
DisplayName	M	M	M	M	M	M	M	M
EventNotifier			M					M
Executable							M	
InverseName					O			
IsAbstract		M		M	M			
MinimumSamplingInterval	O							
NodeClass	M	M	M	M	M	M	M	M
NodeId	M	M	M	M	M	M	M	M
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
Value	M	O						

## 5.10 Subtyping of ObjectTypes and VariableTypes

The *HasSubtype* standard *ReferenceType* defines subtypes of types. Subtyping can only occur between *Nodes* of the same *NodeClass*. Subtypes do not inherit the parent type's *NodeId* or the parent type's *BrowseName*. Rules for subtyping *ReferenceTypes* are described in Clause 5.3.3.3. There is no common definition for subtyping *DataTypes*, as described in Clause 5.8.2. This Clause specifies subtyping rules for *ObjectTypes* and *VariableTypes*.

The rules for single inheritance from the parent type are:

- a) Subtypes inherit the fully-inherited parent type's *Attribute* values, except for the *NodeId* and the *BrowseName*. Inherited *Attribute* values may be overridden by the subtype unless restricted by the *NodeClass* definition. Optional *Attributes*, not provided by the parent type, may be added to the subtype. In this context, fully-inherited means that inheritances for parent types are done first, recursively down from the top-level parent.
- b) Subtypes inherit the fully-inherited parent type's *References* to *Nodes* used for instantiation. This depends on the *ModellingRule* of the referenced *Node*. *ModellingRules* are defined in Clause 5.11. In general, inheritance of *References* means that the same *References* are defined for the subtypes. The *TargetNode* for each of these *References* may be the *TargetNode* of the parent type's *Reference*, or another *Node* of the same *NodeClass*. If the *Node* "A" referenced in the parent type has a type definition "Type\_A", the *Node* "B" referenced in the subtype must have the same type definition "Type\_A" or its type definition must be derived from "Type\_A".
- c) Changing the values of the *Attributes* of a supertype is not reflected in its subtypes. Whether changing *References* is reflected in the subtypes is server-dependent.

## 5.11 Instantiation of ObjectTypes and VariableTypes

### 5.11.1 General

OPC UA requires servers to provide type definitions for *Objects* and *Variables*. If an *Object* or *Variable* is created based on a type, the following rules apply:

- a) The fully-inherited type, as defined in Clause 5.10, is used for the instantiation.
- b) Instances inherit the initial values for the *Attributes* that they have in common with the *Node* from which they are instantiated, with the exceptions of the *NodeClass*, *NodeId* and *BrowseName*.
- c) *Nodes* that are referenced from the *TypeDefinitionNode* are instantiated depending on the *ModellingRule* of the *TargetNode*. The *TargetNode ModellingRule* is specified by using the *HasModellingRule Reference* pointing to a *ModellingRule*. Each *Node* may have at most one *ModellingRule*. If no *ModellingRule* is provided, the default *ModellingRule None* is used. If and how the *Node* is instantiated depends on the *ModellingRule*. Clause 5.11.3 defines standard *ModellingRules*.

### 5.11.2 Ownership due to ModellingRules

OPC UA does not provide a general concept of ownership for its *Nodes*. *DataType Nodes*, for example, may exist in the *AddressSpace* without being referenced and owned by other *Nodes*. The same is true for *ObjectTypes* and *VariableTypes*. It is vendor-specific whether a deletion of such a type leads to the deletion of its subtypes or not.

However, OPC UA provides the concept of ownership for *Variables* and *Methods*, since they must always be part of at least one other *Node*. It also provides the concept of ownership for *Objects*, but the ownership of *Objects* is optional, that is, not every *Object* must be owned.

For each *Method*, *Variable* and *Object* exactly one *ModellingRule* is defined, either using the *HasModellingRule Reference* or using the default *ModellingRule*. Clause 5.11.3 defines standard *ModellingRules*. However, *ModellingRules* are extensible and therefore vendors may define *ModellingRules* having an ownership semantic for other *NodeClasses*, too. Each *ModellingRule* applicable for *Methods* or *Variables* must specify their ownership semantic since they must be referenced by at least one *Aggregates Reference*.

Independent of any ownership semantic and assigned *ModellingRule*, any *Node* may be deleted as result of the deletion of another *Node* or other Service invocations. OPC UA does not forbid this, but any rules behind that are server-specific. The same apply for any changes in the *AddressSpace*, e.g. a *Node* may be added when a client connects to the system.

### 5.11.3 Standard ModellingRules

#### 5.11.3.1 General

OPC UA defines standard *ModellingRules*. The following subclauses define standard *ModellingRules* in this document. Other parts of this multi-part specification may define additional *ModellingRules*. *ModellingRules* are an extendable concept in OPC UA; therefore vendors may define their own *ModellingRules*. *ModellingRules* are represented in the *AddressSpace* as *Objects* of the *ObjectType ModellingRuleType* or one of its subtypes. [UA Part 5] specifies the representation of the *ModellingRule Objects* and its type in the *AddressSpace*.

#### 5.11.3.2 None

The standard *ModellingRule None* indicates that the *Node* marked with this rule is neither considered for instantiation of a type nor created due to *InstanceDeclarations* of a type.

If a *Node* referenced by a *TypeDefinitionNode* is marked with the *ModellingRule None* it indicates that this *Node* only belongs to the *TypeDefinitionNode* and not to the instances. For example, an *ObjectType Node* may contain a *Property* that describes scenarios where the type could be used. This *Property* would not be considered when creating instances of the type. This is also true for subtyping, that is, subtypes of the type definition would not inherit the referenced *Node*.

If a *Node* referenced by an instance “A” is marked with the *ModellingRule None*, it indicates that this *Node* was not created due to the *InstanceDeclarations* of the *TypeDefinitionNode* of “A”. For example, a *DataVariable* representing a heat sensor instantiated from a heat sensor *VariableType* may have an additional *Property* not defined by its type, containing the latest maintenance report of the *DataVariable*. However, the *Property* is based on the *.PropertyType*.

If the *None ModellingRule* is used for a *Method* or *Variable*, it implies an ownership for those *Nodes*. Any *Method* or *Variable* having a *None ModellingRule* or no *ModellingRule* assigned to it must be the *TargetNode* of exactly one *HasComponent* or *HasProperty Reference*. If the *SourceNode* of this *Reference* is deleted, the *TargetNode* must also be deleted.

There is no ownership semantic assigned for other *NodeClasses* having no *ModellingRule* or *None* assigned to them.

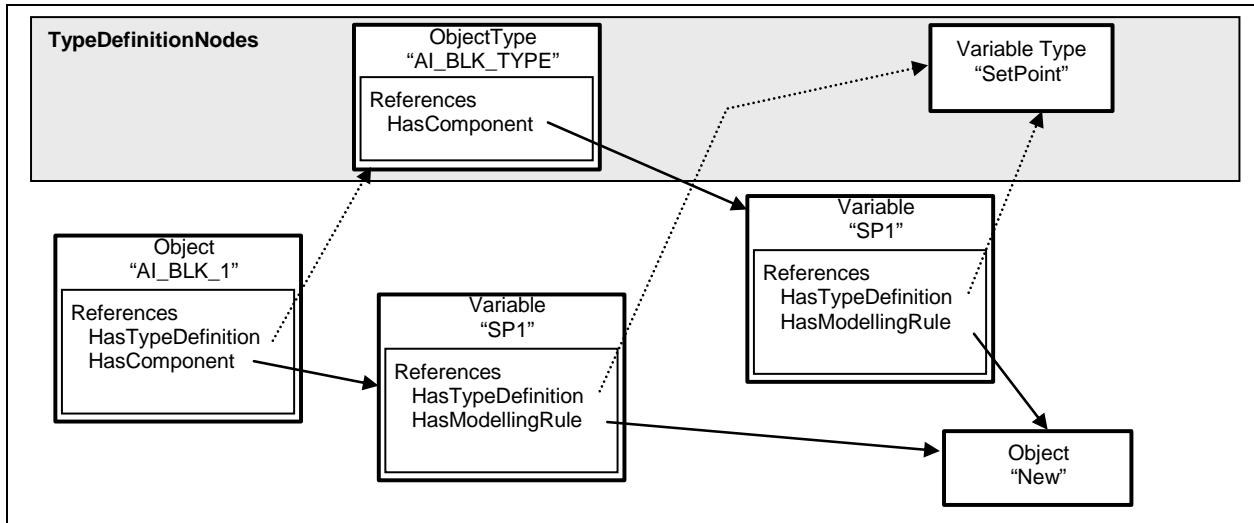
Since *None* is the default *ModellingRule*, omitting the *ModellingRule* has the same semantic.

Clause 5.11.3.5 gives another example how this standard *ModellingRule* is used.

#### 5.11.3.3 New

The standard *ModellingRule New* indicates that the *Node* referenced by a *TypeDefinitionNode* is newly-created for each instance. This *ModellingRule* applies only to *Methods*, *Objects*, and *Variables*, and affects only *HasProperty* and *HasComponent References* or their subtypes.

If a *Node* contained by a *TypeDefinitionNode* is marked with the *ModellingRule New* it indicates that a copy of this *Node* will be newly-created for each instance of the type. For example, the *TypeDefinitionNode* of a functional block “AI\_BLK\_TYPE” will have a setpoint “SP1”. An instance of this type “AI\_BLK\_1” will have a newly-created setpoint “SP1”, created as a copy of the “SP1” of the type. Figure 12 illustrates the example. “AI\_BLK\_1” has no *ModellingRule* that is similar to the *None ModellingRule* since it was directly created based on a *TypeDefinitionNode* and not based on an *InstanceDeclaration* of a *TypeDefinitionNode*.



**Figure 12 – Use of the Standard ModellingRule New**

The following rules for creating a copy apply:

1. A new *Node* will be created having the same *NodeClass* and a different *NodeId*. All other *Attributes* have initially the same values; the *BrowseName* must always be the same.
2. All referenced *Nodes* will be instantiated for the copy depending on their *ModellingRules*.

If a *Node* contained by an instance is marked with the *ModellingRule New*, one of the following cases apply: It indicates that the referenced *Node* was either created due to the *ModellingRule* or that it indirectly belongs to a *TypeDefinitionNode*.

When a *TypeDefinitionNode* referencing a *Node* marked as *New* is subtyped, its subtype either references the same *Node* or another *Node* of the same *NodeClass*. If another *Node* is referenced, either the same type definition or a subtype of it must be used. The *BrowseName* must be the same.

There are two different cases affecting how the ownership of *Nodes* having the *New ModellingRule* applies. The *Node* is either referenced by a *HasComponent* or *HasProperty* of a *TypeDefinitionNode*, or it is not.

In the first case, it may be the *TargetNode* of either one or more *HasComponent References* or one or more *HasProperty References*. The *SourceNodes* of these *References* must all be *TypeDefinitionNodes*, of the same type hierarchy. The server must delete the *Node* if the last *Node* referencing it with a *HasComponent* or *HasProperty Reference* is deleted.

In the second case, it must be the *TargetNode* of exactly one *HasComponent* or one *HasProperty Reference*. If the *SourceNode* of this *Reference* is deleted, the *TargetNode* must also be deleted.

#### 5.11.3.4 Shared

The standard *ModellingRule Shared* specifies that this *Node* can be shared by many other *Nodes*, that is, having many *Nodes* pointing with an *Aggregates Reference* to it.

In general, every *Node* can be marked as *Shared*, that is, many *Aggregates References* can have this *Node* as *TargetNode*. There is no ownership defined for shared *Nodes*; a server may or may not delete a *Node* if a shared *Node* has no *Aggregates Reference* pointing to it.

If the *Node* is a *Variable* or *Method*, the server must delete the *Node* if the last *Node* referencing it with a *HasComponent* or *HasProperty* Reference is deleted, since *Variables* and *Methods* may never stand alone.

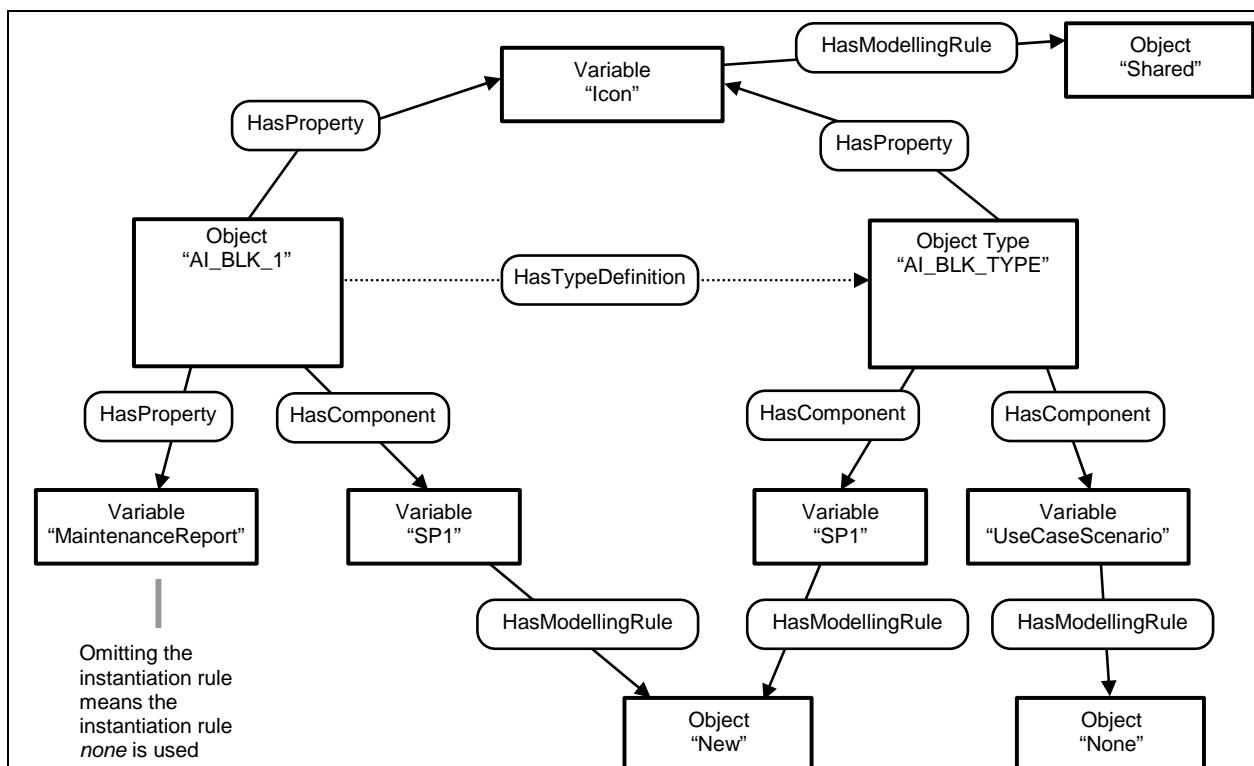
If a *TypeDefinitionNode* references a *Node* with the *ModellingRule Shared*, each instance of this type must have the same *Reference* referencing the same *Node*. For example, an *ObjectType* may have a *HasProperty* Reference to a *Property* pointing to an icon. Each instance of the *ObjectType* would also have a *HasProperty* Reference to the same *Property*. However, it is not specified if deleting the *Reference* from the *ObjectType* to the shared *Node* will lead to the same behaviour on the instances.

When a type definition is subtyped, its subtype either references the same *Shared Node* or another *Shared Node* of the same *NodeClass*. If another *Node* is referenced, either the same type definition or a subtype of it must be used.

#### 5.11.3.5 Examples using standard ModellingRules

In Figure 13 an example using the standard *ModellingRules* is shown. The *ObjectType* "AI\_BLK\_TYPE" contains three *Variables*. "SP1" represents a setpoint and has the *ModellingRule New*, since it must be instantiated for each instance. The "UseCaseScenario" describes how to use the type; therefore it is not needed for the instances and has the *ModellingRule None*. The "Icon" is used to represent each instance of the type in a graphical display, thus it has the *ModellingRule Shared* because all instances can use the same instance.

An instance of "AI\_BLK\_TYPE" is also shown in Figure 13, called "AI\_BLK\_1". Due to the *ModellingRules* it has a *Reference* to the shared "Icon" and a newly created "SP1". The "UseCaseScenario" was not considered for instantiation purposes. A new *Variable*, "MaintenanceReport" was added to the instance. Since this was not the result of the instantiation of "AI\_BLK\_1", but directly instantiated due to a *VariableType* not shown in the figure, it has no *ModellingRule* – having the same semantic as the *None ModellingRule*. "AI\_BLK\_1" itself has no *ModellingRule*, since it was directly created based on a *TypeDefinitionNode* and not based on an *InstanceDeclaration* of a *TypeDefinitionNode*.

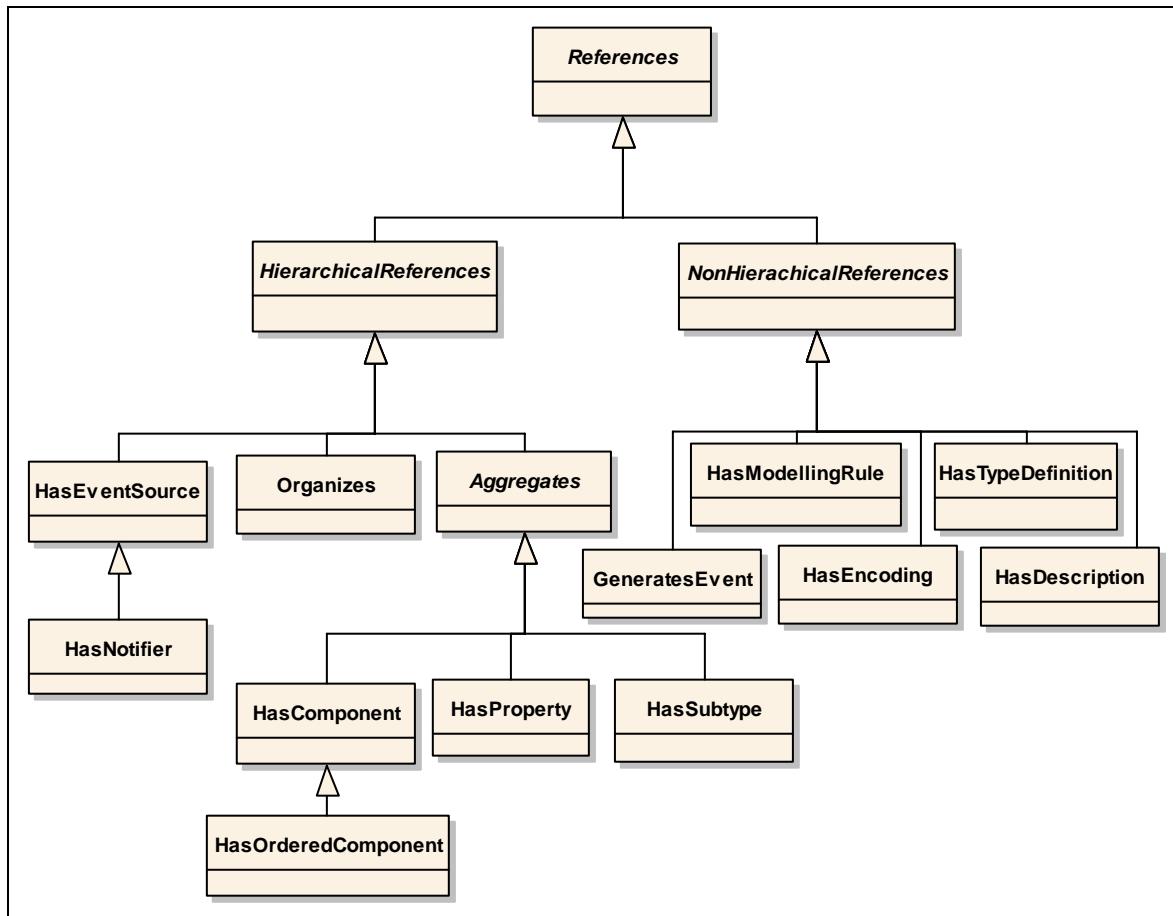


**Figure 13 – Example of usage of Standard ModellingRules**

## 6 Standard ReferenceTypes

### 6.1 General

OPC UA defines standard *ReferenceTypes* as an inherent part of the OPC UA Address Space Model. Figure 14 informally describes the hierarchy of these standard *ReferenceTypes*. Other parts of this multi-part specification may specify additional *ReferenceTypes*. The following subclauses define the standard *ReferenceTypes*. [UA Part 5] defines their representation in the *AddressSpace*.



**Figure 14 – Standard ReferenceType Hierarchy**

### 6.2 References ReferenceType

The *References ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

There is no semantic associated with this *ReferenceType*. This is the base type of all *ReferenceTypes*. All *ReferenceTypes* must be a subtype of this base *ReferenceType* – either direct or indirect. The main purpose of this *ReferenceType* is allowing simple filter and queries in the corresponding Services of [UA Part 4].

There are no constraints defined for this abstract *ReferenceType*.

### 6.3 HierarchicalReferences ReferenceType

The *HierarchicalReferences ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

The semantic of *HierarchicalReferences* is to denote that *References* of *HierarchicalReferences* span a hierarchy. It means that it may be useful to present *Nodes* related with *References* of this

type in a hierarchy-like way. It does not forbid loops, that is, starting from *Node “A”* and following *HierarchicalReferences* may lead to browse to *Node “A”*, again.

It is not permitted to have a *Property* as *SourceNode* of a *Reference* of any subtype of this abstract *ReferenceType*.

#### 6.4 NonHierarchicalReferences ReferenceType

The *NonHierarchicalReferences ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

The semantic of *NonHierarchicalReferences* is to denote that its subtypes do not span a hierarchy and should not be followed when trying to present a hierarchy. To distinguish hierarchical and non-hierarchical *References*, all concrete *ReferenceTypes* must inherit from either *hierarchical References* or *non-hierarchical References*, either direct or indirect.

There are no constraints defined for this abstract *ReferenceType*.

#### 6.5 Aggregates ReferenceType

The *Aggregates ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used. It is a subtype of *HierarchicalReferences*.

The semantic is to indicate that *References* of this type span a non-looping hierarchy.

Starting from *Node “A”* and only following *References* of the subtypes of the *Aggregates ReferenceType* must never be able to return to “A”. But it is allowed that following the *References* there may be more than one path leading to another *Node “B”*.

#### 6.6 HasComponent ReferenceType

The *HasComponent ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType*.

The semantic is a part-of relationship. The *TargetNode* of a *Reference* of the *HasComponent ReferenceType* is a part of the *SourceNode*. This *ReferenceType* is used to relate *Objects* or *ObjectTypes* with their containing *Objects*, *DataVariables*, and *Methods* as well as complex *Variables* or *VariableTypes* with their *DataVariables*.

Like all other *ReferenceTypes*, this *ReferenceType* does not specify anything about the ownership of the parts, although it represents a part-of relationship semantic. That is, it is not specified if the *TargetNode* of a *Reference* of the *HasComponent ReferenceType* is deleted when the *SourceNode* is deleted. *ModellingRules*, as defined in Clause 5.11, can be used for this purpose.

The *TargetNode* of this *ReferenceType* must be a *Variable*, an *Object* or a *Method*.

If the *TargetNode* is a *Variable*, the *SourceNode* must be an *Object*, an *ObjectType*, a *DataVariable* or a *VariableType*. By using the *HasComponent Reference*, the *Variable* is defined as *DataVariable*.

If the *TargetNode* is an *Object* or a *Method*, the *SourceNode* must be an *Object* or *ObjectType*.

#### 6.7 HasProperty ReferenceType

The *HasProperty ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType*.

The semantic is to identify the *Properties* of a *Node*. *Properties* are described in Clause 4.4.2.

The *SourceNode* of this *ReferenceType* can be of any *NodeClass*. The *TargetNode* must be a *Variable*. By using the *HasProperty Reference*, the *Variable* is defined as *Property*. Since *Properties* must not have *Properties*, a *Property* must never be the *SourceNode* of a *HasProperty Reference*.

## 6.8 HasOrderedComponent ReferenceType

The *HasOrderedComponent ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *HasComponent ReferenceType*.

The semantic of the *HasOrderedComponent ReferenceType* – besides the semantic of the *HasComponent ReferenceType* – is that when browsing from a *Node* and following *References* of this type or its subtype all *References* are returned in the *Browse Service* defined in [UA Part 4] in a well-defined order. The order is server-specific, but the client can assume that the server always returns them in the same order.

There are no additional constraints defined for this abstract *ReferenceType*.

## 6.9 HasSubtype ReferenceType

The *HasSubtype ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType*.

The semantic of this *ReferenceType* is to express a subtype relationship of types. It is used to span the *ReferenceType* hierarchy, which semantic is specified in Clause 5.3.3.3; a *DataType* hierarchy as specified in Clause 5.8.2, as well as other subtype hierarchies as specified in Clause 5.10.

The *SourceNode* of *References* of this type must be an *ObjectType*, a *VariableType*, a *DataType* or a *ReferenceType* and the *TargetNode* must be of the same *NodeClass* as the *SourceNode*. Each *ReferenceType* must be the *TargetNode* of at most one *Reference* of type *HasSubtype*.

## 6.10 Organizes ReferenceType

The *Organizes ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HierarchicalReferences*.

The semantic of this *ReferenceType* is to organise *Nodes* in the *AddressSpace*. It can be used to span multiple hierarchies independent of any hierarchy created with the non-looping *Aggregates References*.

The *SourceNode* of *References* of this type must be an *Object*; it should be an *Object* of the *ObjectType FolderType* or one of its subtypes (see Clause 5.5.3).

The *TargetNode* of this *ReferenceType* can be of any *NodeClass*.

## 6.11 HasModellingRule ReferenceType

The *HasModellingRule ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to bind the *ModellingRule* to an *Object*, *Variable* or *Method*. The *ModellingRule* mechanisms are described in Clause 5.11.

The *SourceNode* of this *ReferenceType* must be an *Object*, *Variable* or *Method*. The *TargetNode* must be an *Object* of the *ObjectType "ModellingRule"* or one of its subtypes.

Each *Node* may be the *SourceNode* of at most one *HasModellingRule Reference*.

## 6.12 HasTypeDefinition ReferenceType

The *HasTypeDefinition ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to bind an *Object* or *Variable* to its *ObjectType* or *VariableType*, respectively. The relationships between types and instances are described in Clause 4.5.

The *SourceNode* of this *ReferenceType* must be an *Object* or *Variable*. If the *SourceNode* is an *Object*, the *TargetNode* must be an *ObjectType*; if the *SourceNode* is a *Variable*, the *TargetNode* must be a *VariableType*.

Each *Variable* and each *Object* must be the *SourceNode* of exactly one *HasTypeDefinition Reference*.

## 6.13 HasEncoding ReferenceType

The *HasEncoding ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to reference *DataTypeEncodings* of a *DataType*.

The *SourceNode* of *References* of this type must be a *DataType*.

The *TargetNode* of this *ReferenceType* must be an *Object* of the *ObjectType* *DataTypeEncodingType* or one of its subtypes (see Clause 5.8.3).

## 6.14 HasDescription ReferenceType

The *HasDescription ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to reference the *DataTypeDescription* of a *DataTypeEncoding*.

The *SourceNode* of *References* of this type must be an *Object* of the *ObjectType* *DataTypeEncodingType* or one of its subtypes.

The *TargetNode* of this *ReferenceType* must be an *Object* of the *ObjectType* *DataTypeDescriptionType* or one of its subtypes (see Clause 5.8.3).

## 6.15 GeneratesEvent

The *GeneratesEvent ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to identify the types of *Events* instances of *ObjectTypes* or *VariableTypes* may generate.

The *SourceNode* of *References* of this type must be an *ObjectType* or a *VariableType*.

The *TargetNode* of this *ReferenceType* must be an *ObjectType* representing *EventTypes*, i.e. the *BaseEventType* or one of its subtypes.

## 6.16 HasEventSource

The *HasEventSource ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HierarchicalReferences*.

The semantic of this *ReferenceType* is to relate event sources in a hierarchical, non-looping organization. This *ReferenceType* and any subtypes are intended to be used for discovery of *Event* generation in a server. They are not required to be present for a server to generate *Event* from its source to its notifying *Nodes*. In particular, the root notifier of a server – the *Server Object* defined in [UA Part 5] – is always capable of supplying all *Events* from a server and as such has implied *HasEventSource References* to every event source in a server.

The *SourceNode* of this *ReferenceType* must be an *Object* that is a source of event subscriptions. A source of event subscriptions is an *Object* that has its “SubscribeToEvents” bit set within the *EventNotifier Attribute*.

The *TargetNode* of this *ReferenceType* can be a *Node* of any *NodeClass* that can generate event notifications via a subscription to the reference source.

Starting from *Node “A”* and only following *References* of the *HasEventSource ReferenceType* or its subtypes must never be able to return to “A”. But it is permitted that, following the *References*, there may be more than one path leading to another *Node “B”*.

## 6.17 HasNotifier

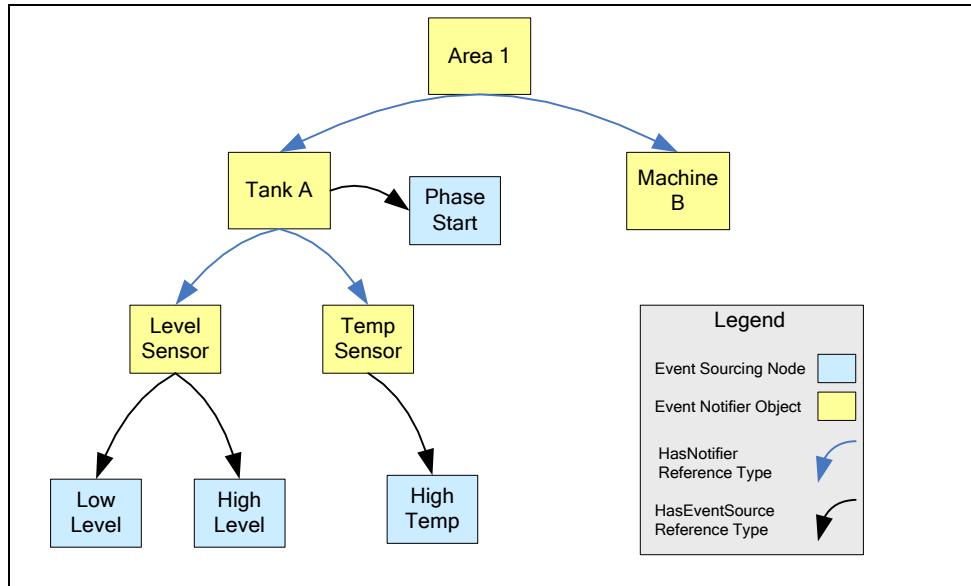
The *HasNotifier ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HasEventSource*.

The semantic of this *ReferenceType* is to relate *Object Nodes* that are notifiers with other notifier *Object Nodes*. The *ReferenceType* is used to establish a hierarchical organization of event notifying *Objects*. It is a subtype of the *HasEventSource ReferenceType* defined in Clause 6.16.

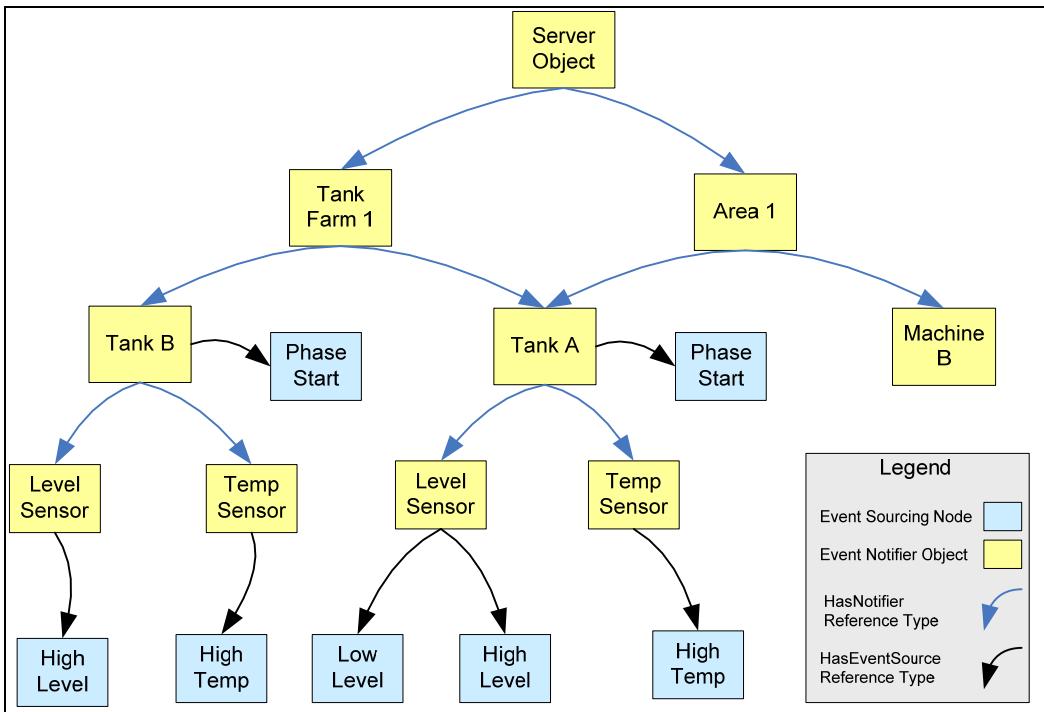
The *SourceNode* and *TargetNode* of this *ReferenceType* must be *Objects* that are a source of event subscriptions. A source of event subscriptions is an *Object* that has its “SubscribeToEvents” bit set within the *EventNotifier Attribute*.

If the *TargetNode* of a *Reference* of this type generates an *Event*, this *Event* must also be provided in the *SourceNode* of the *Reference*.

An example of a possible organization of *Event References* is represented in Figure 15. In this example an unfiltered *Event* subscription directed to the “Level Sensor” *Object* will provide the *Event* sources “Low Level” and “High Level” to the subscriber. An unfiltered *Event* subscription directed to the “Area 1” *Object* will provide *Event* sources from “Machine B”, “Tank A” and all notifier sources below “Tank A”.

**Figure 15 – Event Reference Example**

A second example of a more complex organization of *Event References* is represented in Figure 16. In this example, explicit *References* are included from the server's *Server Object*, which is a source of all server *Events*. A second *Event* organization has been introduced to collect the *Events* related to "Tank Farm 1". An unfiltered *Event* subscription directed to the "Tank Farm 1" *Object* will provide *Event* sources from "Tank B", "Tank A" and all notifier sources below "Tank B" and "Tank A".

**Figure 16 – Complex Event Reference Example**

## 7 Standard DataTypes

### 7.1 General

The following subclauses define standard *DataTypes* of OPC UA. Their representation in the *AddressSpace* and the *DataType* hierarchy is specified in [UA Part 5]. Other parts of this multi-part specification may specify additional *DataTypes*.

### 7.2 NodId

#### 7.2.1 General

*NodId*s are composed of three elements that identify a *Node* within a server. They are defined in Table 12.

**Table 12 – NodId Definition**

Name	Type	Description
NodId	structure	
namespaceIndex	UInt32	The index for a namespace URI (see Clause 7.2.2).
identifierType	Enum	The format and data type of the identifier (see Clause 7.2.3).
identifier	*	The identifier for a <i>Node</i> in the <i>AddressSpace</i> of a UA server (see Clause 7.2.4).

See [UA Part 6] for a description of the encoding of the identifier into OPC UA Messages.

#### 7.2.2 NamespaceIndex

The namespace is a URI that identifies the naming authority responsible for assigning the identifier element of the *NodId*. Naming authorities include the local server, the underlying system, standards bodies and consortia. It is expected that most *Nodes* will use the URI of the server or of the underlying system.

Using a namespace URI allows multiple OPC UA servers attached to the same underlying system to use the same identifier to identify the same *Object*. This enables clients that connect to those servers to recognise *Objects* that they have in common.

Namespace URIs, like server names, are identified by numeric values in OPC UA Services to permit more efficient transfer and processing (e.g. table lookups). The numeric values used to identify namespaces correspond to the index into the *NamespaceArray*. The *NamespaceArray* is a *Variable* that is part of the server *Object* in the *AddressSpace* (see [UA Part 5] for its definition).

The URI for the OPC UA namespace is:

```
"http://opcfoundation.org/ua/"
```

Its corresponding index in the namespace table is 0. Index 1 is reserved to identify the local server.

#### 7.2.3 IdType

The *IdType* element identifies the type of the *NodId*, its format and its scope. Its values are defined in Table 13.

**Table 13 – IdType Values**

Value	Description
NUMERIC	Numeric value
STRING	String value
URI	Universal Resource ID format. Support for this IdType is required.
GUID	Globally Unique Identifier
OPAQUE	Namespace specific format

Normally the scope of *NodeIds* is the server in which they are defined. For certain types of *NodeIds*, *NodeIds* can uniquely identify a *Node* within a system, or across systems (e.g. GUIDs). System-wide and globally-unique identifiers allow clients to track *Nodes*, such as work orders, as they move between OPC UA servers as they progress through the system.

*NodeIds* of the type GUID and URI should always be globally unique, therefore no namespace is provided for them.

Opaque identifiers are identifiers that are free-format byte strings that may or may not be human interpretable.

#### 7.2.4 Identifier value

The identifier value element is used within the context of the first three elements to identify the *Node*. Its data type and format is defined by the IdType.

A Null *NodeId* has special meaning. For example, many services defined in [UA Part 4] define special behaviour if a Null *NodeId* is passed as a parameter. Each IdType has a set of identifier values that represent a Null *NodeId*. These values are summarised in Table 14.

**Table 14 – NodeId Null Values**

IdType	Identifier
NUMERIC	0
STRING	A Null or Empty String ("")
URI	A Null or Empty String ("")
GUID	A Guid initialised with zeros (e.g. 00000000-0000-0000-0000-00000000)
OPAQUE	A ByteString with Length=0

A Null *NodeId* always has a NamespaceIndex equal to 0.

A *Node* in the *AddressSpace* may not have a Null as its *NodeId*.

#### 7.3 QualifiedName

This primitive *DataType* contains a qualified name. It is, for example, used as *BrowseName*. Its elements are defined in Table 15.

**Table 15 – QualifiedName Definition**

Name	Type	Description
QualifiedName	structure	
NamespaceIndex	UInt32	Index that identifies the namespace that defines the name. This index is the index of that namespace in the local server's <i>NamespaceArray</i> . The client may read the <i>NamespaceArray Variable</i> to access the string value of the namespace.
name	String	The unqualified name.

## 7.4 LocaleId

This primitive *DataType* is specified as a string that is composed of a language component and a country/region component as specified by RFC 3066. The <country/region> component is always preceded by a hyphen. The format of the *LocaleId* string is shown below:

<language>[-<country/region>], where  
     <language> is the two letter ISO 639 code for a language,  
     <country/region> is the two letter ISO 3166 code for the country/region.

The rules for constructing *LocaleIds* defined by RFC 3066 are restricted for OPC UA as follows:

- d) OPC UA permits only zero or one <country/region> component to follow the <language> component,
- e) OPC UA also permits the “-CHS” and “-CHT” three-letter <country/region> codes for “Simplified” and “Traditional” Chinese locales.
- f) OPC UA also allows the use of other <country/region> codes as deemed necessary by the client or the server.

Table 16 shows examples of OPC UA *LocaleIds*. Clients and servers always provide *LocaleIds* that explicitly identify the language and the country/region.

**Table 16 –LocaleId Examples**

Locale	OPC UA LocaleId
English	en
English (US)	en-US
German	de
German (Germany)	de-DE
German (Austrian)	de-AT

This *DataType* defines a special value NULL indicating that the *LocaleId* is unknown.

## 7.5 LocalizedText

This primitive *DataType* defines a structure containing a String in a locale-specific translation specified in the identifier for the locale. Its elements are defined in Table 17.

**Table 17 – LocalizedText Definition**

Name	Type	Description
LocalizedText	structure	
text	String	The localized text.
locale	LocaleId	The identifier for the locale (e.g. “en-US”).

## 7.6 Argument

This structured *DataType* defines a *Method* input or output argument specification. It is for example used in the input and output argument *Properties* for *Methods Node*. Its elements are described in Table 18.

**Table 18 – Argument Definition**

Name	Type	Description
Argument	structure	
name	String	The name of the argument
dataType	NodeId	The <i>NodeId</i> of the <i>DataType</i> of this argument
arraySize	Int32	If the <i>dataType</i> is an array it specifies the number of elements in the array. The value 0 is used to indicate an array whose size is not known. If the <i>dataType</i> is not an array, the value -1 is used.
description	LocalizedText	A localised description of the argument

## 7.7 BaseDataType

This abstract *DataType* defines a value that can have any valid *DataType*.

It defines a special value NULL indicating that a value is not present.

## 7.8 Boolean

This primitive *DataType* defines a value that is either TRUE or FALSE.

## 7.9 Byte

This primitive *DataType* defines a value in the range of 0 to 255.

## 7.10 ByteString

This primitive *DataType* defines a value that is a sequence of Byte values.

## 7.11 Date

This primitive *DataType* defines a Gregorian calendar date.

## 7.12 Double

This primitive *DataType* defines a value that adheres to the IEEE 754 Double Precision data type definition.

## 7.13 Float

This primitive *DataType* defines a value that adheres to the IEEE 754 Single Precision data type definition.

## 7.14 Guid

This primitive *DataType* defines a value that is a 128-bit Globally Unique Identifier.

## 7.15 SByte

This primitive *DataType* defines a value that is a signed integer between -128 and 127 inclusive.

## 7.16 IdType

This *DataType* is an enumeration that identifies the *IdType* of a *NodeId*. Its values are defined in Table 13. See Clause 7.2.3 for a description of the use of this *DataType* in *NodeIds*.

## 7.17 Integer

This abstract *DataType* defines an integer which length is defined by its subtypes.

## 7.18 Int16

This primitive *DataType* defines a value that is a signed integer between -32,768 and 32,767 inclusive.

## 7.19 Int32

This primitive *DataType* defines a value that is a signed integer between -2,147,483,648 and 2,147,483,647 inclusive.

## 7.20 Int64

This primitive *DataType* defines a value that is a signed integer between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 inclusive.

## 7.21 NodeClass

This *DataType* is an enumeration that identifies a *NodeClass*. Its values are defined in Table 19.

**Table 19 – NodeClass Values**

Name
DataType
Method
Object
ObjectType
ReferenceType
Variable
VariableType
View

## 7.22 Number

This abstract *DataType* defines a number. Details are defined by its subtypes.

## 7.23 String

This primitive *DataType* defines a Unicode character string that should exclude control characters that are not whitespaces (0x00 - 0x08, 0x0E-0x1F or 0x7F).

## 7.24 Time

This primitive *DataType* defines a time in terms of hours, minutes, seconds and fractions of a second. Its granularity is specified by its encoding in [UA Part 6].

## 7.25 UInteger

This abstract *DataType* defines an unsigned integer which length is defined by its subtypes.

## 7.26 UInt16

This primitive *DataType* defines a value that is an unsigned integer between 0 and 65,535 inclusive.

## 7.27 UInt32

This primitive *DataType* defines a value that is an unsigned integer between 0 and 4,294,967,295 inclusive.

## 7.28 UInt64

This primitive *DataType* defines a value that is an unsigned integer between 0 and 18,446,744,073,709,551,615 inclusive.

## 7.29 UtcTime

This primitive *DataType* is used to define Coordinated Universal Time (UTC) values. All time values conveyed between servers and clients in OPC UA are UTC values. Clients must provide any conversions between UTC and local time.

This *DataType* is represented as a 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601. [UA Part 6] defines details about this *DataType*.

## 7.30 XmlElement

This primitive *DataType* is used to define XML elements. [UA Part 6] defines details about this *data type*.

## 8 Standard EventTypes

### 8.1 General

The following subclauses define standard *EventTypes* of OPC UA. Their representation in the *AddressSpace* is specified in [UA Part 5]. Other parts of this multi-part specification may specify additional *EventTypes*. Figure 17 informally describes the hierarchy of these standard *EventTypes*.

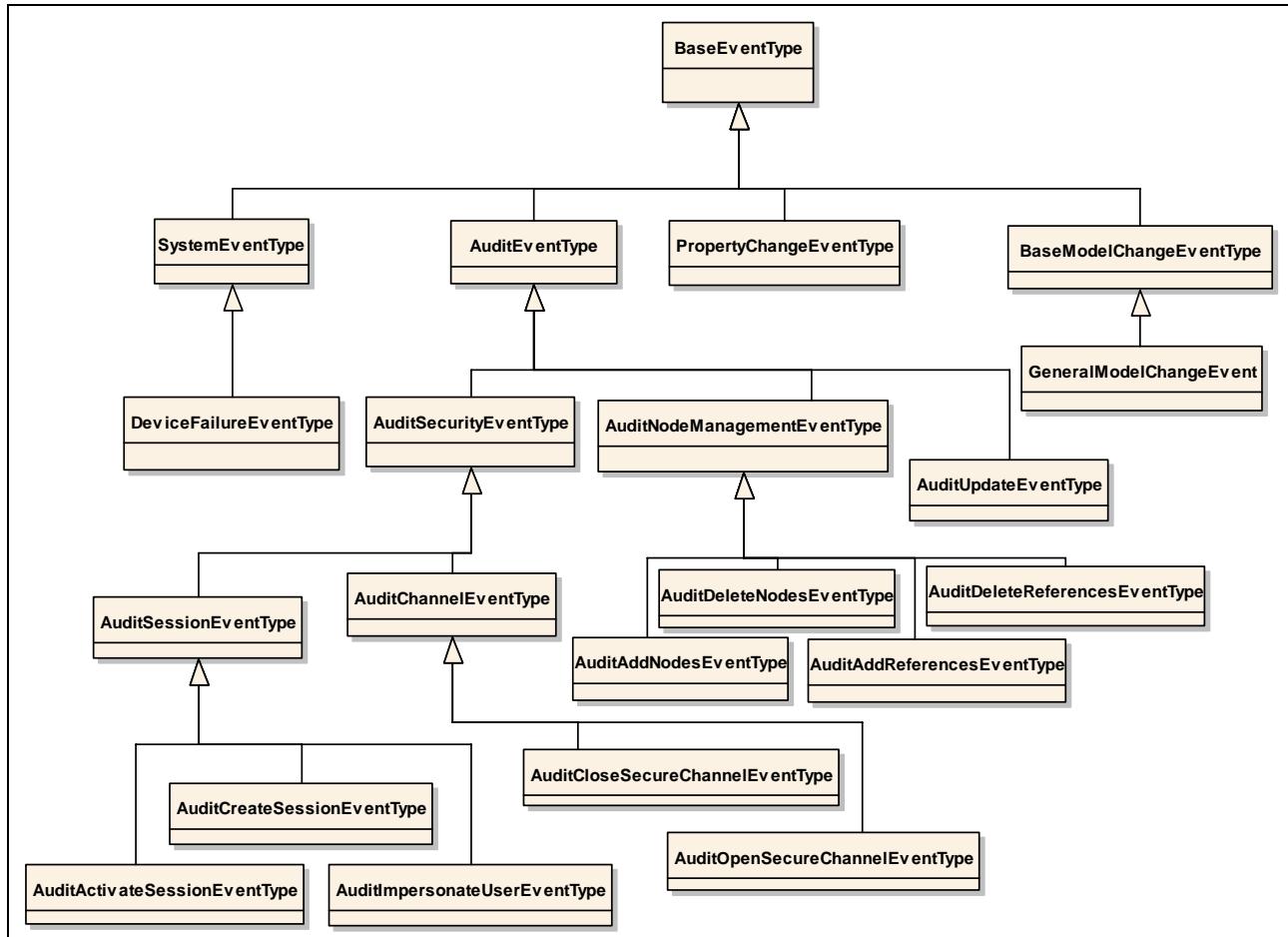


Figure 17 – Standard EventType Hierarchy

### 8.2 BaseEventType

The *BaseEventType* defines all general characteristics of an *Event*. All other *EventTypes* derive from it. There is no other semantic associated with this type.

### 8.3 SystemEventType

*SystemEvents* are generated as a result of some *Event* that occurs within the server or by a system that the server is representing.

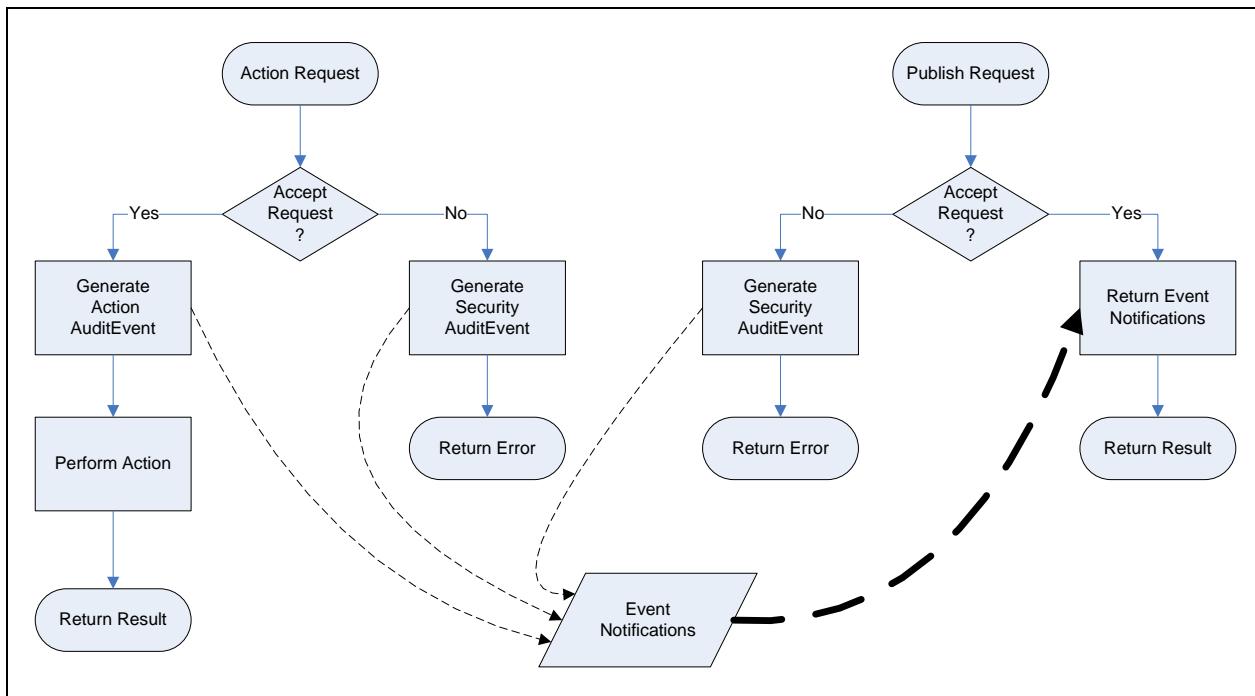
### 8.4 AuditEventType

*AuditEvents* are generated as a result of an action taken on the server by a client of the server. For example, in response to a client issuing a write to a *Variable*, the server would generate an *AuditEvent* describing the *Variable* as the source and the user and client session as the initiators of the *Event*.

Figure 18 illustrates the OPC UA defined behaviour of a server in response to an auditable action request. If the action is accepted, an action *AuditEvent* is generated and processed by the server. If the action is not accepted due to security reasons, a security *AuditEvent* is generated and processed by the server. The server may involve the underlying device or system in the process but it is the server's responsibility to provide the *Event* to any interested clients. Clients are free to subscribe to *Events* from the server and will receive the *AuditEvents* in response to normal Publish requests.

All action requests include a human readable *AuditEntryId*. The *AuditEntryId* is included in the *AuditEvent* to allow human readers to correlate an *Event* with the initiating action. The *AuditEntryId* typically contains who initiated the action and from where it was initiated.

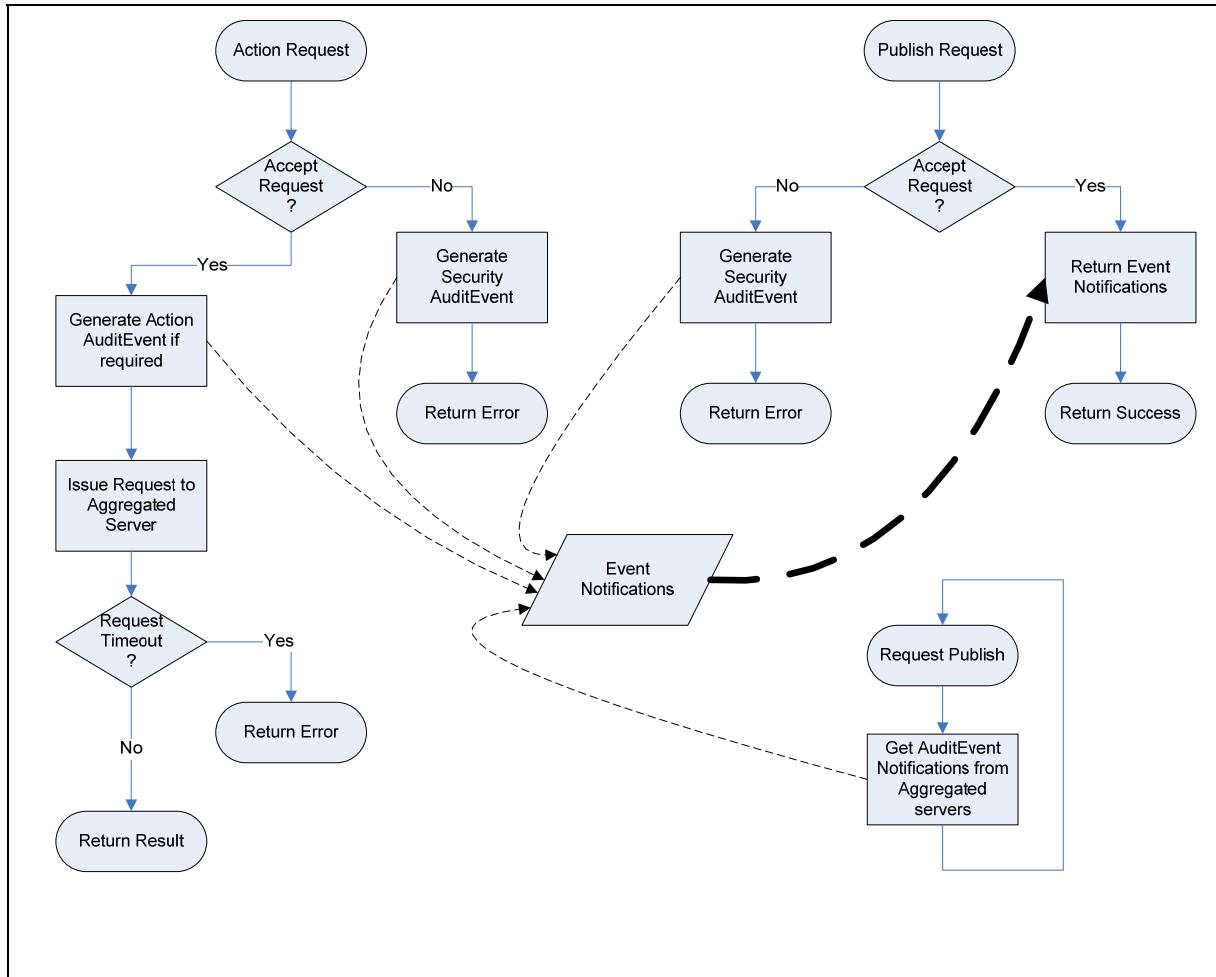
The Server may elect to optionally persist the *AuditEvents* in addition to the mandatory *Event Subscription* delivery to clients.



**Figure 18 – Audit Behaviour of a Server**

Figure 19 illustrates the expected behaviour of an aggregating server in response to an auditable action request. This use case involves the aggregating server passing on the action to one of its aggregated servers. The general behaviour described above is extended by this behaviour and not replaced. That is, the request could fail and generate a security *AuditEvent* within the aggregating server. The normal process is to pass the action down to an aggregated server for processing. The aggregated server will, in turn, follow this behaviour or the general behaviour and generate the appropriate *AuditEvents*. The aggregating server periodically issues publish requests to the aggregated servers. These collected *Events* are merged with self-generated *Events* and made available to subscribing clients. If the aggregating server supports the optional persisting of *AuditEvent*, the collected *Events* are persisted along with locally-generated *Events*.

The aggregating server may map the authenticated user account making the request to one of its own accounts when passing on the request to an aggregated server. It must, however, preserve the *AuditEntryId* by passing it on as received. The aggregating server may also generate its own *AuditEvent* for the request prior to passing it on to the aggregated server, in particular, if the aggregating server needs to break a request into multiple requests that are each directed to separate aggregated servers or if part of a request is denied due to security on the aggregating server.



**Figure 19 – Audit Behaviour of an Aggregating Server**

## 8.5 AuditSecurityEventType

This is a subtype of AuditEventType and is used only for categorization of security-related *Events*. This type follows all behaviour of its parent type.

## 8.6 AuditChannelEventType

This is a subtype of AuditSecurityEventType and is used for categorization of security-related *Events* from the *SecureChannel Service Set* defined in [UA Part 4].

## 8.7 AuditOpenSecureChannelEventType

This is a subtype of AuditChannelEventType and is used for *Events* generated from calling the *OpenSecureChannel Service* defined in [UA Part 4].

## 8.8 AuditCloseSecureChannelEventType

This is a subtype of AuditChannelEventType and is used for *Events* generated from calling the *CloseSecureChannel Service* defined in [UA Part 4].

## 8.9 AuditSessionEventType

This is a subtype of AuditSecurityEventType and is used for categorization of security-related *Events* from the *Session Service Set* defined in [UA Part 4].

## 8.10 AuditCreateSessionEventType

This is a subtype of AuditSessionEventType and is used for *Events* generated from calling the CreateSession Service defined in [UA Part 4].

## 8.11 AuditActivateSessionEventType

This is a subtype of AuditSessionEventType and is used for *Events* generated from calling the ActivateSession Service defined in [UA Part 4].

## 8.12 AuditImpersonateUserEventType

This is a subtype of AuditSessionEventType and is used for *Events* generated from calling the ImpersonateUser Service defined in [UA Part 4].

## 8.13 AuditNodeManagementEventType

This is a subtype of AuditEvent-Type and is used for categorization of node management related *Events*. This type follows all behaviour of its parent type.

## 8.14 AuditAddNodesEventType

This is a subtype of AuditNodeManagementEventType and is used for *Events* generated from calling the AddNodes Service defined in [UA Part 4].

## 8.15 AuditDeleteNodesEventType

This is a subtype of AuditNodeManagementEventType and is used for *Events* generated from calling the DeleteNodes Service defined in [UA Part 4].

## 8.16 AuditAddReferencesEventType

This is a subtype of AuditNodeManagementEventType and is used for *Events* generated from calling the AddReferences Service defined in [UA Part 4].

## 8.17 AuditDeleteReferencesEventType

This is a subtype of AuditNodeManagementEventType and is used for *Events* generated from calling the DeleteReferences Service defined in [UA Part 4].

## 8.18 AuditUpdateEventType

This is a subtype of AuditEvent-Type and is used for categorization of update related *Events*. This type follows all behaviour of its parent type.

## 8.19 DeviceFailureEventType

A *DeviceFailureEvent* indicates a failure in a device of the underlying system.

## 8.20 ModelChangeEvents

### 8.20.1 General

*ModelChangeEvents* are generated to indicate a change of the *AddressSpace* structure. The change may consist of adding or deleting a *Node* or *Reference*. Although the relationship of a *Variable* or *VariableType* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *Variable* or *VariableType* are also considered as model changes and therefore a *ModelChangeEvent* is generated if the *DataType Attribute* changes.

### 8.20.2 NodeVersion Property

There is a correlation between *ModelChangeEvents* and the *NodeVersion* Property of *Nodes*. Every time a *ModelChangeEvent* is issued for a *Node*, its *NodeVersion* must be changed, and every time the *NodeVersion* is changed, a *ModelChangeEvent* must be generated. A server must support both the *ModelChangeEvent* and the *NodeVersion* Property or neither, but never only one of the two mechanisms.

### 8.20.3 Views

A *ModelChangeEvent* is always generated in the context of a *View* including the default *View* where the whole *AddressSpace* is considered. Thus each action generating a *ModelChangeEvent* may lead to several *Events* since it may affect different *Views*. If, for example, a *Node* was deleted from the *AddressSpace*, and this *Node* was also contained in a *View* "A", there would be one *Event* having the *AddressSpace* as context and another having the *View* "A" as context. If a *Node* would only be removed from *View* "A", but still exists in the *AddressSpace*, it would generate only a *ModelChangeEvent* for *View* "A".

If a client does not want to receive duplicates of changes it has to use the filter mechanisms of the *Event* subscription filtering only for the default *View* and suppress the *ModelChangeEvents* having other *Views* as context.

When a *ModelChangeEvent* is issued on a *View* and the *View* supports the *ViewVersion* Property, the *ViewVersion* has to be updated.

### 8.20.4 Event Compression

An implementation is not required to issue an *Event* for every update as it occurs. A UA Server may be capable of grouping a series of transactions or simple updates into a larger unit. This series may constitute a logical grouping or a temporal grouping of changes. A single *ModelChangeEvent* may be issued after the last change of the series, to cover all of the changes. This is referred to as *Event compression*. A change in the *NodeVersion* and the *ViewVersion* may thus reflect a group of changes and not a single change.

### 8.20.5 BaseModelChangeEventType

The *BaseModelChangeEventType* is the base type for *ModelChangeEvents* and does not contain information about the changes but only indicates that changes occurred. Therefore the client must assume that any or all of the *Nodes* may have changed.

### 8.20.6 GeneralModelChangeEventType

The *GeneralModelChangeEventType* is a subtype of the *BaseModelChangeEventType*. It contains information about the *Node* that was changed and the action that occurred the *ModelChangeEvent* (e.g. add a *Node*, delete a *Node*, etc.). If the affected *Node* is a *Variable* or *Object*, the *TypeDefinitionNode* is also present.

To allow *Event* compression, a *GeneralModelChangeEvent* contains an array of this structure.

### 8.20.7 Guidelines for ModelChangeEvents

Two types of standard *ModelChangeEvents* are defined: the *BaseModelChangeEvent* that does not contain any information about the changes and the *GeneralModelChangeEvent* that identifies the changed *Nodes* via an array. The precision used depends on both the capability of the UA server and the nature of the update. A UA server may use either *ModelChangeEvent* type depending on circumstances. It may also define subtypes of these *EventTypes* adding additional information.

To ensure interoperability, the following guidelines for *Events* should be observed:

- If the array of the *GeneralModelChangeEvent* is present, then it should identify every *Node* that has changed since the preceding *ModelChangeEvent*.
- The UA server should emit exactly one *ModelChangeEvent* for an update or series of updates. It should not issue multiple types of *ModelChangeEvent* for the same update.
- Any client that responds to *ModelChangeEvents* should respond to any *Event* of the *BaseModelChangeEvent* type including its subtypes like the *GeneralModelChangeEvent*.

If a client is not capable of interpreting additional information of the subtypes of the *BaseModelChangeEvent*, it should treat *Events* of these types the same way as *Events* of the *BaseModelChangeEvent*.

## 8.21 PropertyChangeEvent Type

### 8.21.1 General

*PropertyChangeEvents* are generated to indicate a change of the *AddressSpace* semantics. The change consists of a change to the *Value Attribute* of a *Property*.

The *PropertyChangeEvent* contains information about the *Node* owning the *Property* that was changed. If this is a *Variable* or *Object*, the *TypeDefinitionNode* is also present.

*PropertyChangeEvents* are not generated by every change of a *Property*, but only when the *Property* describes the semantic of the *Node* that owns the *Property*. For example, a change in a *Property* describing the engineering unit of a *DataVariable* will issue a *PropertyChangeEvent*, whereas the change of a *Property* containing an *Icon* of the *DataVariable* will not. For *Variables* and *VariableTypes* this behaviour is exactly the same as described by the *SemanticsChanged* bit of the *StatusCodes* defined in [UA Part 4]. However, if you subscribe to a *Variable* you should look at the *StatusCodes* to identify if the semantic has changed in order to receive this information before you are processing the value of the *Variable*.

### 8.21.2 ViewVersion and NodeVersion Properties

The *ViewVersion* and *NodeVersion* *Properties* do not change due to the publication of a *PropertyChangeEvent*.

### 8.21.3 Views

*PropertyChangeEvents* are handled in the context of a *View* the same way as *ModelChangeEvents*. This is defined in Clause 8.20.3.

### 8.21.4 Event Compression

*PropertyChangeEvents* can be compressed the same way as *ModelChangeEvents*. This is defined in Clause 8.20.4.

## Appendix A: How to use the Address Space Model

### A.1 Overview

This Appendix points out some general considerations how the Address Space Model can be used. The Appendix is informative, that is each server vendor can model its data in the appropriated way that fits to its needs. However, it gives some hints the server vendor may consider.

Typically OPC UA server will offer data provided by an underlying system like a device, a configuration database, an OPC COM server, etc. Therefore the modelling of the data depends on the model of the underlying system as well as the requirements on the clients accessing the OPC UA server. It is also expected that companion standards will be developed on top of OPC UA with additional rules how to model the data. However, the following subclauses will give some general consideration about the different concepts of OPC UA to model data and when they should be used and when not.

The Appendix of [UA Part 5] gives an overview over the design decisions made when modelling the information about the server defined in [UA Part 5].

### A.2 Type definitions

Type definitions should be used whenever it is expected that the type information may be used more than once in the same system or for interoperability between different systems supporting the same type definitions.

### A.3 ObjectTypes

Clause 5.5.1 states: “*Objects* are used to represent systems, system components, real-world objects, and software objects.” Therefore *ObjectTypes* should be used if a type definition of those is useful (see A.2).

From a more abstract point of view *Objects* are used to group *Variables* and other *Objects* in the *AddressSpace*. Therefore *ObjectTypes* should be used when some common structures / groups of *Objects* and / or *Variables* should be described. Clients can use this knowledge to program against the *ObjectType* structure and use the *TranslateBrowsePathsToNodeIds Service* defined in [UA Part 4] on the instances.

Simple objects only having one value (e.g. a simple heat sensor) can also be modelled as *VariableTypes*. However, extensibility mechanisms should be considered (e.g. a complex heat sensor subtype could have several values) and whether the object should be exposed as an object in the client's GUI or just as a value. Whenever a modeller is in doubt which solution to use the *ObjectType* having one *Variable* should be preferred.

### A.4 VariableTypes

#### A.4.1 General

*VariableTypes* are only used for *DataVariables*<sup>3</sup> and should be used when there are several *Variables* having the same semantic (e.g. set point). It is not needed to define a *VariableType* just reflecting the *DataType* of the *Variable*, e.g. an “*Int32VariableType*”.

---

<sup>3</sup> *VariableTypes* other then the *PropertyType* which is used for all *Properties*

#### A.4.2 Properties or DataVariables

Besides the semantic differences of *Properties* and *DataVariables* described in Clause 4 there are also syntactic differences. A *Property* is identified by its *BrowseName*, i.e. if *Properties* having the same semantic are used several times, they should always have the same *BrowseName*. The same semantic of *DataVariables* is captured in the *VariableType*.

If it's not clear what concept to use based on the semantic described in Clause 4, the different syntax can help. The following points identify that it has to be a *DataVariable*:

- If it's a complex *Variable* or it should contain additional information in the form of *Properties*.
- If the type definition may be refined (subtyping).
- If the type definition should be made available so the client can use the AddNodes Service defined in [UA Part 4] to create new instances of the type definition.
- If it's a component of a complex *Variable* exposing a part of the value of the complex *Variable*.

If none of the above applies and the semantic described in Clause 4 does not make it clear that it has to be a *DataVariable*, it is useful making it a *Property* since *Properties* are easier to handle for the client (e.g. with the BrowseProperties Service defined in [UA Part 4]).

#### A.4.3 Many Variables and / or complex DataTypes

When complex data structures should be made available to the client there are basically three different approaches:

- 1) Create several simple *Variables* using simple *DataTypes* always reflecting parts of the simple structure. *Objects* are used to group the *Variables* according to the structure of the data.
- 2) Create a complex *DataType* and a simple *Variable* using this *DataType*.
- 3) Create a complex *DataType* and a complex *Variable* using this *DataType* and also exposing the complex data structure as *Variables* of the complex *Variable* using simple *DataTypes*.

The advantages of the first approach are that the complex structure of the data is visible in the *AddressSpace*; a generic client can easily access those data without knowledge of user-defined *DataTypes*; and the client can access individual parts of the complex data. The disadvantages of the first approach are that accessing the individual data does not provide any transactional context; and for a specific client the server first has to convert the data and the client has to convert the data, again, to get the data structure the underlying system provides.

The advantages of the second approach are, that the data are accessed in a transaction context and the complex *DataType* can be constructed in a way that the server does not have to convert the data and can pass them directly to the specific client that can directly use them. The disadvantages are that the generic client may not be able to access and interpret the data or has at least the burden to read the *DataTypeDescription* to interpret the data. The structure of the data is not visible in the *AddressSpace*; additional *Properties* describing the data structure cannot be added to the adequate places since they do not exist in the *AddressSpace*. Individual parts of the data cannot be read without accessing the whole data structure.

The third approach combines both other approaches. Therefore the specific client can access the data in its native format in a transactional context, whereas the generic client can access the simple *DataTypes* of the components of the complex *Variable*. The disadvantage is that the server must be able to provide the native format and also interpret it to be able to provide the information in simple *DataTypes*.

It is recommended to use the first approach. When a transactional context is needed or the client should be able to get a large amount of data instead of subscribing to several individual values, the third approach is suitable. However, the server may not always have the knowledge to interpret the complex data of the underlying system and therefore has to use the second approach just passing the data to the specific client who is able to interpret the data.

## A.5 Views

Server-defined *Views* can be used to present an excerpt of the *AddressSpace* suitable for a special class of clients, e.g. maintenance clients, engineering clients, etc. The *View* only provides the information needed for the purpose of the client and hides unnecessary information.

## A.6 Methods

*Methods* should be used whenever some input is expected and the server delivers a result. One should avoid using *Variables* to write the input values and other *Variables* to get the output results as it was needed to do in OPC COM since there was no concept of a *Method* available. However, a simple OPC COM wrapper may not be able to do this.

*Methods* can also be used to trigger some execution in the server that does not require input and / or output parameters.

Global *Methods*, i.e. *Methods* that cannot directly be assigned to a special *Object*, should be assigned to the *Server Object* defined in [UA Part 5].

## A.7 Defining ReferenceTypes

Defining new *ReferenceTypes* should only be done if the predefined *ReferenceTypes* are not suitable. Whenever a new *ReferenceType* is defined, the most appropriate *ReferenceType* should be used as its supertype.

It is expected that servers will have new defined hierarchical *ReferenceTypes* to expose different hierarchies and new non-hierarchical *References* to expose relationships between *Nodes* in the *AddressSpace*.

## A.8 Defining ModellingRules

New *ModellingRules* have to be defined if the predefined *ModellingRules* are not appropriated for the model exposed by the server.

Depending on the model used by the underlying system the server may need to define new *ModellingRules*, since the OPC UA server may only pass the data to the underlying system and this system may use its own internal rules for instantiation, subtyping, etc.

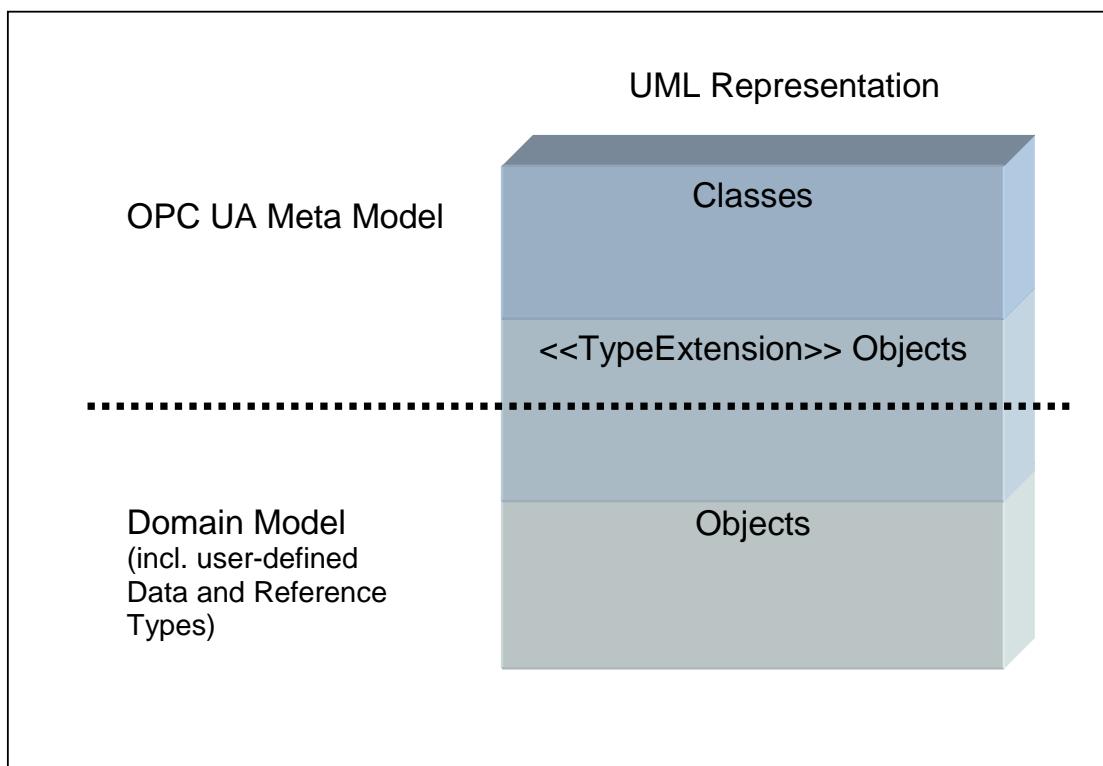
Beside this the predefined *ModellingRules* may not be sufficient to specify the needed behaviour for instantiation and subtyping.

## Appendix B: OPC UA Meta Model in UML

### B.1 Background

The OPC UA Meta Model (the OPC UA Address Space Model) is represented by UML classes and UML objects marked with the stereotype <<TypeExtension>>. Those stereotyped UML objects represent *DataTypes* or *ReferenceTypes*. The domain model can contain user-defined *ReferenceTypes* and *DataTypes*, also marked as <<TypeExtension>>. In addition, the domain model contains *ObjectTypes*, *VariableTypes* etc. represented as UML objects (see Figure 20).

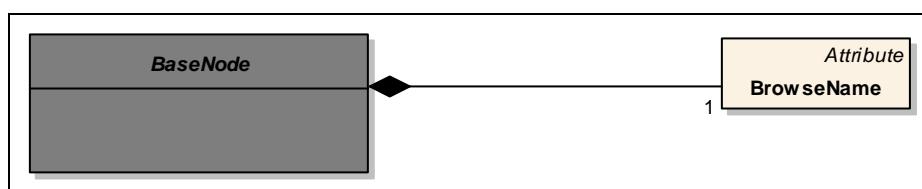
The OPC Foundation specifies not only the OPC UA Meta Model, but also defines some *Nodes* to organise the *AddressSpace* and to provide information about the server as specified in [UA Part 5].



**Figure 20 – Background of OPC UA Meta Model**

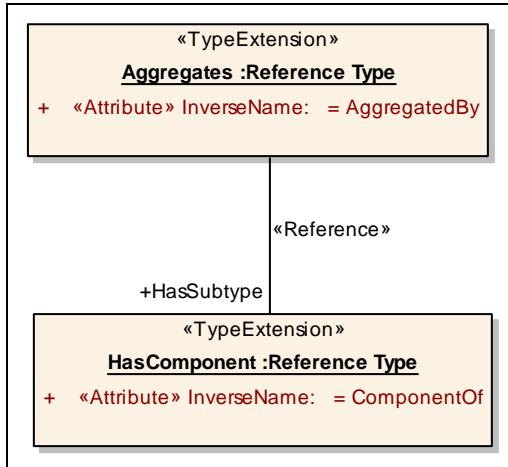
### B.2 Notation

An example of a UML class representing the OPC UA concept *BaseNode* is given in the UML class diagram in Figure 21. OPC Attributes inherit from the abstract class *Attribute* and have a value identifying their data type. They are composed to a *Node* either optional (0..1) or required (1), like *BrowseName* to *BaseNode* in Figure 21.



**Figure 21 – Notation (I)**

UML object diagrams are used to display <<TypeExtension>> objects (e.g. *HasComponent* in Figure 22). In object diagrams, OPC *Attributes* are represented as UML attributes without data types and marked with the stereotype <<Attribute>>, like *InverseName* in the UML object *HasComponent*. They have values, like *InverseName* = *ComponentOf* for *HasComponent*. To keep the object diagrams simple, not all *Attributes* are shown (e.g. the *NodeId* of *HasComponent*).



**Figure 22 – Notation (II)**

OPC References are represented as UML associations marked with the stereotype <<Reference>>. If a particular *ReferenceType* is used, its name is used as role name; identifying the direction of the Reference (e.g. *Aggregates* has the subtype *HasComponent*). For simplicity, the inverse role name is not shown (in the example *SubclassOf*). When no role name is provided, it means that any *ReferenceType* can be used (only valid for class diagrams).

There are some special *Attributes* in OPC UA containing a *NodeId* and thereby referencing another *Node*. Those *Attributes* are represented as associations marked with the stereotype <<Attribute>>. The name of the *Attribute* is displayed as role name of the *TargetNode*.

The value of the OPC *Attribute BrowseName* is represented by the UML object name, e.g. the *BrowseName* of the UML object *HasComponent* in Figure 22 is "HasComponent".

To highlight the classes explained in a class diagram, they are marked grey (e.g. *BaseNode* in Figure 21). Only those classes have all their relationships to other classes and attributes shown in the diagram. For the other classes, we provide only those attributes and relationships needed to understand the main classes of the diagram.

### B.3 Meta Model

Remark: Other parts of this multi-part specification can extend the OPC UA Meta Model by adding *Attributes* and defining new *ReferenceTypes*.

### B.3.1 BaseNode

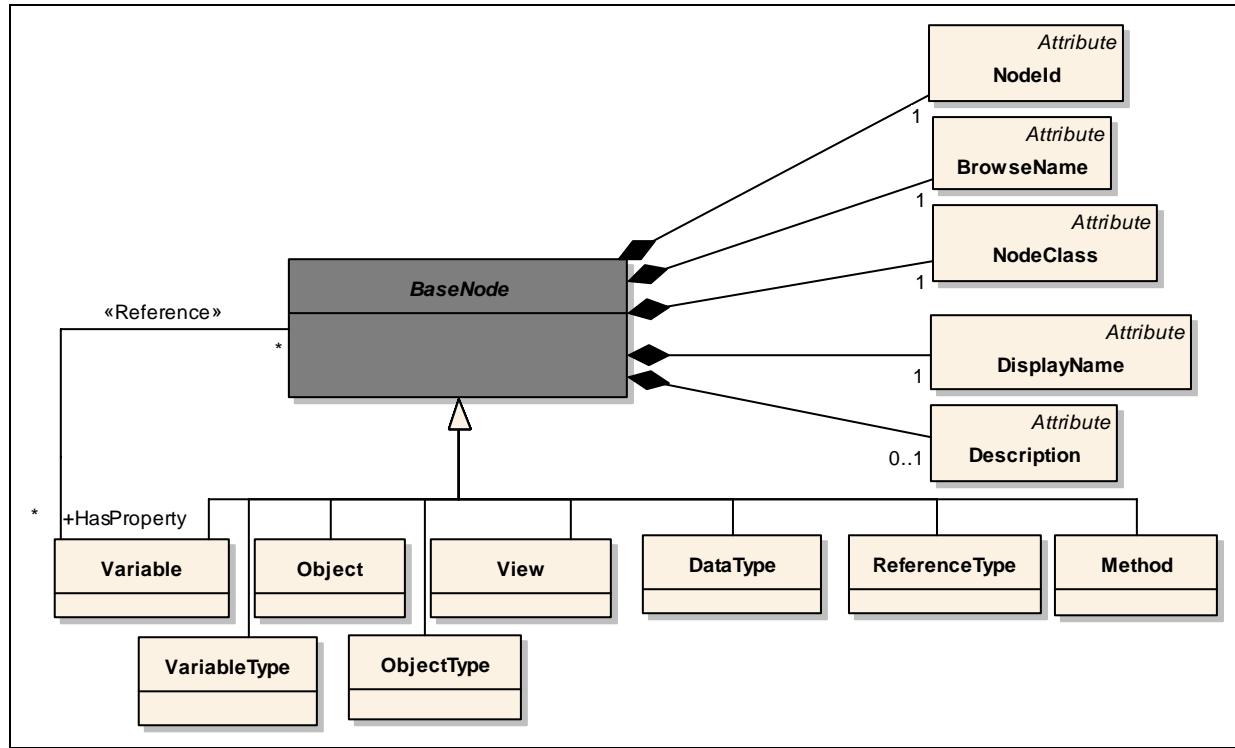


Figure 23 – BaseNode

### B.3.2 ReferenceType

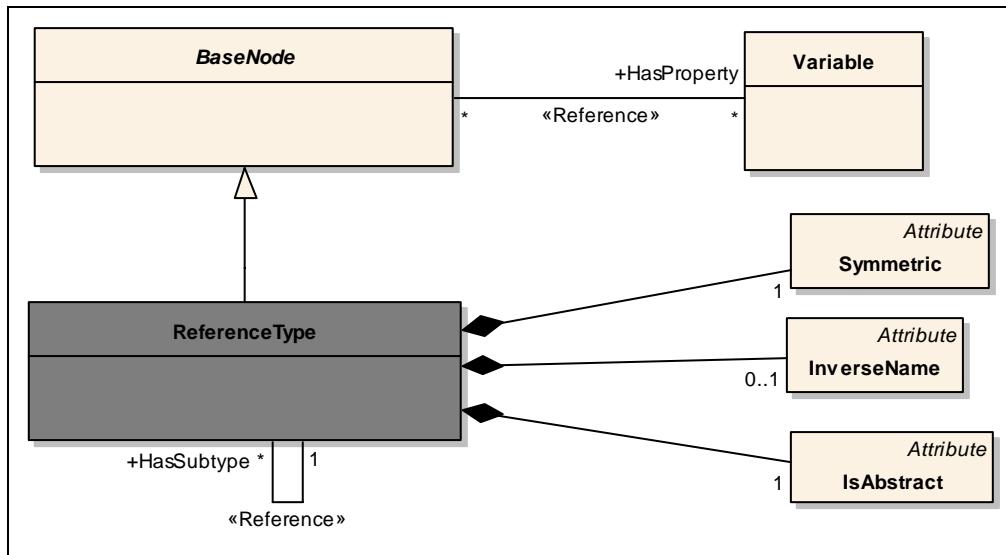


Figure 24 – Reference and ReferenceType

If *Symmetric* is “false” and *IsAbstract* is “false” an *InverseName* must be provided.

### B.3.3 Predefined ReferenceTypes

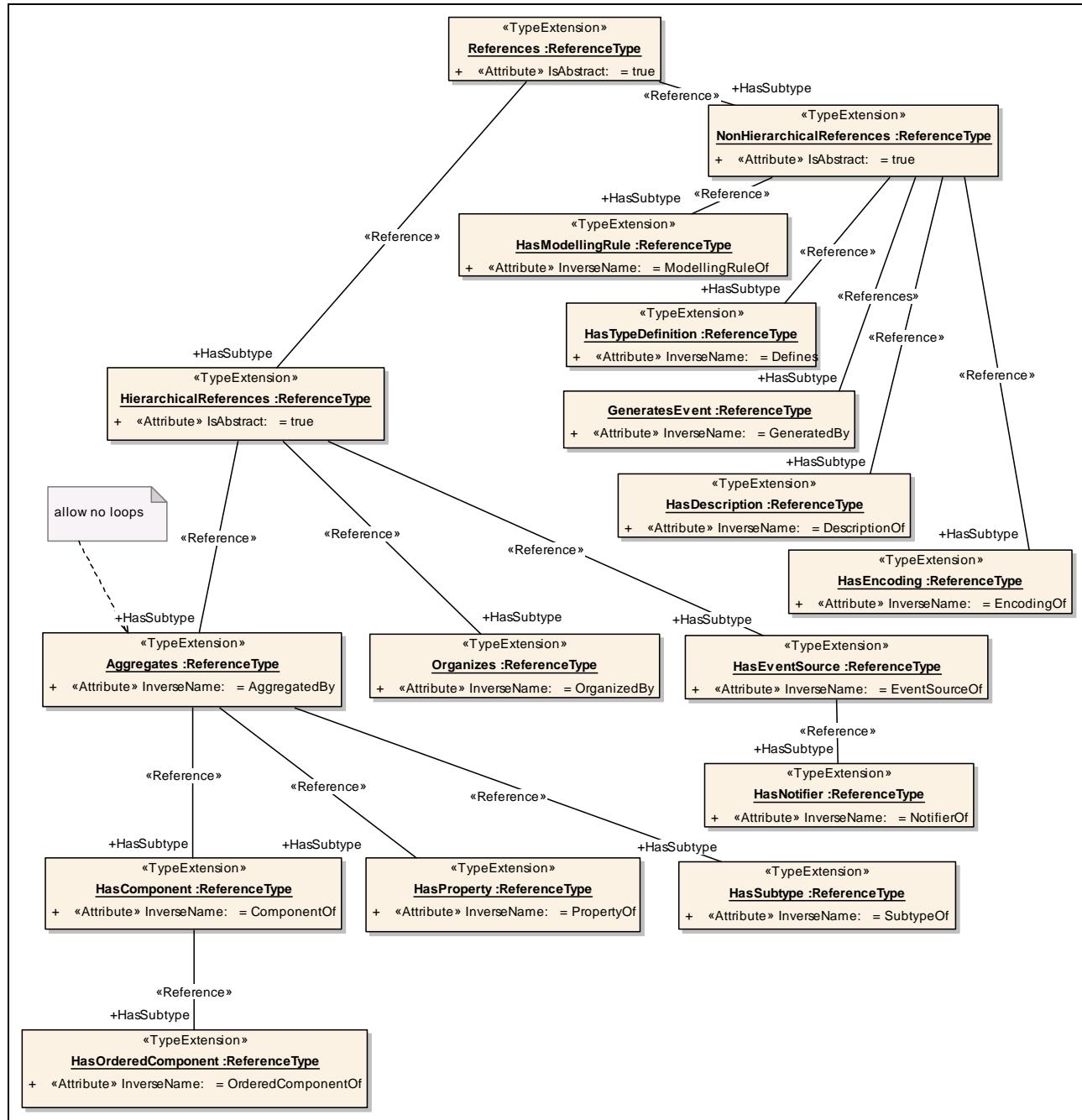
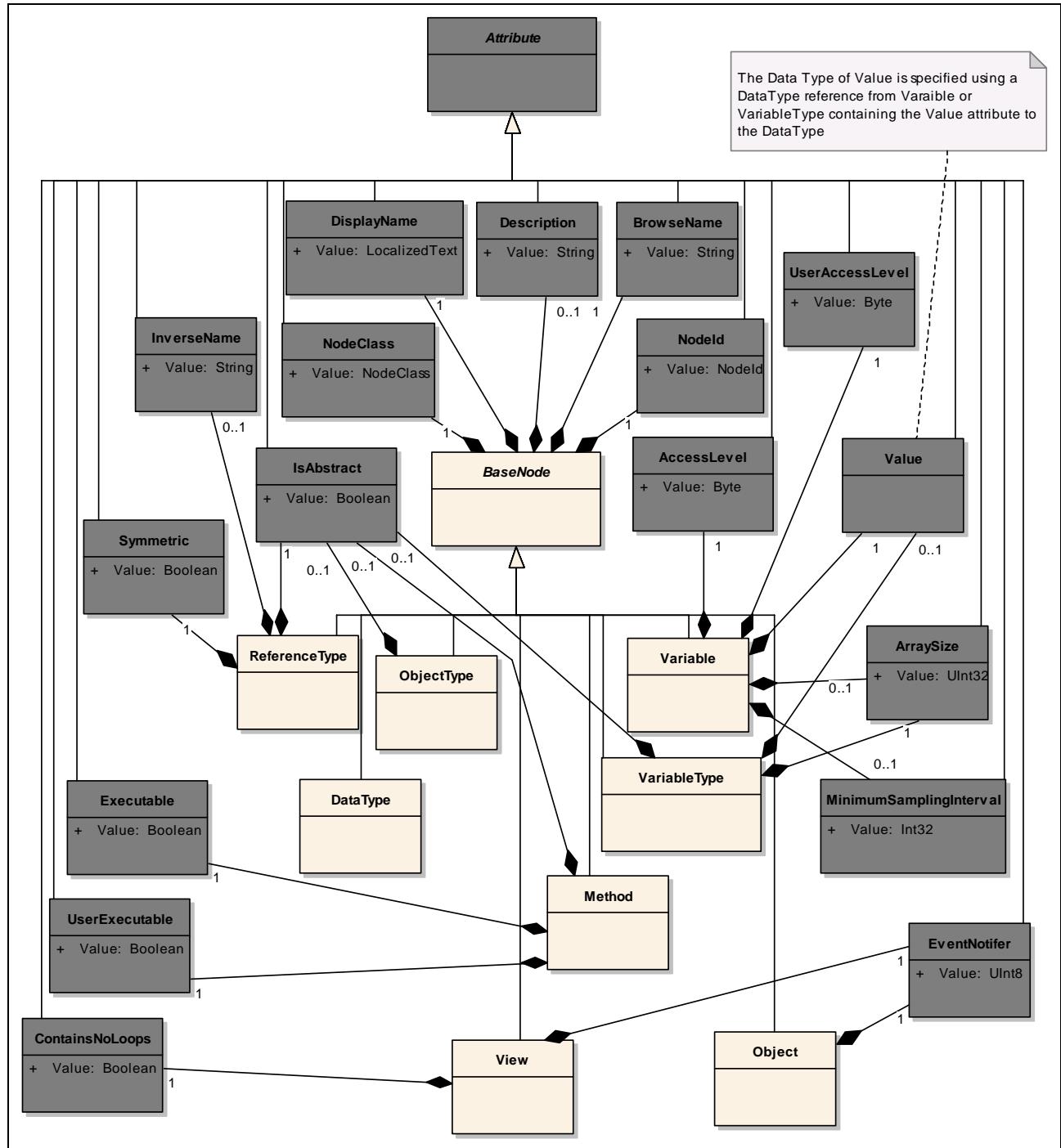


Figure 25 – Predefined ReferenceTypes

### B.3.4 Attributes



**Figure 26 – Attributes**

There may be more *Attributes* defined in other parts of the standard.

*Attributes* used for references, which have a **NodeId** as **DataType**, are not shown in this diagram but as stereotyped associations in the other diagrams.

### B.3.5 Object and ObjectType

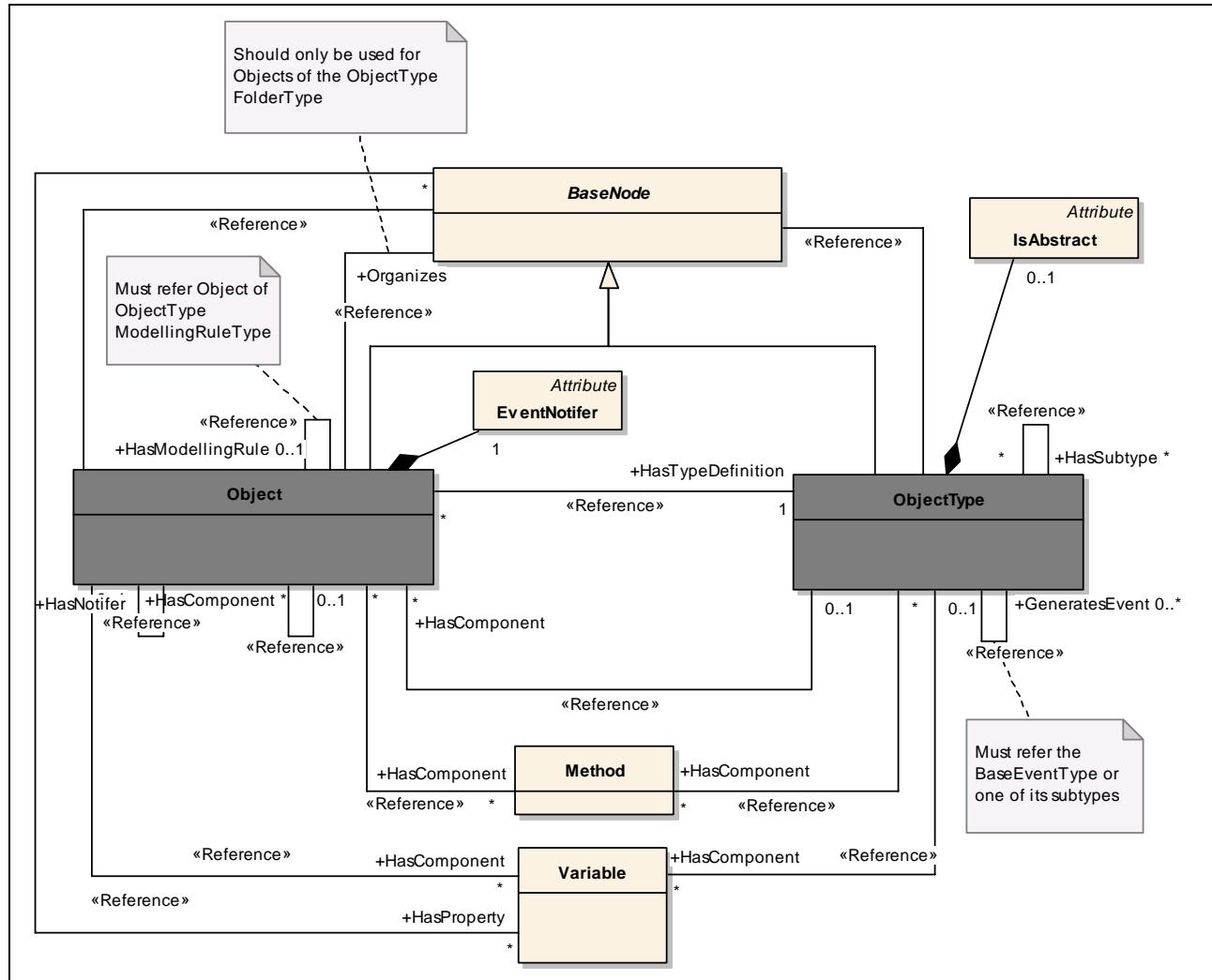
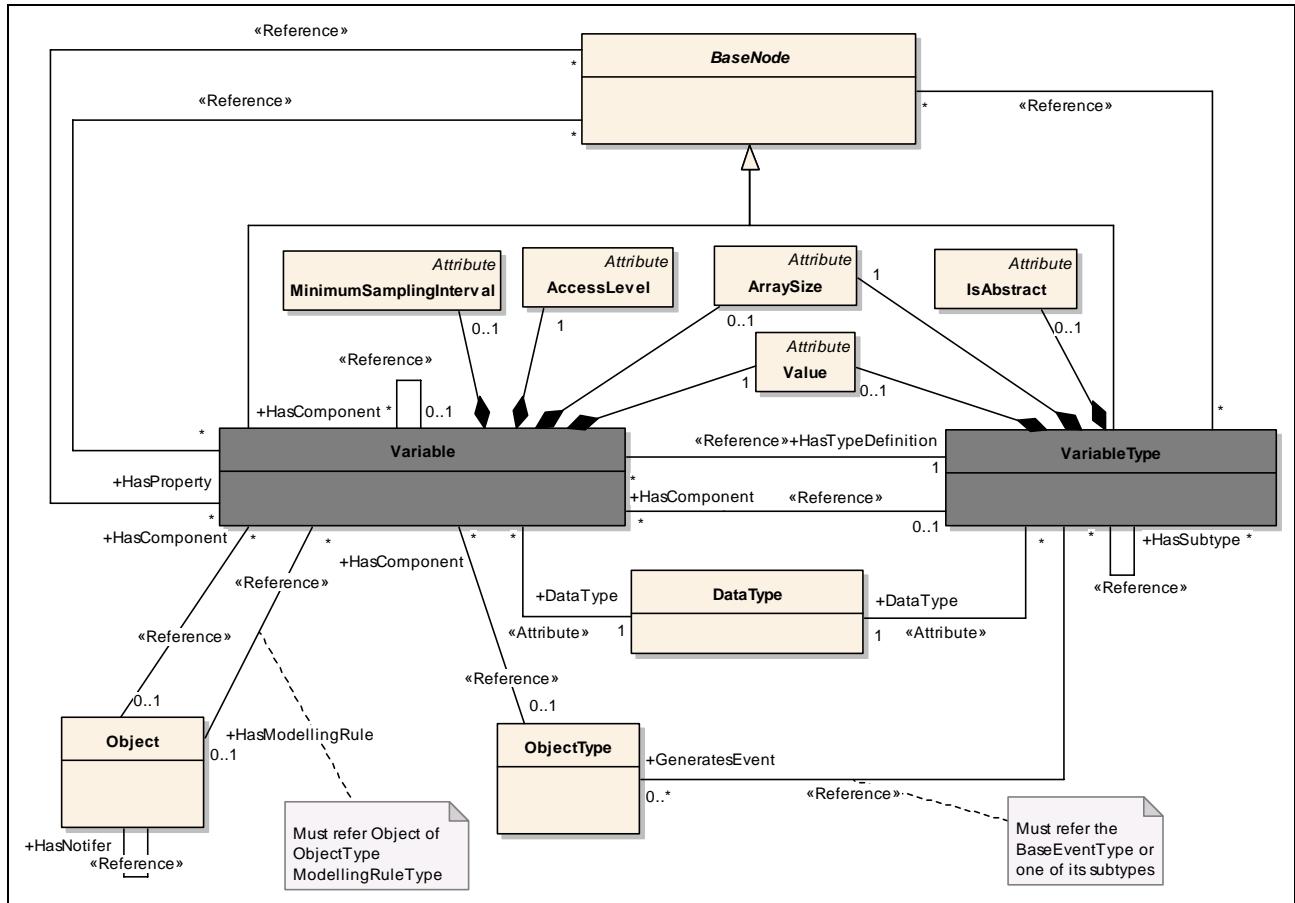


Figure 27 – Object and ObjectType

### B.3.6 Variable and VariableType



**Figure 28 – Variable and VariableType**

The *DataType* of a *Variable* must be the same or a subtype of the *DataType* of its *VariableType* (referred with *HasTypeDefintion*).

If a *HasProperty* points to a *Variable* from a *BaseNode* “A” the following constraints apply:

The *Variable* must not be the *SourceNode* of a *HasProperty* or any other *HierarchicalReferences Reference*.

All *Variables* having “A” as the *SourceNode* of a *HasProperty Reference* must have a unique *BrowseName* in the context of “A”.

### B.3.7 Method

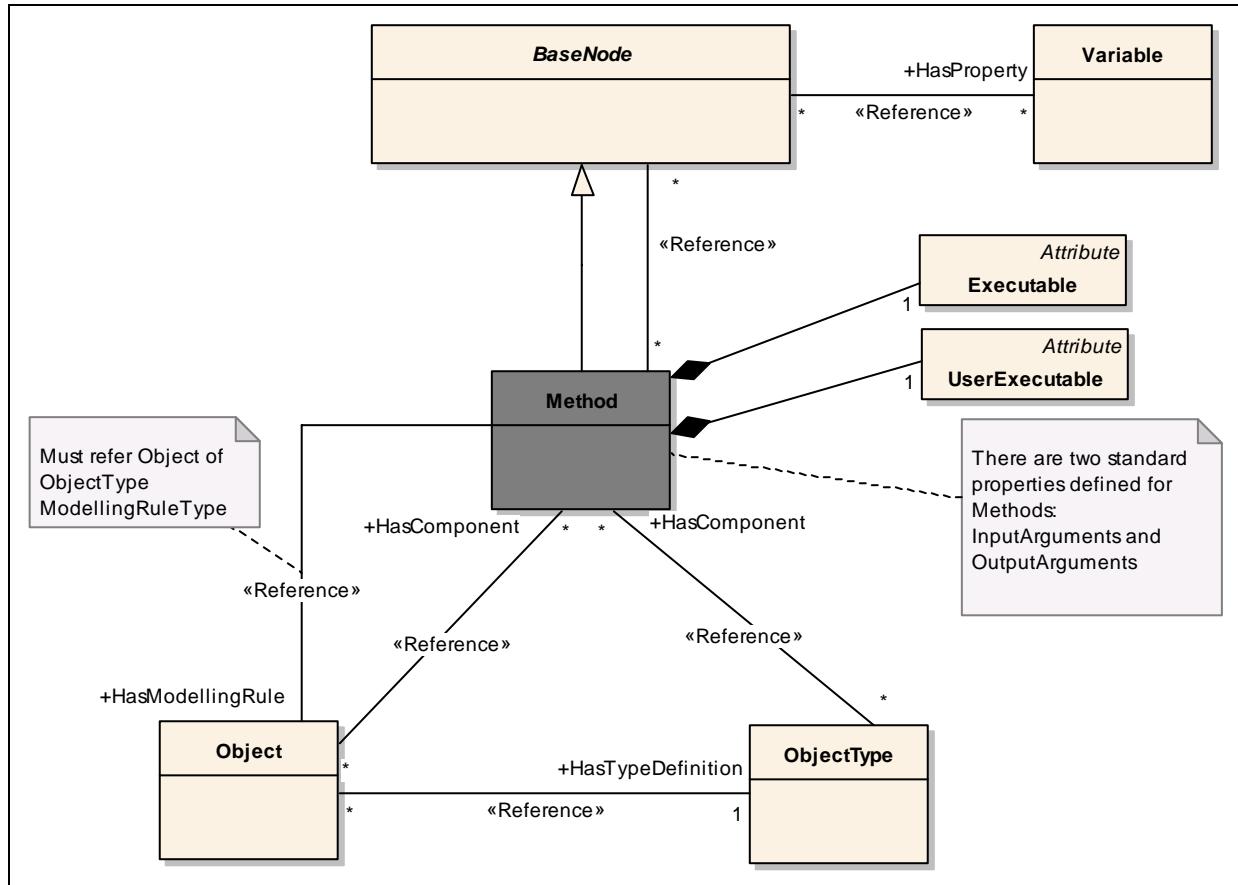


Figure 29 – Method

### B.3.8 EventNotifier

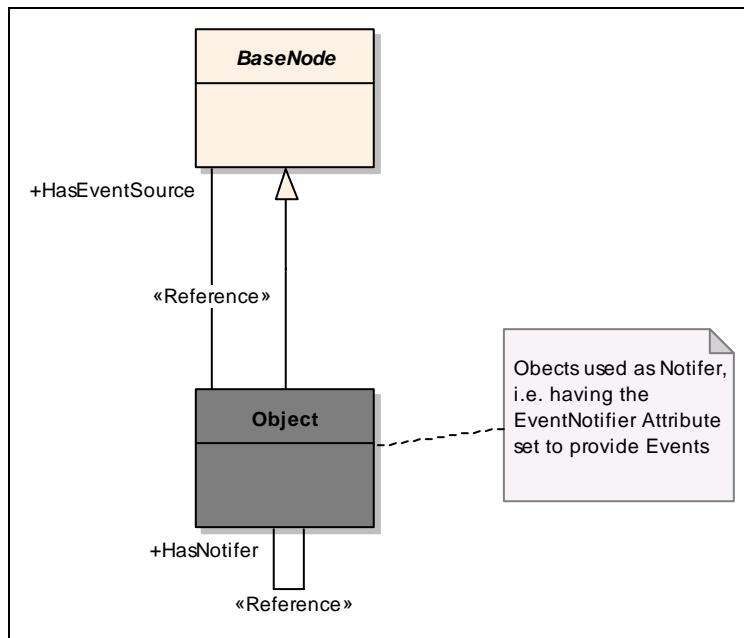


Figure 30 – EventNotifier

### B.3.9 DataType

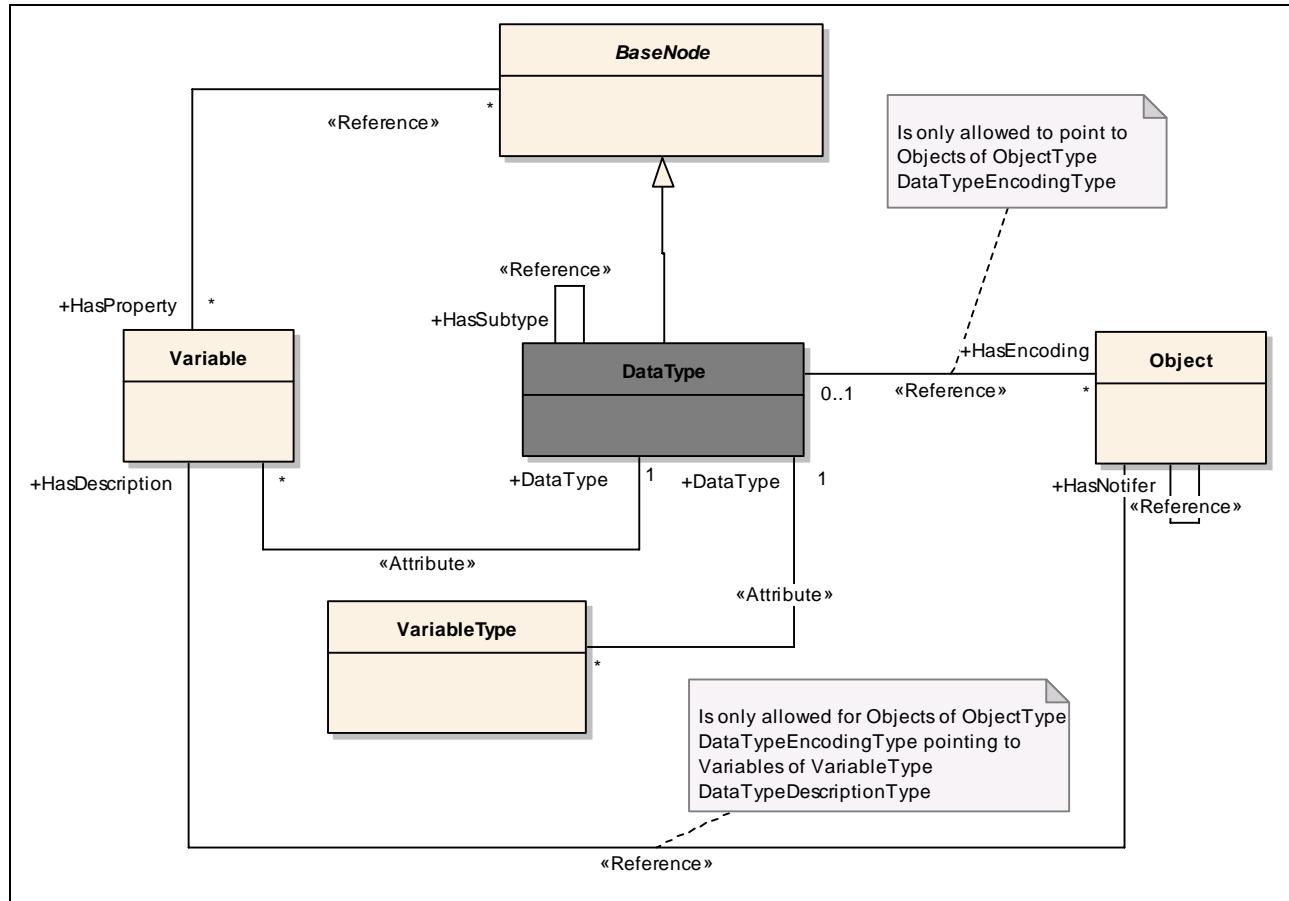


Figure 31 – **DataType**

### B.3.10 View

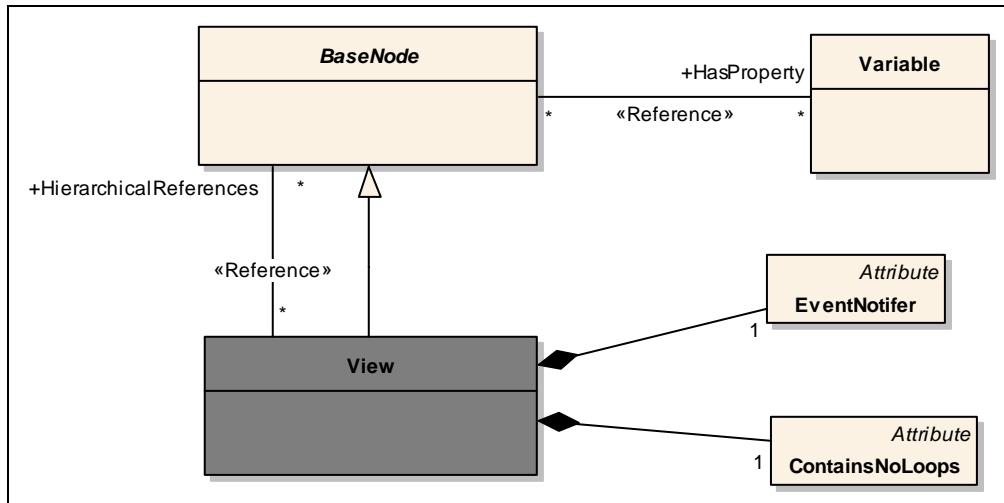


Figure 32 – **View**

## Appendix C: OPC Binary Type Description System

### C.1 Concepts

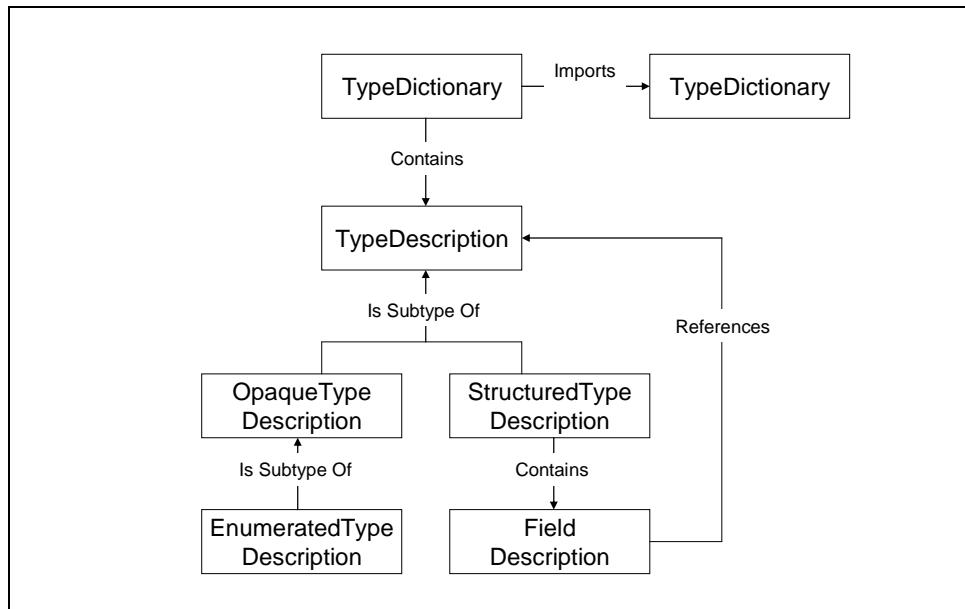
The OPC Binary Schema defines the format of OPC Binary *TypeDictionaries*. Each OPC Binary *TypeDictionary* is an XML document that contains one or more *TypeDescriptions* that describe the format of a binary-encoded value. Applications that have no advance knowledge of a particular binary encoding can use the OPC Binary *TypeDescription* to interpret or construct a value.

The OPC Binary Type Description System does not define a standard mechanism to *encode* data in binary. It only provides a standard way to *describe* an existing binary encoding. Many binary encodings will have a mechanism to describe types that could be encoded; however, these descriptions are useful only to applications that have knowledge of the type description system used with each binary encoding. The OPC Binary Type Description System is a generic syntax that can be used by any application to interpret any binary encoding.

The OPC Binary Type Description System was originally defined in the OPC Complex Data Specification. The OPC Binary Type Description System described in this Annex is quite different and is correctly described as the OPC Binary Type Description System Version 2.0.

Each *TypeDescription* is identified by a *TypeName* which must be unique within the *TypeDictionary* that defines it. Each *TypeDictionary* also has a *TargetNamespace* which should be unique among all OPC Binary *TypeDictionaries*. This mean that the *TypeName* qualified with the *TargetNamespace* for the dictionary should be a globally-unique identifier for a *TypeDescription*.

Figure 33 below illustrates the structure of an OPC Binary *TypeDictionary*.



**Figure 33 – OPC Binary Dictionary Structure**

Each binary encoding is built from a set of opaque building blocks that are either primitive types with a fixed length or variable-length types with a structure that is too complex to describe properly in an XML document. These building blocks are described with an *OpaqueTypeDescription*. An instance of one of these building blocks is a binary-encoded value.

The OPC Binary Type Description System defines a set of standard *OpaqueTypeDescriptions* that all OPC Binary *TypeDictionaries* should use to build their *TypeDescriptions*. These standard type descriptions are described in Clause C.3.

In some cases, the binary encoding described by an *OpaqueTypeDescription* may have a fixed size which would allow an application to skip an encoded value that it does not understand. If that is the case, the *LengthInBits* attribute should be specified for the *OpaqueTypeDescription*. If authors of *TypeDictionaries* need to define new *OpaqueTypeDescriptions* that do not have a fixed size then they should use the documentation elements to describe how to encode binary values for the type. This description should provide enough detail to allow a human to write a program that can interpret instances of the type.

A *StructuredTypeDescription* breaks a complex value into a sequence of values that are described by a *FieldDescription*. Each *FieldDescriptions* has a name, type and a number of qualifiers that specify when the field is used and how many instances of the type exist. A *FieldDescription* is described completely in Clause C.2.6.

An *EnumeratedTypeDescription* describes a numeric value that has a limited set of possible values, each of which has a descriptive name. *EnumeratedTypeDescriptions* provide a convenient way to capture semantic information associated with what would otherwise be an opaque numeric value.

## C.2 Schema Description

### C.2.1 TypeDictionary

The *TypeDictionary* element is the root element of an OPC Binary dictionary. The components of this element are described in Table 20

**Table 20 – TypeDictionary Components**

Name	Type	Description
Documentation	Documentation	An element that contains human-readable text and XML that provides an overview of what is contained in the dictionary.
Import	ImportDirective[]	Zero or more elements that specify other <i>TypeDictionaries</i> that are referenced by <i>StructuredTypeDescriptions</i> defined in the dictionary. Each import element specifies the <i>NamespaceURI</i> of the <i>TypeDictionary</i> being imported. The <i>TypeDictionary</i> element must declare an XML namespace prefix for each imported namespace.
TargetNamespace	xs:string	Specifies the URI that qualifies all <i>TypeDescriptions</i> defined in the dictionary.
DefaultByteOrder	ByteOrder	Specifies the default <i>ByteOrder</i> for all <i>TypeDescriptions</i> that have the <i>ByteOrderSignificant</i> attribute set to “true”. This value overrides the setting in any imported <i>TypeDictionary</i> . This value is overridden by the <i>DefaultByteOrder</i> specified on a <i>TypeDescription</i> .
TypeDescription	TypeDescription[]	One or more elements that describe the structure of a binary encoded value. A <i>TypeDescription</i> is an abstract type. A dictionary may only contain the <i>OpaqueTypeDescription</i> , <i>EnumeratedTypeDescription</i> and <i>StructuredTypeDescription</i> elements.

### C.2.2 TypeDescription

A *TypeDescription* describes the structure of a binary encoded value. A *TypeDescription* is an abstract base type and only instances of sub-types may appear in a *TypeDictionary*. The components of a *TypeDescription* are described in Table 21

**Table 21 – TypeDescription Components**

Name	Type	Description
Documentation	Documentation	An element that contains human readable text and XML that describes the type. This element should capture any semantic information that would help a human understand what is contained in the value.
Name	xs: NCName	An attribute that specifies a name for the <i>TypeDescription</i> that is unique within the dictionary. The fields of structured types reference <i>TypeDescriptions</i> by using this name qualified with the dictionary namespace URI.
DefaultByteOrder	ByteOrder	An attribute that specifies the default <i>ByteOrder</i> for the type description. This value overrides the setting in any <i>TypeDictionary</i> or in any <i>StructuredTypeDescription</i> that references the type description.

### C.2.3 OpaqueTypeDescription

An *OpaqueTypeDescription* describes a binary encoded value that is either a primitive fixed length type or that has a structure too complex to capture in an OPC Binary type dictionary. Authors of type dictionaries should avoid defining *OpaqueTypeDescriptions* that do not have a fixed length because it would prevent applications from interpreting values that use these types without having built-in knowledge of the *OpaqueTypeDescription*. The OPC Binary Type Description System defines many standard *OpaqueTypeDescriptions* that should allow authors to describe most binary encoded values as *StructuredTypeDescriptions*.

The components of an *OpaqueTypeDescription* are described in Table 22.

**Table 22 – OpaqueTypeDescription Components**

Name	Type	Description
TypeDescription	TypeDescription	An <i>OpaqueTypeDescription</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 21.
LengthInBits	xs:string	An attribute which specifies the length of the <i>OpaqueTypeDescription</i> in bits. This value should always be specified. If this value is not specified the <i>Documentation</i> element should describe the encoding in a way that a human understands.
ByteOrderSignificant	xs:boolean	An attribute that indicates whether byte order is significant for the type. If byte order is significant then the application must determine the byte order to use for the current context before interpreting the encoded value. The application determines the byte order by looking for the <i>DefaultByteOrder</i> attribute specified for containing <i>StructuredTypeDescriptions</i> or the <i>TypeDictionary</i> . If <i>StructuredTypeDescriptions</i> are nested the inner <i>StructuredTypeDescriptions</i> override the byte order of the outer descriptions. If the <i>DefaultByteOrder</i> attribute is specified for the <i>OpaqueTypeDescription</i> , then the <i>ByteOrder</i> is fixed and does not change according to context. If this attribute is “true”, then the <i>LengthInBits</i> attribute must be specified and it must be an integer multiple of 8 bits.

### C.2.4 EnumeratedTypeDescription

An *EnumeratedTypeDescription* describes a binary-encoded numeric value that has a fixed set of valid values. The encoded binary value described by an *EnumeratedTypeDescription* is always an unsigned integer with a length specified by the *LengthInBits* attribute.

The names for each of the enumerated values are not required to interpret the binary encoding, however, they form part of the documentation for the type.

The components of an *EnumeratedTypeDescription* are described in Table 23.

**Table 23 – EnumeratedTypeDescription Components**

Name	Type	Description
OpaqueTypeDescription	OpaqueTypeDescription	An <i>EnumeratedTypeDescription</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 21 and for an <i>OpaqueTypeDescription</i> defined in Table 22. The <i>LengthInBits</i> attribute must always be specified.
EnumeratedValue	EnumeratedValue	One or more elements that describe the possible values for the instances of the type.

### C.2.5 StructuredTypeDescription

A *StructuredTypeDescription* describes a type as a sequence of binary-encoded values. Each value in the sequence is called a *Field*. Each *Field* references a *TypeDescription* that describes the binary-encoded value that appears in the field. A *Field* may specify that zero, one or multiple instances of the type appear within the sequence described by the *StructuredTypeDescription*.

Authors of type dictionaries should use *StructuredTypeDescriptions* to describe a variety of common data constructs including arrays, unions and structures.

Some fields have lengths that are not multiples of 8 bits. Several of these fields may appear in a sequence in a structure, however, the total number of bits used in the sequence must be fixed and it must be a multiple of 8 bits. Any field which does not have a fixed length must be aligned on a byte boundary.

A sequence of fields which do not line up on byte boundaries are specified from the least significant bit to the most significant bit. Sequences which are longer than one byte overflow from the most significant bit of the first byte into the least significant bit of the next byte.

The components of a *StructuredTypeDescription* are described in Table 24.

**Table 24 – StructuredTypeDescription Components**

Name	Type	Description
TypeDescription	TypeDescription	A <i>StructuredTypeDescription</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 21.
Field	FieldDescription	One or more elements that describe the fields of the structure. Each field must have a name that is unique within the <i>StructuredTypeDescription</i> . Some fields may reference other fields in the <i>StructuredTypeDescription</i> by using this name.
anyAttribute	*	Authors of a <i>TypeDictionary</i> may add their own attributes to any <i>StructuredTypeDescription</i> that must be qualified with a namespace defined by the author. Applications should not be required to understand these attributes in order to interpret a binary encoded instance of the type.

### C.2.6 FieldDescription

A *FieldDescription* describes a binary encoded value that appears in sequence within a *StructuredTypeDescription*. Every *FieldDescription* must reference a *TypeDescription* that describes the encoded value for the field.

A *FieldDescription* may specify an array of encoded values.

*Fields* may be optional and they reference other *FieldDescriptions*, which indicate if they are present in any specific instance of the type.

The components of a *FieldDescription* are described in Table 25.

**Table 25 – FieldDescription Components**

Name	Type	Description												
Documentation	Documentation	An element that contains human readable text and XML that describes the field. This element should capture any semantic information that would help a human understand what is contained in the field.												
Name	xs:string	An attribute that specifies a name for the <i>Field</i> that is unique within the <i>StructuredTypeDescription</i> . Other fields in the structured type reference a <i>Field</i> by using this name.												
TypeName	xs:QName	An attribute that specifies the <i>TypeDescription</i> that describes the contents of the field. A field may contain zero or more instances of this type depending on the settings for the other attributes and the values in other fields												
Length	xs:unsignedInt	An attribute that indicates length of the field. This value may be the total number of encoded bytes or it may be the number of instances of the type referenced by the field. The <i>IsLengthInBytes</i> attributes specifies which of these definitions applies.												
LengthField	xs:string	An attribute that indicates which other field in the <i>StructuredTypeDescription</i> specifies the length of the field. The length of the field may be in bytes or it may be the number of instances of the type referenced by the field. The <i>IsLengthInBytes</i> attributes specifies which of these definitions applies. If this attribute refers to a field that is not present in an encoded value, then the default value for the length is 1. This situation could occur if the field referenced is an optional field (see the <i>SwitchField</i> attribute). The length field must be a fixed length Base-2 representation of an integer. If the length field is one of the standard signed integer types and the value is a negative integer, then the field is not present in the encoded stream. The <i>FieldDescription</i> referenced by this attribute must precede the field with the <i>StructuredTypeDescription</i> .												
IsLengthInBytes	xs:boolean	An attribute that indicates whether the <i>Length</i> or <i>LengthField</i> attributes specify the length of the field in bytes or in the number of instances of the type referenced by the field.												
SwitchField	xs:string	If this attribute is specified, then the field is optional and many not appear in every instance of the encoded value. This attribute specifies the name of another <i>Field</i> that controls whether this field is present in the encoded value. The field referenced by this attribute must be an integer value (see the <i>LengthField</i> attribute). The current value of the switch field is compared to the <i>SwitchValue</i> attribute using the <i>SwitchOperand</i> . If the condition evaluates to true then the field appears in the stream. If the <i>SwitchValue</i> attribute is not specified, then this field is present if the value of the switch field is non-zero. The <i>SwitchOperand</i> field is ignored if it is present. If the <i>SwitchOperand</i> attribute is missing, then the field is present if the value of the switch field is equal to the value of the <i>SwitchValue</i> attribute. The <i>Field</i> referenced by this attribute must precede the field with the <i>StructuredTypeDescription</i> .												
SwitchValue	xs:unsignedInt	This attribute specifies when the field appears in the encoded value. The value of the field referenced by the <i>SwitchName</i> attribute is compared using the <i>SwitchOperand</i> attribute to this value. The field is present if the expression evaluates to true. The field is not present otherwise.												
SwitchOperand	xs:string	This attribute specifies how the value of the switch field should be compared to the switch value attribute. This field is an enumeration with the following values:  <table> <tr> <td>Equal</td> <td><i>SwitchField</i> is equal to the <i>SwitchValue</i>.</td> </tr> <tr> <td>GreaterThan</td> <td><i>SwitchField</i> is greater than the <i>SwitchValue</i>.</td> </tr> <tr> <td>LessThan</td> <td><i>SwitchField</i> is less than the <i>SwitchValue</i>.</td> </tr> <tr> <td>GreaterThanOrEqual</td> <td><i>SwitchField</i> is greater than or equal to the <i>SwitchValue</i>.</td> </tr> <tr> <td>LessThanOrEqual</td> <td><i>SwitchField</i> is less than or equal to the <i>SwitchValue</i>.</td> </tr> <tr> <td>NotEqual</td> <td><i>SwitchField</i> is not equal to the <i>SwitchValue</i>.</td> </tr> </table> In each case the field is present if the expression is true.	Equal	<i>SwitchField</i> is equal to the <i>SwitchValue</i> .	GreaterThan	<i>SwitchField</i> is greater than the <i>SwitchValue</i> .	LessThan	<i>SwitchField</i> is less than the <i>SwitchValue</i> .	GreaterThanOrEqual	<i>SwitchField</i> is greater than or equal to the <i>SwitchValue</i> .	LessThanOrEqual	<i>SwitchField</i> is less than or equal to the <i>SwitchValue</i> .	NotEqual	<i>SwitchField</i> is not equal to the <i>SwitchValue</i> .
Equal	<i>SwitchField</i> is equal to the <i>SwitchValue</i> .													
GreaterThan	<i>SwitchField</i> is greater than the <i>SwitchValue</i> .													
LessThan	<i>SwitchField</i> is less than the <i>SwitchValue</i> .													
GreaterThanOrEqual	<i>SwitchField</i> is greater than or equal to the <i>SwitchValue</i> .													
LessThanOrEqual	<i>SwitchField</i> is less than or equal to the <i>SwitchValue</i> .													
NotEqual	<i>SwitchField</i> is not equal to the <i>SwitchValue</i> .													
Terminator	xs:hexBinary	This attribute indicates that the field contains one or more instances of <i>TypeDescription</i> referenced by this field and that the last value has the binary												

		<p>encoding specified by the value of this attribute.</p> <p>If this attribute is specified then the <i>TypeDescription</i> referenced by this field must either have a fixed byte order (i.e. byte order is not significant or explicitly specified) or the containing <i>StructuredTypeDescription</i> must explicitly specify the byte order.</p> <p>Examples:</p> <table border="1"> <thead> <tr> <th><u>Field Data Type</u></th><th><u>Terminator</u></th><th><u>Byte Order</u></th><th><u>Hexadecimal String</u></th></tr> </thead> <tbody> <tr> <td>Char</td><td>tab character</td><td>not applicable</td><td>09</td></tr> <tr> <td>WideChar:</td><td>tab character</td><td>BigEndian</td><td>0009</td></tr> <tr> <td>WideChar:</td><td>tab character</td><td>LittleEndian</td><td>0900</td></tr> <tr> <td>Int16</td><td>1</td><td>BigEndian</td><td>0001</td></tr> <tr> <td>Int16</td><td>1</td><td>LittleEndian</td><td>0100</td></tr> </tbody> </table>	<u>Field Data Type</u>	<u>Terminator</u>	<u>Byte Order</u>	<u>Hexadecimal String</u>	Char	tab character	not applicable	09	WideChar:	tab character	BigEndian	0009	WideChar:	tab character	LittleEndian	0900	Int16	1	BigEndian	0001	Int16	1	LittleEndian	0100
<u>Field Data Type</u>	<u>Terminator</u>	<u>Byte Order</u>	<u>Hexadecimal String</u>																							
Char	tab character	not applicable	09																							
WideChar:	tab character	BigEndian	0009																							
WideChar:	tab character	LittleEndian	0900																							
Int16	1	BigEndian	0001																							
Int16	1	LittleEndian	0100																							
anyAttribute	*	Authors of a <i>TypeDictionary</i> may add their own attributes to any <i>FieldDescription</i> which must be qualified with a namespace defined by the authors. Applications should not be required to understand these attributes in order to interpret a binary encoded field value.																								

### C.2.7 EnumeratedValueDescription

An *EnumeratedValueDescription* describes a possible value for an *EnumeratedTypeDescription*.

The components of an *EnumeratedValue* are described in Table 26.

**Table 26 – EnumeratedValueDescription Components**

Name	Type	Description
Name	xs:string	This attribute specifies a descriptive name for the enumerated value.
Value	xs:unsignedInt	This attribute specifies the numeric value that could appear in the binary encoding.

### C.2.8 ByteOrder

A *ByteOrder* is an enumeration that describes a possible value byte orders for *TypeDescriptions* that allow different byte orders to be used. There are two possible values: BigEndian and LittleEndian. BigEndian indicates the most significant byte appears first in the binary encoding. LittleEndian indicates that the least significant byte appears first.

### C.2.9 ImportDirective

An *ImportDirective* specifies a *TypeDictionary* that is referenced by *FiledDescriptions* defined in the current dictionary.

The components of an *ImportDirective* are described in Table 27.

**Table 27 – ImportDirective Components**

Name	Type	Description
Namespace	xs:string	This attribute specifies the <i>TargetNamespace</i> for the <i>TypeDictionary</i> being imported. This may be a well-known URI which means applications need not have access to the physical file to recognise types that are referenced.
Location	xs:string	This attribute specifies the physical location of the XML file containing the <i>TypeDictionary</i> to import. This value could be a URL for a network resource, a NodeId in a UA server address space or a local file path.

## C.3 Standard Type Descriptions

The OPC Binary Type Description System defines a number of standard type descriptions that can be used to describe many common binary encodings using a *StructuredTypeDescription*. The standard type descriptions are described in

**Table 28 – Standard Type Descriptions**

Type Name	Description
Bit	A single bit value.
Boolean	A two-state logical value represented as an 8-bit value.
SByte	An 8-bit signed integer.
Byte	An 8-bit unsigned integer.
Int16	A 16-bit signed integer.
UInt16	A 16-bit unsigned integer.
Int32	A 32-bit signed integer.
UInt32	A 32-bit unsigned integer.
Int64	A 64-bit signed integer.
UInt64	A 64-bit unsigned integer.
Float	An IEEE-754 single precision floating point value.
Double	An IEEE-754 double precision floating point value.
Char	An 8-bit UTF-8 character value.
WideChar	A 16-bit UTF-16 character value.
String	A null terminated sequence of UTF-8 characters.
CharArray	A sequence of UTF-8 characters preceded by the number of characters.
WideString	A null terminated sequence of UTF-16 characters.
WideCharArray	A sequence of UTF-16 characters preceded by the number of characters.
DateTime	A 64-bit signed integer representing the number of 100 nanoseconds intervals since 1601-01-01 00:00:00. This is the same as the WIN32 FILETIME type.
ByteString	A sequence of bytes preceded by its length in bytes.
Guid	A 128-bit structured type that represents a WIN32 GUID value.

#### C.4 Type Description Examples

1. A 128-bit signed integer.

```
<opc:OpaqueTypeDescription Name="Int128" LengthInBits="128">
  <opc:Documentation>A 128-bit signed integer.</opc:Documentation>
</opc:OpaqueTypeDescription>
```

2. A 16-bit value divided into several fields.

```
<opc:StructuredTypeDescription Name="Quality">
  <opc:Documentation>An OPC COM-DA quality value.</opc:Documentation>
  <opc:Field Name="LimitBits" TypeName="opc:Bit" Length="2" />
  <opc:Field Name="QualityBits" TypeName="opc:Bit" Length="6" />
  <opc:Field Name="VendorBits" TypeName="opc:Byte" />
</opc:StructuredTypeDescription>
```

When using bit fields, the least significant bits within a byte must appear first.

3. A structured type with optional fields.

```
<opc:StructuredTypeDescription Name="DataValue">
  <opc:Documentation>A value with an associated timestamp, and quality.</opc:Documentation>
  <opc:Field Name="ValueSpecified" TypeName="Bit" />
  <opc:Field Name="StatusCodeSpecified" TypeName="Bit" />
  <opc:Field Name="TimestampSpecified" TypeName="Bit" />
  <opc:Field Name="Reserved1" TypeName="Bit" Length="5" />
  <opc:Field Name="Value" TypeName="Variant" SwitchField="ValueSpecified" />
  <opc:Field Name="Quality" TypeName="Quality" SwitchField="StatusCodeSpecified" />
  <opc:Field Name="Timestamp" TypeName="opc:DateTime" SwitchField="SourceTimestampSpecified" />
</opc:StructuredTypeDescription>
```

It is necessary to explicitly specify any padding bits required to ensure subsequent fields line up on byte boundaries.

4. An array of integers.

```
<opc:StructuredTypeDescription Name="IntegerArray">
  <opc:Documentation>An array of integers prefixed by its length.</opc:Documentation>
  <opc:Field Name="Size" TypeName="opc:Int32" />
```

```
<opc:Field Name="Array" TypeName="opc:Int32" LengthField="Size" />
</opc:StructuredTypeDescription>
```

Nothing is encoded for the Array field if the Size field has a value <= 0.

## 5. An array of integers with a terminator instead of a length prefix.

```
<opc:StructuredTypeDescription Name="IntegerArray" DefaultByteOrder="LittleEndian">
  <opc:Documentation>An array of integers terminated with a known value.</opc:Documentation>
  <opc:Field Name="Value" TypeName="opc:Int16" Terminator="FF7F" />
</opc:StructuredTypeDescription>
```

The terminator is 32,767 converted to hexadecimal with LittleEndian byte order.

## 6. A simple union.

```
<opc:StructuredTypeDescription Name="Variant">
  <opc:Documentation>A union of several types.</opc:Documentation>
  <opc:Field Name="ArrayLengthSpecified" TypeName="opc:Bit" Length="1" />
  <opc:Field Name="VariantType" TypeName="opc:Bit" Length="7" />
  <opc:Field Name="ArrayLength" TypeName="opc:Int32"
    SwitchField="ArrayLengthSpecified" />
  <opc:Field Name="Int32" TypeName="opc:Int32" LengthField="ArrayLength"
    SwitchField="VariantType" SwitchValue="1" />
  <opc:Field Name="String" TypeName="opc:String" LengthField="ArrayLength"
    SwitchField="VariantType" SwitchValue="2" />
  <opc:Field Name="DateTime" TypeName="opc:DateTime" LengthField="ArrayLength"
    SwitchField="VariantType" SwitchValue="3" />
</opc:StructuredTypeDescription>
```

The *ArrayLength* field is optional. If it is not present in an encoded value, then the length of all fields with *LengthField* set to "ArrayLength" have a length of 1.

It is valid for the the *VariantType* field to have a value that has no matching field defined. This simply means all optional fields are not present in the encoded value.

## 7. An enumerated type.

```
<opc:EnumeratedTypeDescription Name="TrafficLight" LengthInBits="32">
  <opc:Documentation>The possible colours for a traffic signal.</opc:Documentation>
  <opc:EnumeratedValue Name="Red" Value="4">
    <opc:Documentation>Red says stop immediately.</opc:Documentation>
  </opc:EnumeratedValue>
  <opc:EnumeratedValue Name="Yellow" Value="3">
    <opc:Documentation>Yellow says prepare to stop.</opc:Documentation>
  </opc:EnumeratedValue>
  <opc:EnumeratedValue Name="Green" Value="2">
    <opc:Documentation>Green says you may proceed.</opc:Documentation>
  </opc:EnumeratedValue>
</opc:EnumeratedTypeDescription>
```

The documentation element is used to provide human readable description of the type and values.

## 8. A nullable array.

```
<opc:StructuredTypeDescription Name="NullableArray">
  <opc:Documentation>An array where a length of -1 means null.</opc:Documentation>
  <opc:Field Name="Length" TypeName="opc:Int32" />
  <opc:Field
    Name="Int32"
    TypeName="opc:Int32"
    LengthField="Length"
    SwitchField="Length"
    SwitchValue="0"
    SwitchOperand="GreaterThanOrEqual" />
</opc:StructuredTypeDescription>
```

If the length of the array is -1 then the array does not appear in the stream.

## C.5 OPC Binary XML Schema

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<xs:schema
  targetNamespace="http://opcfoundation.org/BinarySchema/"
  elementFormDefault="qualified"
  xmlns="http://opcfoundation.org/BinarySchema/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
<xs:element name="Documentation">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:any minOccurs="0" maxOccurs="unbounded"/>
    </xs:choice>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>

<xs:complexType name="ImportDirective">
  <xs:attribute name="Namespace" type="xs:string" use="optional" />
  <xs:attribute name="Location" type="xs:string" use="optional" />
</xs:complexType>

<xs:simpleType name="ByteOrder">
  <xs:restriction base="xs:string">
    <xs:enumeration value="BigEndian" />
    <xs:enumeration value="LittleEndian" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="TypeDescription">
  <xs:sequence>
    <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:NCName" use="required" />
  <xs:attribute name="DefaultByteOrder" type="ByteOrder" use="optional" />
</xs:complexType>

<xs:complexType name="OpaqueTypeDescription">
  <xs:complexContent>
    <xs:extension base="TypeDescription">
      <xs:attribute name="LengthInBits" type="xs:int" use="optional" />
      <xs:attribute name="ByteOrderSignificant" type="xs:boolean" default="false" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="EnumeratedValueDescription">
  <xs:sequence>
    <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="optional" />
  <xs:attribute name="Value" type="xs:unsignedInt" use="optional" />
</xs:complexType>

<xs:complexType name="EnumeratedTypeDescription">
  <xs:complexContent>
    <xs:extension base="OpaqueTypeDescription">
      <xs:sequence>
        <xs:element name="EnumeratedValue" type="EnumeratedValueDescription" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="SwitchOperand">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Equals" />
    <xs:enumeration value="GreaterThan" />
    <xs:enumeration value="LessThan" />
    <xs:enumeration value="GreaterThanOrEqual" />
    <xs:enumeration value="LessThanOrEqual" />
    <xs:enumeration value="NotEqual" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="FieldDescription">
  <xs:sequence>
    <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required" />
```

```

<xs:attribute name="TypeName" type="xs:QName" use="optional" />
<xs:attribute name="Length" type="xs:unsignedInt" use="optional" />
<xs:attribute name="LengthField" type="xs:string" use="optional" />
<xs:attribute name="IsLengthInBytes" type="xs:boolean" default="false" />
<xs:attribute name="SwitchField" type="xs:string" use="optional" />
<xs:attribute name="SwitchValue" type="xs:unsignedInt" use="optional" />
<xs:attribute name="SwitchOperand" type="SwitchOperand" use="optional" />
<xs:attribute name="Terminator" type="xs:hexBinary" use="optional" />
<xs:anyAttribute processContents="lax" />
</xs:complexType>

<xs:complexType name="StructuredTypeDescription">
  <xs:complexContent>
    <xs:extension base="TypeDescription">
      <xs:sequence>
        <xs:element name="Field" type="FieldDescription" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:anyAttribute processContents="lax" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="TypeDictionary">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
      <xs:element name="Import" type="ImportDirective" minOccurs="0" maxOccurs="unbounded" />
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="OpaqueTypeDescription" type="OpaqueTypeDescription" />
        <xs:element name="EnumeratedTypeDescription" type="EnumeratedTypeDescription" />
        <xs:element name="StructuredTypeDescription" type="StructuredTypeDescription" />
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="TargetNamespace" type="xs:string" use="required" />
    <xs:attribute name="DefaultByteOrder" type="ByteOrder" use="optional" />
  </xs:complexType>
</xs:element>
</xs:schema>

```

## C.6 OPC Binary Standard TypeDictionary

```

<?xml version="1.0" encoding="utf-8"?>
<opc:TypeDictionary
  xmlns="http://opcfoundation.org/BinarySchema/"
  xmlns:opc="http://opcfoundation.org/BinarySchema/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  TargetNamespace="http://opcfoundation.org/BinarySchema/">
</
  <opc:Documentation>This dictionary defines the standard types used by the OPC Binary type
description system.</opc:Documentation>

  <opc:OpaqueTypeDescription Name="Bit" LengthInBits="1">
    <opc:Documentation>A single bit.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="Boolean" LengthInBits="8">
    <opc:Documentation>A two state logical value represented as a 8-bit value.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="SByte" LengthInBits="8">
    <opc:Documentation>An 8-bit signed integer.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="Byte" LengthInBits="8">
    <opc:Documentation>A 8-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="Int16" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit signed integer.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="UInt16" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueTypeDescription>

  <opc:OpaqueTypeDescription Name="Int32" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 32-bit signed integer.</opc:Documentation>
  </opc:OpaqueTypeDescription>

```

```
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="UInt32" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 32-bit unsigned integer.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="Int64" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 64-bit signed integer.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="UInt64" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>A 64-bit unsigned integer.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="Float" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>An IEEE-754 single precision floating point value.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="Double" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>An IEEE-754 double precision floating point value.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:OpaqueTypeDescription Name="Char" LengthInBits="8">
    <opc:Documentation>A 8-bit character value.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:StructuredTypeDescription Name="String">
    <opc:Documentation>A UTF-8 null terminated string value.</opc:Documentation>
    <opc:Field Name="Value" TypeName="Char" Terminator="00" />
</opc:StructuredTypeDescription>

<opc:StructuredTypeDescription Name="CharArray">
    <opc:Documentation>A UTF-8 string prefixed by its length in characters.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="Char" LengthField="Length" />
</opc:StructuredTypeDescription>

<opc:OpaqueTypeDescription Name="WideChar" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit character value.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:StructuredTypeDescription Name="WideString">
    <opc:Documentation>A UTF-16 null terminated string value.</opc:Documentation>
    <opc:Field Name="Value" TypeName="WideChar" Terminator="0000" />
</opc:StructuredTypeDescription>

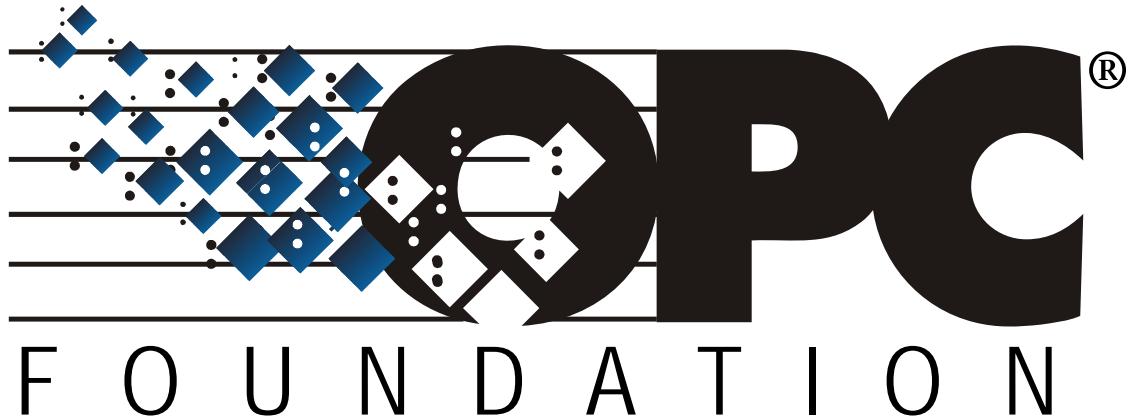
<opc:StructuredTypeDescription Name="WideCharArray">
    <opc:Documentation>A UTF-16 string prefixed by its length in characters.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="WideChar" LengthField="Length" />
</opc:StructuredTypeDescription>

<opc:StructuredTypeDescription Name="ByteString">
    <opc:Documentation>An array of bytes prefixed by its length.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="Byte" LengthField="Length" />
</opc:StructuredTypeDescription>

<opc:OpaqueTypeDescription Name="DateTime" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>The number of 100 nanosecond intervals since January 01, 1601.</opc:Documentation>
</opc:OpaqueTypeDescription>

<opc:StructuredTypeDescription Name="Guid">
    <opc:Documentation>A 128-bit globally unique identifier.</opc:Documentation>
    <opc:Field Name="Data1" TypeName="UInt32" />
    <opc:Field Name="Data2" TypeName="UInt16" />
    <opc:Field Name="Data3" TypeName="UInt16" />
    <opc:Field Name="Data4" TypeName="Byte" Length="8" />
</opc:StructuredTypeDescription>

</opc>TypeDictionary>
```



# **OPC Unified Architecture**

## **Specification**

### **Part 4: Services**

**Version 1.01.05**

**February 01, 2007**



Specification Type	Industry Standard Specification	Comments:
Title:	OPC Unified Architecture Part 4 Services	Date: February 01, 2007
Version:	Draft 1.01.05	Software Source: MS-Word OPC UA Part 4 - Services Draft 1.01.05 Specification.doc
Author:	OPC Foundation	Status: Draft

## CONTENTS

	Page
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, and conventions.....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 2 terms .....	2
3.3 OPC UA Part 3 terms .....	2
3.4 OPC UA Services terms .....	3
3.4.1 Deadband .....	3
3.4.2 Endpoint .....	3
3.4.3 ServerUri .....	3
3.4.4 SoftwareCertificate.....	3
3.5 Abbreviations and symbols .....	3
3.6 Conventions for Service definitions.....	3
4 Overview .....	4
4.1 Service Set model.....	4
4.2 Request/response Service procedures.....	8
5 Service Sets .....	9
5.1 General .....	9
5.2 Service request and response header.....	9
5.3 Service results .....	9
5.4 Discovery Service Set .....	10
5.4.1 Overview.....	10
5.4.2 FindServers.....	10
5.4.3 GetEndpoints .....	11
5.4.4 RegisterServer .....	13
5.5 SecureChannel Service Set.....	15
5.5.1 Overview.....	15
5.5.2 OpenSecureChannel .....	17
5.5.3 CloseSecureChannel .....	19
5.6 Session Service Set.....	20
5.6.1 Overview.....	20
5.6.2 CreateSession.....	20
5.6.3 ActivateSession.....	23
5.6.4 CloseSession .....	26
5.6.5 Cancel .....	26
5.7 NodeManagement Service Set .....	27
5.7.1 Overview.....	27
5.7.2 AddNodes .....	27
5.7.3 AddReferences.....	29
5.7.4 DeleteNodes .....	31
5.7.5 DeleteReferences .....	32
5.8 View Service Set.....	33
5.8.1 Overview .....	33
5.8.2 BrowseProperties .....	33
5.8.3 Browse.....	34
5.8.4 BrowseNext.....	36

5.8.5	TranslateBrowsePathsToNodeIds .....	37
5.9	Query Service Set.....	38
5.9.1	Overview.....	38
5.9.2	Querying Views .....	39
5.9.3	QueryFirst .....	39
5.9.4	QueryNext.....	43
5.9.5	Example for Queries and ContentFilters .....	44
5.10	Attribute Service Set .....	58
5.10.1	Overview.....	58
5.10.2	Read.....	58
5.10.3	HistoryRead .....	60
5.10.4	Write .....	62
5.10.5	HistoryUpdate .....	64
5.11	Method Service Set.....	65
5.11.1	Overview.....	65
5.11.2	Call .....	65
5.12	MonitoredItem Service Set .....	66
5.12.1	MonitoredItem model .....	66
5.12.2	CreateMonitoredItems .....	70
5.12.3	ModifyMonitoredItems .....	72
5.12.4	SetMonitoringMode .....	74
5.12.5	SetTriggering .....	74
5.12.6	DeleteMonitoredItems.....	76
5.13	Subscription Service Set .....	77
5.13.1	Subscription model.....	77
5.13.2	CreateSubscription .....	84
5.13.3	ModifySubscription .....	85
5.13.4	SetPublishingMode.....	86
5.13.5	Publish.....	87
5.13.6	Republish.....	89
5.13.7	TransferSubscriptions.....	90
5.13.8	DeleteSubscriptions.....	91
6	Service behaviours .....	93
6.1	Security .....	93
6.1.1	Overview.....	93
6.1.2	Obtaining and Installing an Application Instance Certificate .....	93
6.1.3	Obtaining and Installing an Software Certificate .....	95
6.1.4	Creating a SecureChannel.....	96
6.1.5	Creating a Session .....	97
6.1.6	Impersonating a User .....	98
6.2	UA Auditing .....	98
6.2.1	Overview.....	98
6.2.2	General audit logs .....	98
6.2.3	General audit Events .....	99
6.2.4	Auditing for SecureChannel Service Set.....	99
6.2.5	Auditing for Session Service Set.....	99
6.2.6	Auditing for NodeManagement Service Set .....	99
6.2.7	Auditing for Attribute Service Set .....	100
6.2.8	Auditing for Method Service Set.....	100

6.2.9	Auditing for View, Query, MonitoredItem and Subscription Service Set .....	100
6.3	Redundancy.....	101
6.3.1	Redundancy overview.....	101
6.3.2	Server redundancy overview.....	101
6.3.3	Client redundancy .....	103
7	Common parameter type definitions .....	104
7.1	BuildInfo .....	104
7.2	ContentFilter.....	104
7.3	Counter .....	107
7.4	DataValue.....	107
7.5	DiagnosticInfo.....	109
7.6	Duration.....	110
7.7	ExpandedNodeID .....	110
7.8	Index .....	110
7.9	IntegerID.....	110
7.10	MessageSecurityMode .....	110
7.11	MonitoringAttributes .....	111
7.12	MonitoringMode .....	111
7.13	NotificationMessage.....	112
7.14	NumericRange .....	112
7.15	QueryDataSet .....	112
7.16	ReadValueID.....	113
7.17	ReferenceDescription.....	114
7.18	RelativePath .....	115
7.19	RequestHeader.....	116
7.20	ResponseHeader .....	117
7.21	SecurityHeader .....	117
7.22	SecurityPolicy .....	118
7.23	ServerDescription .....	118
7.24	ServiceFault .....	118
7.25	SignatureData.....	119
7.26	SignedSoftwareCertificate .....	119
7.27	SoftwareCertificate.....	119
7.28	StatusCodes .....	120
7.29	TimestampsToReturn .....	125
7.30	UserTokenPolicy .....	125
7.31	ViewDescription .....	126
8	Extensible parameter type definition.....	126
8.1	Overview .....	126
8.2	ExtensibleParameter .....	126
8.3	MonitoringFilter parameters.....	126
8.3.1	Overview .....	126
8.3.2	DataChangeFilter .....	127
8.3.3	EventFilter .....	127
8.4	FilterOperand parameters .....	128
8.4.1	Overview .....	128
8.4.2	ElementOperand .....	128
8.4.3	LiteralOperand .....	128
8.4.4	AttributeOperand.....	129

8.4.5	PropertyOperand .....	129
8.5	NotificationData parameters .....	129
8.5.1	Overview .....	129
8.5.2	DataChangeNotification parameter .....	130
8.5.3	EventNotification parameter .....	130
8.6	NodeAttributes parameters .....	131
8.6.1	Overview .....	131
8.6.2	ObjectAttributes parameter .....	131
8.6.3	VariableAttributes parameter .....	131
8.6.4	MethodAttributes parameter .....	132
8.6.5	ObjectTypeAttributes parameter .....	132
8.6.6	VariableTypeAttributes parameter .....	132
8.6.7	ReferenceTypeAttributes parameter .....	132
8.6.8	DataTypeAttributes parameter .....	133
8.6.9	ViewAttributes parameter .....	133
8.7	UserIdentityToken parameters .....	133
8.7.1	Overview .....	133
8.7.2	UserName identity tokens .....	133
8.7.3	X509v3 identity tokens .....	134
8.7.4	WSS identity tokens .....	134
Appendix A	: BNF definitions .....	135
A.1	Overview over BNF .....	135
A.2	BNF of RelativePath .....	135
A.3	BNF of NumericRange .....	135

## FIGURES

Figure 1 – Discovery Service Set.....	4
Figure 2 – SecureChannel Service Set .....	5
Figure 3 – Session Service Set.....	5
Figure 4 – NodeManagement Service Set .....	5
Figure 5 – View Service Set .....	6
Figure 6 – Attribute Service Set.....	6
Figure 7 – Method Service Set.....	7
Figure 8 – MonitoredItem and Subscription Service Sets .....	7
Figure 9 – The Discovery Process .....	10
Figure 10 – The Registration Process – Manually Launched Servers .....	14
Figure 11 – The Registration Process – Automatically Launched Servers .....	14
Figure 12 – SecureChannel and Session Services .....	16
Figure 13 – Multiplexing Users on a Session.....	21
Figure 14 – Example Type Nodes .....	45
Figure 15 - Example Instance Nodes .....	46
Figure 16 - Example 1 Filter .....	47
Figure 17 – Example 2 Filter Logic Tree .....	48
Figure 18 – Example 3 Filter Logic Tree .....	50
Figure 19 – Example 4 Filter Logic Tree .....	51
Figure 20 – Example 5 Filter Logic Tree .....	52
Figure 21 – Example 6 Filter Logic Tree .....	53
Figure 22 – Example 7 Filter Logic Tree .....	55
Figure 23 – Example 8 Filter Logic Tree .....	56
Figure 24 – Example 9 Filter Logic Tree .....	57
Figure 25 – MonitoredItem Model .....	67
Figure 26 – Typical delay in change detection.....	68
Figure 27 – Triggering Model.....	70
Figure 28 – Obtaining and Installing an Application Instance Certificate .....	94
Figure 29 – Obtaining and Installing a Software Certificate.....	95
Figure 30 – Establishing a SecureChannel.....	96
Figure 31 – Establishing a Session .....	97
Figure 32 – Impersonating a User.....	98
Figure 33 – Transparent Redundancy setup.....	101
Figure 34 – Non-Transparent Redundancy setup .....	102
Figure 35 – Redundancy mode .....	103
Figure 36 – Filter Logic Tree Example .....	106
Figure 37 – Filter Logic Tree Example .....	106

**TABLES**

Table 1 – Service Definition Table .....	3
Table 2 – Parameter Types defined in [UA Part 3].....	4
Table 3 – FindServers Service Parameters .....	11
Table 4 – GetEndpoints Service Parameters .....	13
Table 5 – RegisterServers Service Parameters .....	15
Table 6 – RegisterServers Service Result Codes .....	15
Table 7 – OpenSecureChannel Service Parameters .....	18
Table 8 – OpenSecureChannel Service Result Codes .....	19
Table 9 – CloseSecureChannel Service Parameters.....	19
Table 10 – CloseSecureChannel Service Result Codes.....	19
Table 11 – CreateSession Service Parameters .....	22
Table 12 – CreateSession Service Result Codes.....	23
Table 13 – ActivateSession Service Parameters .....	25
Table 14 – ActivateSession Service Result Codes.....	26
Table 15 – CloseSession Service Parameters.....	26
Table 16 – CloseSession Service Result Codes .....	26
Table 17 – Cancel Service Parameters .....	27
Table 18 – Cancel Service Result Codes .....	27
Table 19 – AddNodes Service Parameters .....	28
Table 20 – AddNodes Service Result Codes .....	28
Table 21 – AddNodes Operation Level Result Codes .....	29
Table 22 – AddReferences Service Parameters .....	30
Table 23 – AddReferences Service Result Codes .....	30
Table 24 – AddReferences Operation Level Result Codes.....	30
Table 25 – DeleteNodes Service Parameters .....	31
Table 26 – DeleteNodes Service Result Codes .....	31
Table 27 – DeleteNodes Operation Level Result Codes .....	32
Table 28 – DeleteReferences Service Parameters .....	32
Table 29 – DeleteReferences Service Result Codes.....	33
Table 30 – DeleteReferences Operation Level Result Codes .....	33
Table 31 – BrowseProperties Service Parameters .....	34
Table 32 – BrowseProperties Service Result Codes .....	34
Table 33 – BrowseProperties Operation Level Result Codes .....	34
Table 34 – Browse Service Parameters .....	35
Table 35 – Browse Service Result Codes.....	36
Table 36 – BrowseNext Service Parameters .....	36
Table 37 – BrowseNext Service Result Codes.....	37
Table 38 – TranslateBrowsePathsToNodelds Service Parameters .....	38
Table 39 – TranslateBrowsePathsToNodelds Service Result Codes .....	38
Table 40 – TranslateBrowsePathsToNodelds Operation Level Result Codes .....	38
Table 41 – QueryFirst Request Parameters .....	41
Table 42 – QueryFirst Response Parameters .....	42

Table 43 – QueryFirst Service Result Codes .....	42
Table 44 – QueryNext Service Parameters .....	43
Table 45 – QueryNext Service Result Codes.....	43
Table 46 – Example 1 NodeTypeDescription .....	47
Table 47 – Example 1 ContentFilter .....	47
Table 48 – Example 1 QueryDataSets .....	47
Table 49 – Example 2 NodeTypeDescription .....	48
Table 50 – Example 2 ContentFilter .....	48
Table 51 – Example 2 QueryDataSets .....	49
Table 52 – Example 3 - NodeTypeDescriptions .....	49
Table 53 – Example 3 ContentFilter .....	50
Table 54 – Example 3 QueryDataSets .....	50
Table 55 – Example 4 NodeTypeDescription .....	50
Table 56 – Example 4 ContentFilter .....	51
Table 57 – Example 4 QueryDataSets .....	51
Table 58 – Example 5 NodeTypeDescription .....	51
Table 59 – Example 5 ContentFilter .....	52
Table 60 – Example 5 QueryDataSets .....	52
Table 61 – Example 6 NodeTypeDescription .....	53
Table 62 – Example 6 ContentFilter .....	53
Table 63 – Example 6 QueryDataSets .....	53
Table 64 – Example 6 QueryDataSets without Additional Information .....	54
Table 65 – Example 7 NodeTypeDescription .....	54
Table 66 – Example 7 ContentFilter .....	55
Table 67 – Example 7 QueryDataSets .....	55
Table 68 – Example 8 NodeTypeDescription .....	56
Table 69 – Example 8 ContentFilter .....	56
Table 70 – Example 8 QueryDataSets .....	56
Table 71 – Example 9 NodeTypeDescription .....	57
Table 72 – Example 9 ContentFilter .....	57
Table 73 – Example 9 QueryDataSets .....	58
Table 74 – Read Service Parameters.....	59
Table 75 – Read Service Result Codes .....	59
Table 76 – Read Operation Level Result Codes .....	60
Table 77 – HistoryRead ServiceParameters .....	61
Table 78 – HistoryRead Service Result Codes .....	62
Table 79 – HistoryRead Operation Level Result Codes .....	62
Table 80 – Write Service Parameters.....	63
Table 81 – Write Service Result Codes .....	63
Table 82 – Write Operation Level Result Codes .....	64
Table 83 – HistoryUpdate Service Parameters .....	64
Table 84 – HistoryUpdate Service Result Codes .....	65
Table 85 – HistoryUpdate Operation Level Result Codes .....	65

Table 86 – Call Service Parameters .....	66
Table 87 – Call Service Result Codes .....	66
Table 88 – CreateMonitoredItems Service Parameters .....	71
Table 89 – CreateMonitoredItems Service Result Codes .....	72
Table 90 – CreateMonitoredItems Operation Level Result Codes .....	72
Table 91 – ModifyMonitoredItems Service Parameters .....	73
Table 92 – ModifyMonitoredItems Service Result Codes .....	73
Table 93 – ModifyMonitoredItems Operation Level Result Codes .....	73
Table 94 – SetMonitoringMode Service Parameters .....	74
Table 95 – SetMonitoringMode Service Result Codes .....	74
Table 96 – SetMonitoringMode Operation Level Result Codes .....	74
Table 97 – SetTriggering Service Parameters .....	75
Table 98 – SetTriggering Service Result Codes .....	75
Table 99 – SetTriggering Operation Level Result Codes .....	75
Table 100 – DeleteMonitoredItems Service Parameters .....	76
Table 101 – DeleteMonitoredItems Service Result Codes .....	76
Table 102 – DeleteMonitoredItems Operation Level Result Codes .....	76
Table 103 – Subscription States .....	79
Table 104 – Subscription State Table .....	80
Table 105 – State variables and parameters .....	82
Table 106 – Functions .....	83
Table 107 – CreateSubscription Service Parameters .....	84
Table 108 – CreateSubscription Service Result Codes .....	85
Table 109 – ModifySubscription Service Parameters .....	85
Table 110 – ModifySubscription Service Result Codes .....	86
Table 111 – SetPublishingMode Service Parameters .....	86
Table 112 – SetPublishingMode Service Result Codes .....	86
Table 113 – SetPublishingMode Operation Level Result Codes .....	86
Table 114 – Publish Service Parameters .....	88
Table 115 – Publish Service Result Codes .....	88
Table 116 – Publish Operation Level Result Codes .....	88
Table 117 – Republish Service Parameters .....	89
Table 118 – Republish Service Result Codes .....	89
Table 119 – TransferSubscriptions Service Parameters .....	90
Table 120 – TransferSubscriptions Service Result Codes .....	90
Table 121 – TransferSubscriptions Operation Level Result Codes .....	91
Table 122 – DeleteSubscriptions Service Parameters .....	91
Table 123 – DeleteSubscriptions Service Result Codes .....	91
Table 124 – DeleteSubscriptions Operation Level Result Codes .....	92
Table 125 – Redundancy failover actions .....	102
Table 126 – BuildInfo Structure .....	104
Table 127 – ContentFilter Structure .....	104
Table 128 – FilterOperator Definition .....	105

Table 129 – Wildcard characters .....	105
Table 130 – ContentFilter Example .....	106
Table 131 – ContentFilter Example .....	107
Table 132 – Casting rules.....	107
Table 133 – DataValue.....	107
Table 134 – DiagnosticInfo .....	109
Table 135 – ExpandedNodeId.....	110
Table 136 – MessageSecurityMode Values .....	110
Table 137 – MonitoringAttributes .....	111
Table 138 – MonitoringMode Values .....	111
Table 139 – NotificationMessage .....	112
Table 140 – NumericRange .....	112
Table 141 – QueryDataSet .....	112
Table 142 – ReadValueId .....	113
Table 143 – ReferenceDescription .....	114
Table 144 – RelativePath .....	115
Table 145 – RelativePath Examples .....	115
Table 146 – RequestHeader.....	116
Table 147 – ResponseHeader .....	117
Table 148 – SecurityPolicy .....	118
Table 149 – ServerDescription.....	118
Table 150 – ServiceFault.....	118
Table 151 – SignatureData .....	119
Table 152 – SignedSoftwareCertificate .....	119
Table 153 – SoftwareCertificate.....	119
Table 154 – StatusCode Bit Assignments .....	121
Table 155 – DataValue InfoBits .....	122
Table 156 – Common Service Result Codes .....	123
Table 157 – Common Operation Level Result Codes.....	124
Table 158 – TimestampsToReturn Values.....	125
Table 159 – UserTokenPolicy .....	125
Table 160 – ViewDescription .....	126
Table 161 – ExtensibleParameter Base Type .....	126
Table 162 – MonitoringFilter parameterTypelds .....	126
Table 163 – DataChangeFilter .....	127
Table 164 – EventFilter structure.....	128
Table 165 – FilterOperand parameterTypelds .....	128
Table 166 – ElementOperand .....	128
Table 167 – LiteralOperand .....	129
Table 168 – AttributeOperand.....	129
Table 169 – PropertyOperand.....	129
Table 170 – NotificationData parameterTypelds .....	130
Table 171 – DataChangeNotification.....	130

Table 172 –EventNotification .....	131
Table 173 – NodeAttributes parameterTypeIds.....	131
Table 174 – ObjectAttributes .....	131
Table 175 – VariableAttributes.....	131
Table 176 – MethodAttributes .....	132
Table 177 – ObjectTypeAttributes.....	132
Table 178 – VariableTypeAttributes .....	132
Table 179 – ReferenceTypeAttributes .....	132
Table 180 – DataTypeAttributes .....	133
Table 181 – ViewAttributes .....	133
Table 182 – UserIdentityToken parameterTypeIds .....	133
Table 183 – UserName Identity Token .....	133
Table 184 – X509 Identity Token .....	134
Table 185 – Issued Identity Token .....	134

## OPC FOUNDATION

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these

materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>



## 1 Scope

This specification specifies the OPC Unified Architecture Services.

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 6] OPC UA Specification: Part 6 – Mappings, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part6/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part8/>

[UA Part 11] OPC UA Specification: Part 11 – Historical Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part11/>

[UA Part 12] OPC UA Specification: Part 12 – Discovery, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part12/>

## 3 Terms, definitions, and conventions

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Attribute
- 3) Certificate
- 4) Client
- 5) Communication Stack
- 6) Event
- 7) EventNotifier
- 8) Message
- 9) MonitoredItem
- 10) Node
- 11) NodeClass
- 12) Notification

- 13) NotificationMessage
- 14) Object
- 15) ObjectType
- 16) Profile
- 17) Reference
- 18) ReferenceType
- 19) Server
- 20) Service
- 21) Service Set
- 22) Session
- 23) Subscription
- 24) Variable
- 25) View

### 3.2 OPC UA Part 2 terms

The following terms defined in [UA Part 2] apply.

- 1) Authentication
- 2) Authorization
- 3) Confidentiality
- 4) Integrity
- 5) Nonce
- 6) OPC UA Application
- 7) SecureChannel
- 8) SecurityToken
- 9) SessionKeySet
- 10) PrivateKey
- 11) PublicKey
- 12) X.509 Certificate

### 3.3 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

- 1) EventType
- 2) HierarchicalReference
- 3) InstanceDeclaration
- 4) ModellingRule
- 5) Property
- 6) SourceNode
- 7) TargetNode
- 8) TypeDefinitionNode
- 9) VariableType

### 3.4 OPC UA Services terms

#### 3.4.1 Deadband

A *Deadband* specifies a permitted range for value changes that will not trigger a data change *Notification*. It can be applied as filter when subscribing to *Variables* and is used to keep noisy signals from updating the *Client* unnecessarily.

This specification defines *AbsoluteDeadband* as a common filter. [UA Part 8] defines an additional *Deadband* filter.

#### 3.4.2 Endpoint

An *Endpoint* is a physical address available on a network that allows *Clients* to access one or more *Services* provided by a *Server*. Each *Server* may have multiple *Endpoints*.

#### 3.4.3 ServerUri

A *ServerUri* is a globally unique identifier for a *Server* application instance. It may be a *GUID* generated automatically during install or it could be a unique *URL* assigned by the administrator.

#### 3.4.4 SoftwareCertificate

A digital certificate for a software product, which can be installed on several hosts to describe the capabilities of the software product. Different installations of one software product could have the same software certificate.

### 3.5 Abbreviations and symbols

API	Application Programming Interface
BNF	Backus-Naur Form
CA	Certificate Authority
CRL	Certificate Revocation List
CTL	Certificate Trust List
DA	Data Access
UA	Unified Architecture
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

### 3.6 Conventions for Service definitions

OPC UA Services contain parameters that are conveyed between the *Client* and the *Server*. The UA Service specifications use tables to describe *Service* parameters, as shown in Table 1. Parameters are organised in this table into request parameters and response parameters.

**Table 1 – Service Definition Table**

Name	Type	Description
<b>Request</b>		Defines the request parameters of the Service
Simple Parameter Name		Description of this parameter
Constructed Parameter Name		Description of the constructed parameter
Component Parameter Name		Description of the component parameter
<b>Response</b>		Defines the response parameters of the Service

The Name, Type and Description columns contain the name, data type and description of each parameter. All parameters are mandatory, although some may be unused under certain circumstances. The description column specifies the value to be supplied when a parameter is unused.

Two types of parameters are defined in these tables, simple and constructed. Simple parameters have a simple data type, such as *Boolean* or *String*.

Constructed parameters are composed of two or more component parameters, which can be simple or constructed. Component parameter names are indented below the constructed parameter name.

The data types used in these tables may be base types, common types to multiple *Services* or *Service-specific* types. Base data types are defined in [UA Part 3]. The base types used in *Services* are listed in Table 2. Data types that are common to multiple *Services* are defined in Clause 7 and Clause 8. Data types that are *Service-specific* are defined in the parameter table of the *Service*.

**Table 2 – Parameter Types defined in [UA Part 3]**

Parameter Type
BaseDataType
NodeId
QualifiedName
LocaleId
Boolean
ByteString
Double
Guid
Int32
String
UInt32
UtcTime
Xmlelement

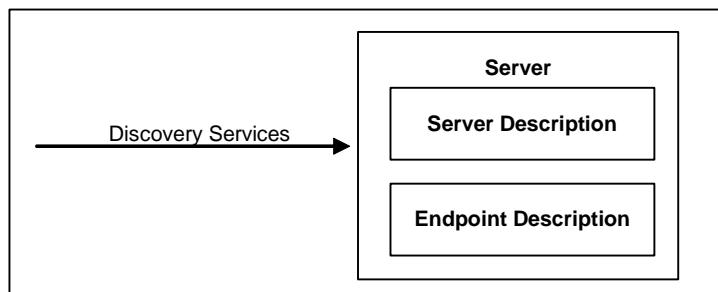
## 4 Overview

### 4.1 Service Set model

This clause specifies the OPC UA *Services*. The OPC UA *Service* definitions are abstract descriptions and do not represent a specification for implementation. The mapping between the abstract descriptions and the *Communication Stack* derived from these *Services* are defined in [UA Part 6]. In the case of an implementation as web services, the OPC UA *Services* correspond to the web service and an OPC UA *Service* corresponds to an operation of the web service.

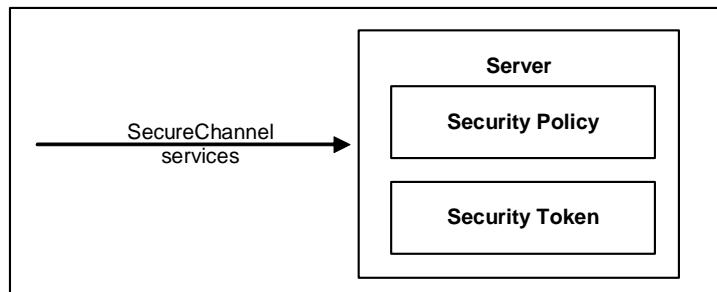
These *Services* are organised into *Service Sets*. Each *Service Set* defines a set of related *Services*. The organisation in *Service Sets* is a logical grouping used in the specification and is not used in the implementation.

The *Discovery Service Set*, illustrated in Figure 1, defines *Services* that allow a *Client* to discover the *Endpoints* implemented by a *Server* and to read the security configuration for each of those *Endpoints*.



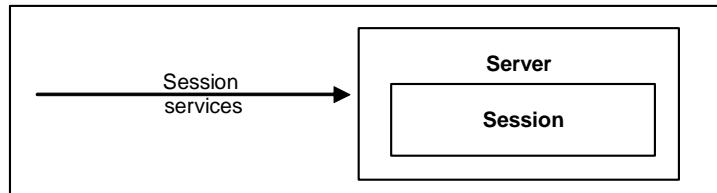
**Figure 1 – Discovery Service Set**

The *SecureChannel Service Set*, illustrated in Figure 2, defines *Services* that allow a *Client* to establish a communication channel to ensure the *Confidentiality* and *Integrity* of *Messages* exchanged with the *Server*.



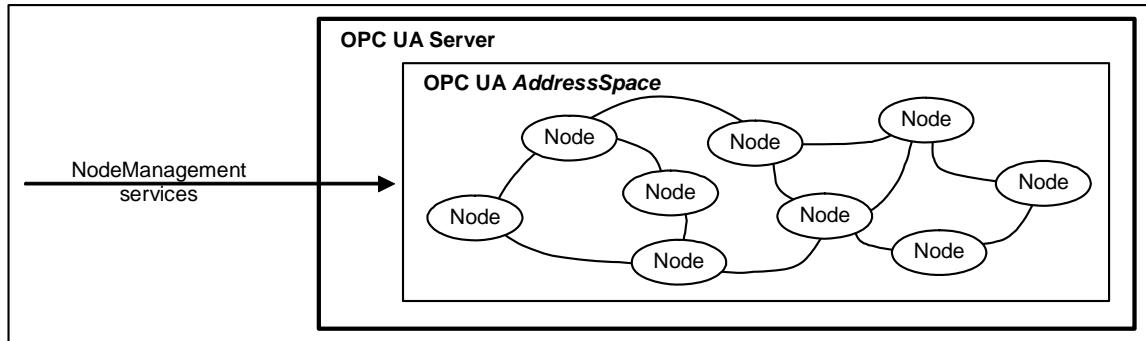
**Figure 2 – SecureChannel Service Set**

The *Session Service Set*, illustrated in Figure 3, defines *Services* that allow the *Client* to authenticate the User it is acting on behalf of and to manage *Sessions*.



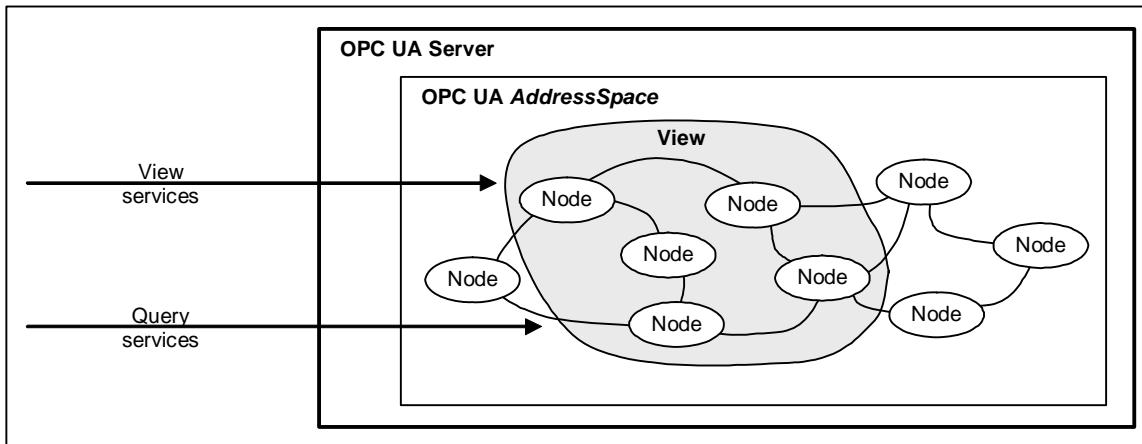
**Figure 3 – Session Service Set**

The *NodeManagement Service Set*, illustrated in Figure 4, defines *Services* that allow the *Client* to add, modify and delete *Nodes* in the *AddressSpace*.



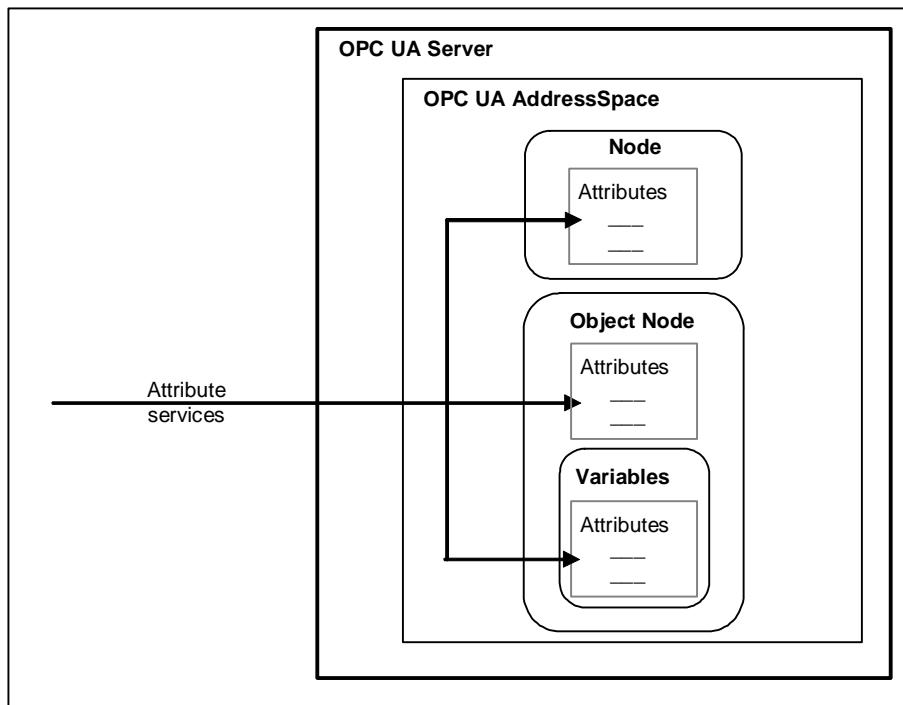
**Figure 4 – NodeManagement Service Set**

The *View Service Set*, illustrated in Figure 5, defines *Services* that allow *Clients* to browse through the *AddressSpace* or subsets of the *AddressSpace* called *Views*. The *Query Service Set* allows *Clients* to return a subset of data from the *View*.



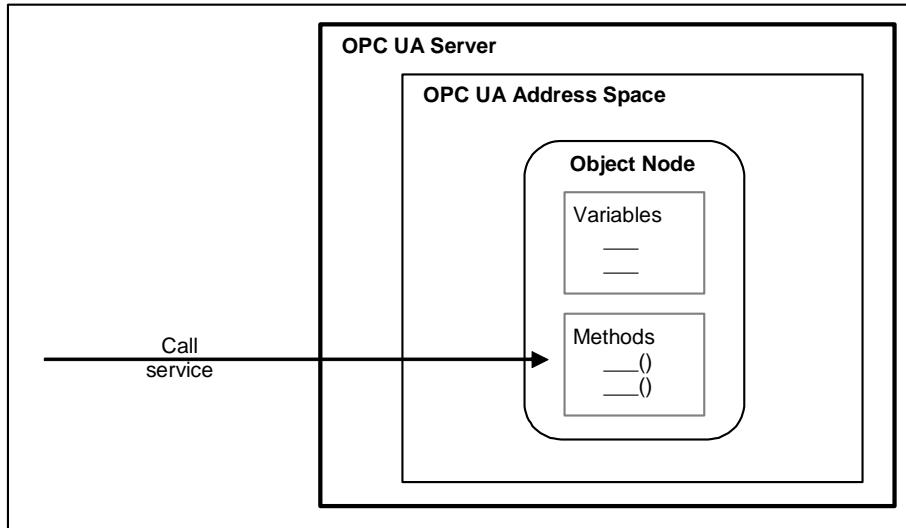
**Figure 5 – View Service Set**

The *Attribute Service Set* is illustrated in Figure 6. It defines *Services* that allow *Clients* to read and write *Attributes* of *Nodes*, including their historical values. Since the value of a *Variable* is modelled as an *Attribute*, these *Services* allow *Clients* to read and write the values of *Variables*.



**Figure 6 – Attribute Service Set**

The *Method Service Set* is illustrated in Figure 7. It defines Services that allow *Clients* to call methods. Methods run to completion when called. They may be called with method-specific input parameters and may return method-specific output parameters.



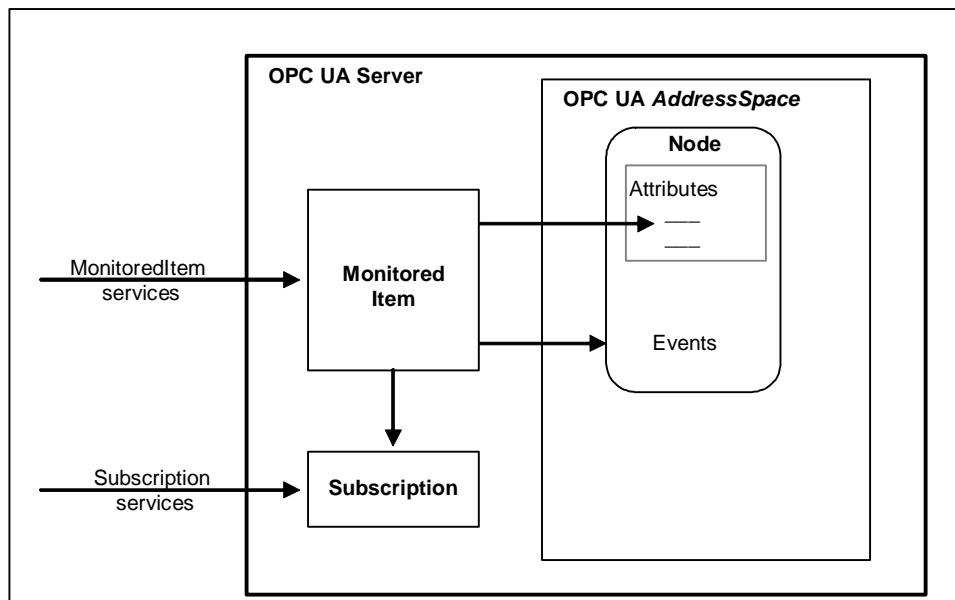
**Figure 7 – Method Service Set**

The *MonitoredItem Service Set* and the *Subscription Service Set*, illustrated in Figure 8, are used together to subscribe to *Nodes* in the OPC UA AddressSpace.

The *MonitoredItem Service Set* defines Services that allow *Clients* to create, modify, and delete *MonitoredItems* external to the OPC UA AddressSpace. *MonitoredItems* monitor *Attributes* for value changes and *Nodes* for *Events*, and generate *Notifications* for them.

These *Notifications* are queued for transfer to the *Client* by *Subscriptions*.

The *Subscription Service Set* defines Services that allow *Clients* to create, modify and delete *Subscriptions*. *Subscriptions* send *Notifications* generated by *MonitoredItems* to the *Client*. *Subscription Services* also provide for *Client* recovery from missed *Messages* and communication failures.



**Figure 8 – MonitoredItem and Subscription Service Sets**

## 4.2 Request/response Service procedures

Request/response Service procedures describe the processing of requests received by the *Server*, and the subsequent return of responses. The procedures begin with the requesting *Client* submitting a *Service request Message* to the *Server*.

Upon receipt of the request, the *Server* processes the *Message* in two steps. In the first step, it attempts to decode and locate the *Service* to execute. The error handling is specific to the communication technology used and is described in [UA Part 6].

If it succeeds, then it attempts to access each operation identified in the request and perform the requested operation. For each operation in the request, it generates a separate success/failure code that it includes in a positive response *Message* along with any data that is to be returned.

To perform these operations, both the *Client* and the *Server* may make use of the API of a *Communication Stack* to construct and interpret *Messages* and to access the requested operation.

The implementation of each service request or response handling must check that each service parameter lies within the specified range for that parameter.

## 5 Service Sets

### 5.1 General

This clause defines the OPC UA *Service Sets* and their *Services*. Clause 7 and Clause 8 contain the definitions of common parameters used by these *Services*.

Whether or not a *Server* supports a *Service Set*, or a *Service* within a *Service Set*, is defined by its *Profile*. *Profiles* are described in [UA Part 7].

### 5.2 Service request and response header

Each *Service* request has a *RequestHeader* and each *Service* response has a *ResponseHeader*.

The *RequestHeader* structure is defined in Clause 7.19 and contains common request parameters such as *sessionId*, *timestamp* and *sequenceNumber*.

The *ResponseHeader* structure is defined in Clause 7.20 and contains common response parameters such as *serviceResult* and *diagnosticInfo*.

### 5.3 Service results

*Service* results are returned at two levels in OPC UA responses, one that indicates the status of the *Service* call, and the other that indicates the status of each operation requested by the *Service*.

*Service* results are defined via the *StatusCodes* (see Clause 7.28).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see Clause 7.20). The mechanism for returning this parameter is specific to the communication technology used to convey the *Service* response and is defined in [UA Part 6].

The status of partial operations in a request is represented by individual *StatusCodes*.

The following cases define the use of these parameters.

- a) A bad code is returned in *serviceResult* if the *Service* itself failed. In this case, a *ServiceFault* is returned. The *ServiceFault* is defined in Clause 7.23.
- b) The good code is returned in *serviceResult* if the *Service* fully or partially succeeded. In this case, other response parameters are returned. The *Client* must always check the response parameters, especially all *StatusCodes* associated with each operation. These *StatusCodes* may indicate bad or uncertain results for one or more operations requested in the *Service* call.

All *Services* with arrays of operations in the request must return a bad code in the *serviceResult* if the array is empty. The bad *StatusCodes* indicates which parameter was wrong.

The *Services* define various specific *StatusCodes* and a *Server* must use these specific *StatusCodes* as described in the *Service*.

If the *Server* discovers, through some out-of-band mechanism, that the application or user credentials used to create a *Session* or *SecureChannel* have been compromised, then the *Server* must immediately terminates all sessions and channels that use those credentials. In this case, the *Service* result code must be either *Bad\_UserIdentityNoLongerValid* or *Bad\_CertificateNoLongerValid*.

## 5.4 Discovery Service Set

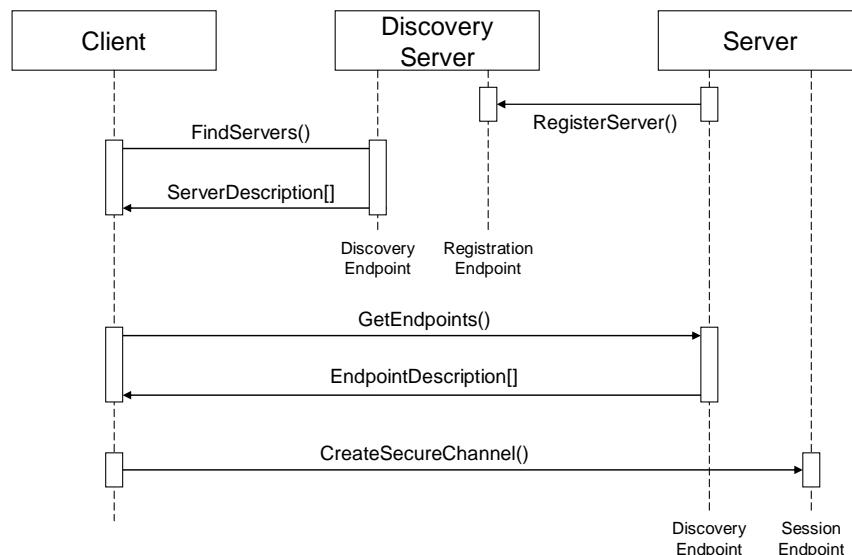
### 5.4.1 Overview

This Service Set defines Services used to discover the *Endpoints* implemented by a Server and to read the security configuration for those *Endpoints*. The *Discovery Services* are implemented by individual *Servers* and by dedicated *Discovery Servers*. [UA Part 12] describes how to use the *Discovery Services* with dedicated *Discovery Servers*.

Every *Server* must have a *Discovery Endpoint* that *Clients* can access without establishing a *Session*. This *Endpoint* may or might not be the same *Session Endpoint* that *Clients* use to establish a *SecureChannel*. *Clients* read the security information necessary to establish a *SecureChannel* by calling the *GetEndpoints* Service on the *Discovery Endpoint*.

In addition, *Servers* may register themselves with a well known *Discovery Server* using the *RegisterServer* service. *Clients* can later discover any registered *Servers* by calling the *FindServers* Service on the *Discovery Server*.

The complete discovery process is illustrated in Figure 9.



**Figure 9 – The Discovery Process**

The URL for a *Discovery Endpoint* must provide all of the information that the *Client* needs to connect to the *Endpoint*. This implies that no security can be applied to service call itself, however, some implementations may use transport layer security where the security protocol is identified in the URL (e.g. HTTPS).

Once a *Client* retrieves the *Endpoints*, the *Client* can save this information and use it to connect directly to the *Server* again without going through the discovery process. If the *Client* finds that it cannot connect then that could mean the *Server* configuration has changed and the *Client* needs to go through the discovery process again.

### 5.4.2 FindServers

#### 5.4.2.1 Description

This Service returns the *Servers* known to a *Discovery Server*. The behavoir of *Discovery Servers* is described in detail in [UA Part 12].

The *Client* may reduce the number of results returned by specifying filter criteria. A *Discovery Server* returns an empty list if no *Servers* match the criteria specified by the client. The filter criteria supported by this *Service* are described in Clause 5.4.2.2.

Every *Server* must provide a *Discovery Endpoint* that supports this *Service*, however, the *Server* will only return a single record that describes itself.

Every *Server* must have a globally unique identifier called the *ServerUri*. This identifier should be a fully qualified domain name, however, it may be a GUID or similar construct that ensures global uniqueness. The *ServerUri* returned by this *Service* must be the same value that appears in index 0 of the *ServerArray* property (see [UA Part 5]).

Every *Server* must also have a human readable identifier called the *ServerName* which is not necessarily globally unique. This identifier may be available in multiple locales.

This *Service* must not require any message security but it may require transport layer security.

#### 5.4.2.2 Parameters

Table 3 defines the parameters for the *Service*.

**Table 3 – FindServers Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
localeIds []	LocaleId	List of locales to use. The server should return the <i>ServerName</i> using one of locales specified. If the server supports more than one of the requested locales then the server must use the locale that appears first in this list. If the server does not support any of the requested locales it chooses an appropriate default locale. The server chooses an appropriate default locale if this list is empty.
serverUris []	String	List of servers to return. All known servers are returned if the list is empty.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.20.
servers []	ServerDescription	List of <i>Servers</i> that meet criteria specified in the request. This list is empty if no servers meet the criteria. The <i>ServerDescription</i> type is defined in Clause 7.23.

#### 5.4.2.3 Service results

Common *StatusCodes* are defined in Table 156.

#### 5.4.3 GetEndpoints

##### 5.4.3.1 Description

This *Service* returns the *Endpoints* supported by a *Server* and all of the configuration information required to establish a *SecureChannel* and a *Session*.

This *Service* must not require any message security but it may require transport layer security.

A *Client* may reduce the number of results returned by specifying filter criteria. The *Server* returns an empty list if no *Endpoints* match the criteria specified by the client. The filter criteria supported by this *Service* are described in Clause 5.4.3.2.

A *Server* may support multiple security configurations for the same *Endpoint*. In this situation, the *Server* must return separate *EndpointDescription* records for each available configuration. *Clients* should treat each of these configurations as distinct *Endpoints* even if the physical URL happens to be the same.

The security configuration for an *Endpoint* has four components:

- 1) Server Application Instance Certificate
- 2) Message Security Mode
- 3) Security Policy
- 4) Supported User Identity Tokens

The *ApplicationInstanceCertificate* is used to secure the *OpenSecureChannel* request (See Clause 5.5.2). The *MessageSecurityMode* and the *SecurityPolicy* tell the *Client* how to secure messages sent via the *SecureChannel*. The *UserIdentityTokens* tell the client what user credentials must be passed to the *Server* in the *CreateSession* request (See Clause 5.6.2).

Each *EndpointDescription* also specifies one or more URIs for supported *Profiles*. These values indicate which *Transport Profiles* defined in [UA Part 7] that the *Endpoint* supports. *Transport Profiles* specify information such as message encoding format and protocol version. The *Server* should only return *Profiles* that a *Client* needs to be aware of before it connects to the *Server*. *Clients* must fetch the *Server's SoftwareCertificates* if they want to discover the complete list of *Profiles* supported by the *Server* (See Clause 7.25).

Messages are secured by applying standard cryptography algorithms to the messages before they are sent over the network. The exact set of algorithms used depends on the *SecurityPolicy* for the *Endpoint*. [UA Part 7] defines *Profiles* for common *SecurityPolicies* and assigns a unique URI to them. It is expected that applications have built in knowledge of the *SecurityPolicies* that they support, as a result, only the Profile URI for the *SecurityPolicy* is specified in the *EndpointDescription*. A *Client* cannot connect to an *Endpoint* that does not support a *SecurityPolicy* that it recognizes.

An *EndpointDescription* may specify that the message security mode is NONE. This configuration is not recommended unless the applications are communicating on a physically isolated network where the risk of intrusion is extremely small. If the message security is NONE then it is possible for *Clients* to deliberately or accidentally hijack Sessions created by other *Clients*.

### 5.4.3.2 Parameters

Table 4 defines the parameters for the *Service*.

**Table 4 – GetEndpoints Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
localeIds []	LocaleId	List of locales to use. Specifies the locale to use when returning human readable strings. This parameter is described in Clause 5.4.2.2.
profileUris []	String	List of profiles that the returned <i>Endpoints</i> must support. All <i>Endpoints</i> are returned if the list is empty.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.20.
Endpoints []	EndpointDescription	List of <i>Endpoints</i> that meet criteria specified in the request. This list is empty if no <i>Endpoints</i> meet the criteria.
endpointUrl	String	The URL for the <i>Endpoint</i> described.
server	ServerDescription	The description for the <i>Server</i> that the <i>Endpoint</i> belongs to. The <i>ServerDescription</i> type is defined in Clause 7.23.
serverCertificate	ByteString	The application instance <i>Certificate</i> issued to the <i>Server</i> . This is a DER encoded X509v3 certificate.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type <i>MessageSecurityMode</i> type is defined in Clause 7.10. A <i>SecureChannel</i> may have to be created even if the <i>securityMode</i> is NONE. The exact behavoir depends on the mapping used and is described in the [UA Part 6].
securityPolicy	String	The URI for <i>SecurityPolicy</i> to use when securing messages. This value identifies an instance of the <i>SecurityPolicy</i> type described in Clause 7.22. The set of known URIs and the <i>SecurityPolicies</i> associated with them are defined in [UA Part 7].
userIdentityTokens[]	UserTokenPolicy	The user identity tokens that the <i>Server</i> will accept. The <i>Client</i> must pass one of the <i>UserIdentityTokens</i> in the <i>ActivateSession</i> request. The <i>UserTokenPolicy</i> type is described in Clause 7.30.
supportedProfiles	String	The URI of the <i>Transport Profile</i> supported by the <i>Endpoint</i> . [UA Part 7] defines URIs for the standard <i>Transport Profiles</i> .

### 5.4.3.3 Service Results

Common *StatusCodes* are defined in Table 156.

## 5.4.4 RegisterServer

### 5.4.4.1 Description

This *Service* registers a *Server* with a *Discovery Server*. This *Service* will be called by a *Server* or a separate configuration utility. *Clients* will not use this *Service*.

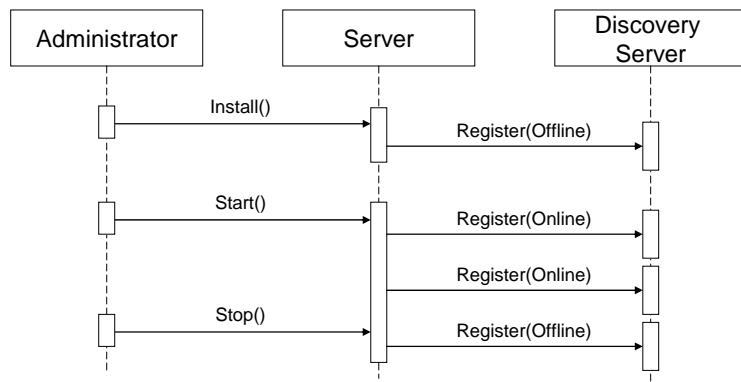
A *Server* only provides its *ServerUri* and the URLs of the *Discovery Endpoints* to the *Discovery Server*. *Clients* must use the *GetEndpoints* service to fetch the most up to date configuration information directly from the *Server*.

The *Server* must provide a localized name for itself in all locales that it supports.

Servers must be able to register themselves with a *Discovery Server* running on the same machine. The exact mechanisms depend on the *Discovery Server* implementation and are described in [UA Part 6].

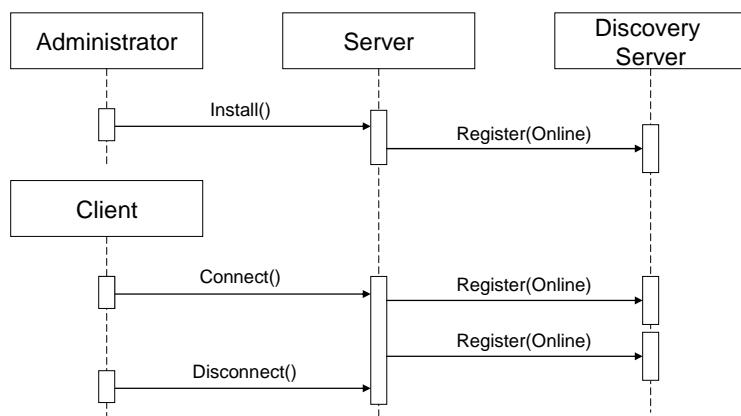
There are two types of *Server* applications: those which are manually launched and those that are automatically launched when a *Client* attempts to connect. The registration process that a *Server* must use depends on which category it falls into.

The registration process for manually launched Servers is illustrated in Figure 10.



**Figure 10 – The Registration Process – Manually Launched Servers**

The registration process for automatically launched Servers is illustrated in Figure 11.



**Figure 11 – The Registration Process – Automatically Launched Servers**

The registration process is designed to be platform independent, robust and able to minimize errors created by misconfiguration. For that reason, *Servers* must register themselves more than once.

Under normal conditions, *Servers* must periodically register with the *Discovery Server* as long as they are able to receive connections from *Clients*. If a *Server* goes offline then it must register itself once more and indicate that it is going offline. The registration frequency should be configurable, however, the default is 10 minutes.

If an error occurs during registration (e.g. the *Discovery Server* is not running) then the *Server* must periodically re-attempt registration. The frequency of these attempts should start at 1 second but gradually increase until the registration frequency is the same as what it would be if no errors

occurred. The recommended approach would double the period each attempt until reaching the maximum.

When a *Server* registers with the a *Discovery Server* it may choose to provide a semaphore file which the *Discovery Server* can use to determine if the *Server* has been uninstalled from the machine. The *Discovery Server* must have read access to the file system that contains the file.

#### 5.4.4.2 Parameters

Table 5 defines the parameters for the *Service*.

**Table 5 – RegisterServers Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
server	RegisteredServer	The server to register.
serverUri	String	The globally unique identifier for the <i>Server</i> .
serverNames []	LocalizedText	A list of localized descriptive names for the <i>Server</i> . The list must have at least one valid entry.
discoveryUrls []	String	A list of <i>Discovery Endpoints</i> for the <i>Server</i> . The list must have at least one valid entry.
semaphoreFilePath	String	The path to the semaphore file used to identify the server instance. The <i>Discovery Server</i> will always check that this file exists before returning the <i>ServerDescription</i> to the client. If the same semaphore file is used by another <i>Server</i> then that registration is deleted and replaced by the one being passed into this method. If this value is null or empty then the <i>DiscoveryServer</i> does not attempt to verify the existence of the file.
isOnline	Boolean	True if the <i>Server</i> is currently able to accept connections from <i>Clients</i> .
<b>Response</b>		
ResponseHeader	ResponseHeader	Common response parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.20.

#### 5.4.4.3 Service Results

Table 6 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 6 – RegisterServers Service Result Codes**

Symbolic Id	Description
Bad_ServerUriInvalid	The <i>ServerUri</i> is not a valid URI.
Bad_ServerNameMissing	No <i>ServerName</i> was specified.
Bad_DiscoveryUrlMissing	No <i>DiscoveryUrl</i> was specified.
Bad_SempahoreFileMissing	The semaphore file specified by the client is not valid.

### 5.5 SecureChannel Service Set

#### 5.5.1 Overview

This *Service Set* defines *Services* used to open a communication channel that ensures the confidentiality and *Integrity* of all *Messages* exchanged with the *Server*. The base concepts for UA security are defined in [UA Part 2].

The *SecureChannel Services* are unlike other *Services* because they are not implemented directly by the *UA Application*. Instead, they are provided by the *Communication Stack* on which the *UA Application* is built. For example, a *UA Server* may be built on a SOAP stack that allows applications to establish a *SecureChannel* using the WS Secure Conversation specification. In

these cases, the *UA Application* must verify that the *Message* it received was in the context of a WS Secure Conversation. [UA Part 6] describes how the *SecureChannel Services* are implemented.

A *SecureChannel* is a long-running logical connection between a single *Client* and a single *Server*. This channel maintains a set of keys known only to the *Client* and *Server*, which are used to authenticate and encrypt *Messages* sent across the network. The *SecureChannel Services* allow the *Client* and *Server* to securely negotiate the keys to use.

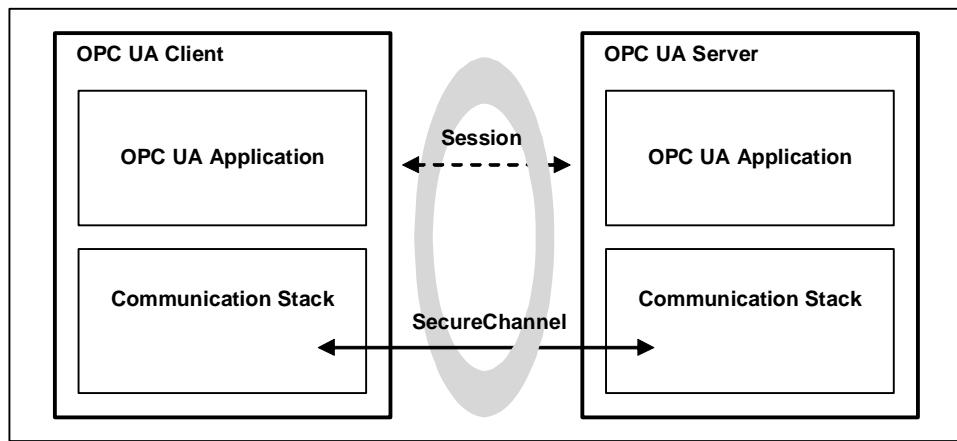
An *EndpointDescription* tells a *Client* how to establish a *SecureChannel* with a given *Endpoint*. A *Client* may obtain the *EndpointDescription* from a *Discovery Server*, via some non-UA defined directory server or from its own configuration.

The exact algorithms used to authenticate and encrypt *Messages* are described in the *SecurityPolicy* field of the *EndpointDescription*. A *Client* must use these algorithms when it creates a *SecureChannel*.

When a *Client* and *Server* are communicating via a *SecureChannel*, they must verify that all incoming *Messages* have been signed and encrypted according to the requirements specified in the *EndpointDescription*. A *UA Application* must not process any *Message* that does not conform to these requirements.

A *SecureChannel* is separate from the *UA Application Session*; however, a single *UA Application Session* may only be accessed via a *SecureChannel* that has been explicitly bound to the *Session*. This implies that the *UA Application* must be able to determine which *SecureChannel* is associated with each *Message*. A *Communication Stack* that provides a *SecureChannel* mechanism but does not allow the *UA Application* to know which *SecureChannel* was used for a given *Message* cannot be used to implement the *SecureChannel Service Set*.

The correlation between the *UA Application Session* and the *SecureChannel* is illustrated in Figure 12. The *Communication Stack* is used by the *UA Applications* to exchange *Messages*. In a first step, the *SecureChannel Services* are used to establish a *SecureChannel* between the two *Communication Stacks* to exchange *Messages* in a secure way. In a second step, the *UA Applications* use the *Session Service Set* to establish a *UA Application Session*.



**Figure 12 – SecureChannel and Session Services**

Once a *Client* has established a *Session* it may wish to access the *Session* from a different *SecureChannel*. The *Client* can do this by validating the new *SecureChannel* with the *ActivateSession* Service described in Clause 5.6.3.

If a *Server* acts as a *Client* to other *Servers*, which is commonly referred to as *Server chaining*, then the *Server* must be able to maintain user level security. By this we mean that the user identity should be passed to the underlying server or it should be mapped to an appropriate user identity in the underlying server. It is unacceptable to ignore user level security. This is required to ensure that

security is maintained and that a user does not obtain information that they should not have access to. For example the server may establish a *SecureChannel* to each *Server* for which it is a *Client*. Further more for each chained *Server* it may establish a *Session*. The *Session* that is established to a chained *Server* may be established using the same user identity, that was used to connect to the orginal *Server*. Another example would be to establish the *Session* using a different user identity, but for each command passed to the underlying server the user Identity must be switched via the *ActivateSession* Service to the user identity of the originating *Servers Client*. Another possibility would be for the highlevel server to map it's *Clients* to distinct underlying *Server* accounts or account groups.

## 5.5.2 OpenSecureChannel

### 5.5.2.1 Description

This Service is used to open or renew a *SecureChannel* that can be used to ensure *Confidentiality* and *Integrity* for *Message* exchange during a *Session*. This Service requires the *Communication Stack* to apply the various security algorithms to the *Messages* as they are sent and received. Specific implementations of this Service for different *Communication Stacks* are described in [UA Part 6].

Each *SecureChannel* has a globally-unique identifier and is valid for a specific combination of *Client* and *Server* application instances. Each channel contains one or more *SecurityTokens* that identify a set of cryptography keys that are used to encrypt and authenticate *Messages*. *SecurityTokens* also have globally-unique identifiers which are attached to each *Message* secured with the token. This allows an authorized receiver to know how to decrypt and verify the *Message*.

*SecurityTokens* have a finite lifetime negotiated with this Service. However, differences between the system clocks on different machines and network latencies mean that valid *Messages* could arrive after the token has expired. To prevent valid *Messages* from being discarded, the applications should do the following:

1. *Clients* should request a new *SecurityToken*s after 75% of its lifetime has elapsed. This should ensure that *Clients* will receive the new *SecurityToken* before the old one actually expires.
2. *Servers* should use the existing *SecurityToken* to secure outgoing *Messages* until it expires, even if it has been renewed early. This should ensure that *Clients* do not reject *Messages* secured with the new *SecurityToken* that arrive before the *Client* receives the new *SecurityToken*.
3. *Clients* should accept *Messages* secured by an expired *SecurityToken* for up to 25% of the token lifetime. This should ensure that *Messages* sent by the *Server* before the token expired are not rejected because of network delays.

Each *SecureChannel* exists until it is explicitly closed or until the last token has expired and the overlap period has elapsed.

The *OpenSecureChannel* request and response *Messages* must be signed with the sender's *Certificate*. These *Messages* must always be encrypted. If the transport layer does not provide encryption, then these *Messages* must be encrypted with the receiver's *Certificate*.

The *Certificates* used in the *OpenSecureChannel* service should be the application instance Certificates, however, some communication stacks will not allow *Certificates* that are specific to single application. If this is the case, then *Certificates* which are specific to a machine or an individual user may be used instead.

### 5.5.2.2 Parameters

Table 7 defines the parameters for the *Service*.

**Table 7 – OpenSecureChannel Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
clientCertificate	ByteString	A <i>Certificate</i> that identifies the <i>Client</i> . The <i>OpenSecureChannel</i> request must be signed with this <i>Certificate</i> .
requestType	enum SecurityToken RequestType	The type of <i>SecurityToken</i> request: An enumeration that must be one of the following: ISSUE_0 creates a new <i>SecurityToken</i> for a new <i>SecureChannel</i> . RENEW_1 creates a new <i>SecurityToken</i> for an existing <i>SecureChannel</i> .
secureChannelId	Guid	The identifier for the <i>SecureChannel</i> that the new token should belong to. This parameter must be null when creating a new <i>SecureChannel</i> .
securityPolicy	String	The URI for <i>SecurityPolicy</i> to use when securing messages sent over the <i>SecureChannel</i> . This value identifies an instance of the <i>SecurityPolicy</i> described in Clause 7.22. The set of known URIs and the <i>SecurityPolicies</i> associated with them are defined in [UA Part 7].
clientNonce	ByteString	A random number that must not be used in any other request. A new <i>clientNonce</i> must be generated for each time a <i>SecureChannel</i> is renewed. This parameter must have a length equal to the <i>symmetricKeyLength</i> specified in the requested <i>SecurityPolicy</i> . See Clause 7.22 for <i>SecurityPolicy</i> definition.
requestedLifetime	Duration	The requested lifetime, in milliseconds, for the new <i>SecurityToken</i> (see Clause 7.6 for <i>Duration</i> definition). It specifies when the <i>Client</i> expects to renew the <i>SecureChannel</i> by calling the <i>OpenSecureChannel</i> Service again. If a <i>SecureChannel</i> is not renewed, then all <i>Messages</i> sent using the current <i>SecurityTokens</i> will be rejected by the receiver.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> type definition). The <i>sessionId</i> is always set to 0 and the <i>securityHeader</i> is always set to null in this response.
serverCertificate	ByteString	A <i>Certificate</i> that identifies the <i>Server</i> . The <i>OpenSecureChannel</i> response must be signed with this <i>Certificate</i> .
securityToken	ChannelSecurity Token	Describes the new <i>SecurityToken</i> issued by the <i>Server</i> .
channelId	Guid	A globally-unique identifier for the <i>SecureChannel</i> . This is the identifier that must be supplied whenever the <i>SecureChannel</i> is renewed.
tokenId	String	A globally-unique identifier for a single <i>SecurityToken</i> within the channel. This is the identifier that must be passed with each <i>Message</i> secured with the <i>SecurityToken</i> .
createdAt	UtcTime	When the <i>SecurityToken</i> was created.
revisedLifetime	Duration	The lifetime of the <i>SecurityToken</i> in milliseconds (see Clause 7.6 for <i>Duration</i> definition). The UTC expiration time for the token may be calculated by adding the lifetime to the <i>createdAt</i> time.
serverNonce	ByteString	A random number that must not be used in any other request. This parameter must have a length equal to the <i>symmetricKeyLength</i> specified in the requested <i>SecurityPolicy</i> . See Clause 7.22 for <i>SecurityPolicy</i> definition. A new <i>serverNonce</i> must be generated for each time a <i>SecureChannel</i> is renewed.

### 5.5.2.3 Service results

Table 8 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 8 – OpenSecureChannel Service Result Codes**

Symbolic Id	Description
Bad_CertificateInvalid	The <i>Client</i> certificate is not valid.
Bad_CertificateExpired	The <i>Client</i> certificate is expired or not yet valid.
Bad_CertificateRevoked	The <i>Client</i> certificate has been revoked by the certification authority.
Bad_CertificateUntrusted	The <i>Client</i> certificate is valid; however, the server does not recognize it as a trusted certificate.
Bad_RequestTypeInvalid	The security token request type is not valid.
Bad_SecurityPolicyRejected	The security policy does not meet the requirements set by the <i>Server</i> .
Bad_SecureChannelIdInvalid	See Table 156 for the description of this result code.
Bad_NonceInvalid	See Table 156 for the description of this result code.

### 5.5.3 CloseSecureChannel

#### 5.5.3.1 Description

This Service is used to terminate a *SecureChannel*.

The request *Messages* must be signed with the same *Certificate* that was used to sign the *OpenSecureChannel* request.

#### 5.5.3.2 Parameters

Table 9 defines the parameters for the Service.

**Table 9 – CloseSecureChannel Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is always set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
secureChannelId	Guid	The identifier for the <i>SecureChannel</i> to close.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).

#### 5.5.3.3 Service results

Table 10 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 10 – CloseSecureChannel Service Result Codes**

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 156 for the description of this result code.

## 5.6 Session Service Set

### 5.6.1 Overview

This Service Set defines Services for an application layer connection establishment in the context of a *Session*.

### 5.6.2 CreateSession

#### 5.6.2.1 Description

This Service is used by an OPC UA Client to create a *Session* and the Server returns a *sessionId* that uniquely identifies the *Session*. The Client submits this *sessionId* on all subsequent Service requests that it submits on the *Session*.

Before calling this Service, the Client must create a *SecureChannel* with the *OpenSecureChannel* Service to ensure the *Integrity* of all *Messages* exchanged during a *Session*. This *SecureChannel* has a unique identifier, which the Server must associate with the *sessionId*. The Server may accept requests with the *sessionId* only if they are associated with the same *SecureChannel* that was used to create the *Session*. The Client may associate a new *SecureChannels* with the *Session* by calling the *ActivateSession* method. The Server must verify that the *Certificate* the Client used to create the new *SecureChannel* is the same as the *Certificate* used to create the original *SecureChannel*.

In most cases, the *SecureChannel* will be managed by the *Communication Stack*, so the Server must be able to *Query* the *Communication Stack* for the *secureChannelId* associated with an incoming request. The exact mechanism depends on the implementation. [UA Part 6] describes how this is done with common *Communication Stacks*.

The *Session* created with this Service must not be used until the Client calls the *ActivateSession* Service and provides its *SoftwareCertificates* and proves possession of its application instance *Certificate* and any user identity token that it provided.

The response also contains a list of *SoftwareCertificates* that identify the capabilities of the Server. It contains the list of OPC UA *Profiles* supported by the Server. OPC UA *Profiles* are defined in [UA Part 7].

Additional *Certificates* issued by other organisations may be included to identify additional Server capabilities. Examples of these *Profiles* include support for specific information models and support for access to specific types of devices.

When a *Session* is created, the Server adds an entry for the Client in its *SessionDiagnosticArray Variable*. See [UA Part 5] for a description of this *Variable*.

*Sessions* are created to be independent of the underlying communications connection. Therefore, if a communications connection fails, the *Session* is not immediately affected. The exact mechanism to recover from an underlying communication connection error depends on the *SecureChannel* mapping described in [UA Part 6].

*Sessions* are terminated by the Server automatically if the Client fails to issue a Service request on the *Session* within the timeout period negotiated by the Server in the *CreateSession* Service response. This protects the Server against Client failures and against situations where a failed underlying connection cannot be re-established. Clients must be prepared to submit requests in a timely manner to prevent the *Session* from closing automatically. Clients may explicitly terminate *Sessions* using the *CloseSession* Service.

When a *Session* is terminated, all outstanding requests on the *Session* are aborted and *Bad\_SessionClosed StatusCodes* are returned to the Client. In addition, the Server deletes the entry for the Client from its *SessionDiagnosticArray Variable* and notifies any other Clients who were subscribed to this entry.

*Subscriptions* assigned to the *Client*, however, are not necessarily terminated when the *Session* is terminated. Each has its own lifetime to protect against data loss if a *Session* terminates abruptly. In these cases, the *Subscription* can be reassigned to another *Client* before its lifetime expires.

Some *Servers*, such as aggregating *Servers*, also act as *Clients* to other *Servers*. These *Servers* typically support more than one system user, acting as their agent to the *Servers* that they represent. Security for these *Servers* is supported at two levels.

First, each UA Service request contains a string parameter that is used to carry an audit record id. A *Client*, or any *Server* operating as a *Client*, such as an aggregating *Server*, can create a local audit log entry for a request that it submits. This parameter allows the *Client* to pass the identifier for this entry with the request. If the *Server* also maintains an audit log, it can include this id in the audit log entry that it writes. When the log is examined and the entry is found, the examiner will be able to relate it directly to the audit log entry created by the *Client*. This capability allows for traceability across audit logs within a system. See [UA Part 2] for additional information on auditing. A *Server* that maintains an audit log must provide the audit log entries also via standard event Messages. The *Audit EventType* is defined in [UA Part 3].

Second, these aggregating *Servers* may open independent *Sessions* to the underlying *Servers* for each *Client* that accesses data from them. Figure 13 illustrates this concept.

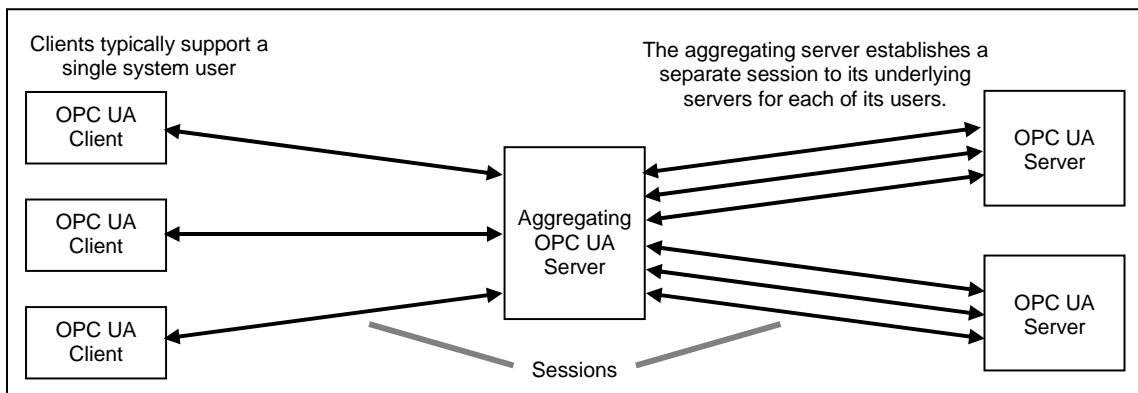


Figure 13 – Multiplexing Users on a Session

### 5.6.2.2 Parameters

Table 11 defines the parameters for the Service.

**Table 11 – CreateSession Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The <i>sessionId</i> is set to 0 in this request. The type <i>RequestHeader</i> is defined in Clause 7.19.
clientName	String	Displayable string that identifies the OPC <i>Client</i> application. The <i>Server</i> makes this name and the <i>sessionId</i> visible in its <i>AddressSpace</i> for diagnostic purposes. The <i>Client</i> should provide a name that is unique for the instance of the <i>Client</i> , to make diagnostic analysis possible in multi- <i>Client</i> environments. A unique name for an instance can be built, for example, by combining the application name, the node name of the system the <i>Client</i> is running on and the process Id.
clientNonce	ByteString	A random number that should never be used in any other request. This number must have a minimum length of 32 bytes. The <i>Server</i> must use this value to prove possession of its application instance <i>Certificate</i> in the response.
clientCertificate	ByteString	The application instance <i>Certificate</i> issued to the <i>Client</i> .
requestedSession Timeout	Duration	Requested maximum number of milliseconds that a <i>Session</i> should remain open without activity. If the <i>Client</i> fails to issue a <i>Service</i> request within this interval, then the <i>Server</i> will automatically terminate the <i>Client Session</i> .
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> type).
sessionId	IntegerId	<i>Server</i> -unique number that identifies the <i>Session</i> (see Clause 7.9 for <i>IntegerId</i> definition). This identifier must be assigned so that the <i>Server</i> can unambiguously identify the <i>Session</i> . The values used must not be reused such that the <i>Client</i> or the <i>Server</i> has a chance of confusing them with a previous or existing <i>Session</i> . 0 is an invalid <i>sessionId</i> .
revisedSession Timeout	Duration	Actual maximum number of milliseconds that a <i>Session</i> will remain open without activity (see Clause 7.6 for <i>Duration</i> definition). The <i>Server</i> should attempt to honour the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.
serverNonce	ByteString	A random number that should never be used in any other request. This number must have a minimum length of 32 bytes. The <i>Client</i> must use this value to prove possession of its application instance <i>Certificate</i> in the <i>ActivateSession</i> request. This value may also be used to prove possession of the <i>userIdentityToken</i> it specified in the <i>ActivateSession</i> request.
serverCertificate	ByteString	The application instance <i>Certificate</i> issued to the <i>Server</i> . A <i>Server</i> must prove possession by using the private key to sign the <i>Nonce</i> provided by the <i>Client</i> in the request.
serverSoftware Certificates []	SignedSoftware Certificate	These are the <i>SoftwareCertificates</i> which have been issued to the <i>Server</i> application. Each <i>SoftwareCertificate</i> has a signature created by the certification authority that issued it. The <i>Client</i> should check this signature to determine if the <i>SoftwareCertificates</i> are valid. The <i>SignedSoftwareCertificate</i> type is defined in Clause 7.25. This parameter is not specified if the <i>Server</i> does not have any <i>SoftwareCertificates</i> . Clients should call <i>CloseSession</i> if they are not satisfied with the <i>SoftwareCertificates</i> provided by the <i>Server</i> .
serverSignature	SignatureData	This is a signature generated with the private key associated with the <i>serverCertificate</i> . This parameter is calculated by appending the <i>clientNonce</i> to the <i>serverCertificate</i> and signing the resulting sequence of bytes. The <i>SignatureAlgorithm</i> must be the <i>asymmetricSignature</i> algorithm specified in the <i>SecurityPolicy</i> for the <i>Endpoint</i> . The <i>SignatureData</i> type is defined in Clause 7.25.

### 5.6.2.3 Service results

Table 12 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 12 – CreateSession Service Result Codes**

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 156 for the description of this result code.
Bad_NonceInvalid	See Table 156 for the description of this result code.
Bad_CertificateInvalid	See Table 156 for the description of this result code.
Bad_CertificateExpired	See Table 156 for the description of this result code.
Bad_CertificateRevoked	See Table 156 for the description of this result code.
Bad_CertificateUntrusted	See Table 156 for the description of this result code.
Bad_TooManySessions	The server has reached its maximum number of sessions.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.

### 5.6.3 ActivateSession

#### 5.6.3.1 Description

This Service is used by the *Client* to submit its *SoftwareCertificates* to the *Server* for validation and to specify the identity of the user associated with the *Session*. This Service request must be issued by the *Client* before it issues any other Service request after *CreateSession*. Failure to do so will cause the *Server* to close the *Session*.

Whenever the *Client* calls this Service the *Client* must prove that it is the same application that called the *CreateSession* Service. The *Client* does this by creating a signature with the private key associated with the *clientCertificate* specified in the *CreateSession* request. This signature is created by appending the last *serverNonce* provided by the *Server* to the *clientCertificate* and calculating the signature of the resulting sequence of bytes.

Once used, a *serverNonce* cannot be used again. For that reason, the *Server* returns a new *serverNonce* each time the *ActivateSession* Service is called.

When the *ActivateSession* Service is called for the first time then the *Server* must reject the request if the *SecureChannel* is not same as the one associated with the *CreateSession* request. Subsequent calls to *ActivateSession* may be associated with different *SecureChannels*. If this is the case then the *Server* must verify that the *Certificate* the *Client* used to create the new *SecureChannel* is the same as the *Certificate* used to create the original *SecureChannel*. In addition, the *Server* must verify that the *Client* supplied a *UserIdentityToken* that is identical to the token currently associated with the *Session*. Once the *Server* accepts the new *SecureChannel* it must reject requests sent via the old *SecureChannel*.

The *ActivateSession* Service is used to associate a user identity with a *Session*. When a *Client* provides a user identity then it must provide proof that is authorized to use that user identity. The exact mechanism used to provide this proof depends on the type of the *UserIdentityToken*. If the token is a *UserNameIdentityToken* then the proof is the *password* that included in the token. If the token is a *X509IdentityToken* then the proof is a signature generated with private key associated with the *Certificate*. The data to sign is created by appending the last *serverNonce* to the *clientCertificate* specified in the *CreateSession* request. Other types of tokens use a mechanism that depends on the token. If the token does not include a secret which proves possession of the token then the *Client* must create a signature with secret associated with the token using the same data to sign, that is used to prove possession of the *clientCertificate*.

*Clients* can change the identity of a user associated with a *Session* by calling the *ActivateSession* Service. The *Server* validates the signatures provided with the request and then validates the new user identity. If no errors occur the *Server* replaces the user identity for the *Session*. Changing the user identity for a *Session* may cause discontinuities in active *Subscriptions* because the *Server*

may have to tear down connections to underlying system and restablish them using the new credentials.

When a *Client* supplies a list of locale ids in the request, each locale id is required to contain the language component. It may optionally contain the <country/region> component. When the *Server* returns the response, it also may return both the language and the country/region or just the language as its default locale id.

When a *Server* returns a string to the *Client*, it first determines if there are available translations for it. If there are, the *Server* returns the string whose locale id exactly matches the locale id with the highest priority in the *Client*-supplied list.

If there are no exact matches, the *Server* ignores the <country/region> component of the locale id, and returns the string whose <language> component matches the <language> component of the locale id with the highest priority in the *Client* supplied list.

If there still are no matches, the *Server* returns the string that it has along with the locale id.

### 5.6.3.2 Parameters

Table 13 defines the parameters for the Service.

**Table 13 – ActivateSession Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters. The type <i>RequestHeader</i> is defined in Clause 7.19.
clientSignature	SignatureData	This is a signature generated with the private key associated with the <i>clientCertificate</i> . The <i>SignatureAlgorithm</i> must be the <i>asymmetricSignature</i> algorithm specified in the <i>SecurityPolicy</i> for the <i>Endpoint</i> . The <i>SignatureData</i> type is defined in Clause 7.25.
clientSoftwareCertificates []	SignedSoftwareCertificate	These are the <i>SoftwareCertificates</i> which have been issued to the <i>Client</i> application. Each <i>SoftwareCertificate</i> has a signature provided by certification authority that issued it. The <i>Server</i> should check this signature to determine if the <i>SoftwareCertificates</i> are valid. The <i>SignedSoftwareCertificate</i> type is defined in Clause 7.25. This parameter is not specified if the <i>Client</i> does not have any <i>SoftwareCertificates</i> . <i>Servers</i> may reject connections from <i>Clients</i> if they are not satisfied with the <i>SoftwareCertificates</i> provided by the <i>Client</i> . This parameter only needs to be specified during the first call to <i>ActivateSession</i> during a single application <i>Session</i> .
localeIds []	LocaleId	List of locale ids in priority order for localized strings. The first <i>localeId</i> in the list has the highest priority. If the <i>Server</i> returns a localized string to the <i>Client</i> , the <i>Server</i> will return the translation with the highest priority that it can. If it does not have a translation for any of the locales identified in this list, then it will return the string value that it has and include the locale id with the string. See [UA Part 3] for more detail on locale ids. If the <i>Client</i> fails to specify at least one locale id, the <i>Server</i> will use any that it has. This parameter only needs to be specified during the first call to <i>ActivateSession</i> during a single application <i>Session</i> . If it is not specified the <i>Server</i> will keep using the current <i>localeIds</i> for the <i>Session</i> .
userIdentityToken	Extensible Parameter UserIdentityToken	The credentials of the user associated with the <i>Client</i> application. The <i>Server</i> uses these credentials to determine whether the <i>Client</i> should be allowed to activate a <i>Session</i> and what resources the <i>Client</i> has access to during this <i>Session</i> . The <i>UserIdentityToken</i> is an extensible parameter type defined in Clause 8.7. The <i>EndpointDescription</i> specifies what <i>UserIdentityTokens</i> the <i>Server</i> will accept.
userTokenSignature	SignatureData	If the <i>Client</i> specified a user identity token that supports digital signatures, then it must create a signature and pass it as this parameter. The <i>SignatureAlgorithm</i> depends on the identity token type. The <i>SignatureData</i> type is defined in Clause 7.25.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
serverNonce	ByteString	A random number that should never be used in any other request. This number must have a minimum length of 32 bytes. The <i>Client</i> must use this value to prove possession of its application instance <i>Certificate</i> in the next call to <i>ActivateSession</i> request.
results []	StatusCodes	List of validation results for the <i>SoftwareCertificates</i> (see Clause 7.28 for <i>StatusCodes</i> definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information associated with <i>SoftwareCertificate</i> validation errors (see Clause 7.5 for <i>DiagnosticInfo</i> definition).

### 5.6.3.3 Service results

Table 14 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 14 – ActivateSession Service Result Codes**

Symbolic Id	Description
Bad_IdentityTokenInvalid	See Table 156 for the description of this result code.
Bad_IdentityTokenRejected	See Table 156 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_ApplicationSignatureInvalid	The signature provided by the client application is missing or invalid.
Bad_UserSignatureInvalid	The user token signature is missing or invalid.
Bad_NoValidCertificates	The Client did not provide at least one software certificate that is valid and meets the profile requirements for the Server.

### 5.6.4 CloseSession

#### 5.6.4.1 Description

This Service is used to terminate a Session. The Server takes the following actions when it receives a *CloseSession* request:

- a) It stops accepting requests for the Session. All subsequent requests received for the Session are discarded.
- b) It returns negative responses with the *StatusCodes* *Bad\_SessionClosed* to all requests that are currently outstanding to provide for the timely return of the *CloseSession* response. Clients are urged to wait for all outstanding requests to complete before submitting the *CloseSession* request.
- c) It removes the entry for the Client in its *SessionDiagnosticArray Variable*.

#### 5.6.4.2 Parameters

Table 15 defines the parameters for the Service.

**Table 15 – CloseSession Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).

#### 5.6.4.3 Service results

Table 16 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 16 – CloseSession Service Result Codes**

Symbolic Id	Description
Bad_SessionIdInvalid	See Table 156 for the description of this result code.

### 5.6.5 Cancel

#### 5.6.5.1 Description

This Service is used to cancel an outstanding Service request.

### 5.6.5.2 Parameters

Table 17 defines the parameters for the Service.

**Table 17 – Cancel Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
sequenceNumber	Counter	The sequence number given by the <i>Client</i> to the request that should be canceled. The <i>Client</i> must assign unique sequence numbers to each service request if he wants to cancel service requests.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).

### 5.6.5.3 Service results

Table 18 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 18 – Cancel Service Result Codes**

Symbolic Id	Description
Bad_SequenceNumberInvalid	The sequence number was invalid or the service response was already sent.

## 5.7 NodeManagement Service Set

### 5.7.1 Overview

This Service Set defines Services to add and delete *AddressSpace Nodes* and *References* between them. All added *Nodes* continue to exist in the *AddressSpace* even if the *Client* that created them disconnects from the *Server*.

In the Services that follow, many of the *NodeIds* are represented by *ExpandedNodeIds*. *ExpandedNodeIds* identify the namespace by their string name rather than by their *NamespaceTable* index. This allows the *Server* to add the namespace to its *NamespaceTable* if necessary.

### 5.7.2 AddNodes

#### 5.7.2.1 Description

This Service is used to add one or more *Nodes* into the *AddressSpace* hierarchy. Using this Service, each *Node* is added as the *TargetNode* of a *HierarchicalReference* to ensure that the *AddressSpace* is fully connected and that the *Node* is added as a child within the *AddressSpace* hierarchy (see [UA Part 3]).

### 5.7.2.2 Parameters

Table 19 defines the parameters for the Service.

**Table 19 – AddNodes Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
nodesToAdd []	AddNodesItem	List of <i>Nodes</i> to add. All <i>Nodes</i> are added as a <i>Reference</i> to an existing <i>Node</i> using a hierarchical <i>ReferenceType</i> .
parentNodeld	Expanded Nodeld	Expanded <i>Nodeld</i> of the parent <i>Node</i> for the <i>Reference</i> . The <i>ExpandedNodeld</i> type is defined in Clause 7.7.
referenceTypeld	Nodeld	<i>Nodeld</i> of the hierarchical <i>ReferenceType</i> to use for the <i>Reference</i> from the parent <i>Node</i> to the new <i>Node</i> .
requestedNewNodeld	Expanded Nodeld	<i>Client</i> requested expanded <i>Nodeld</i> of the <i>Node</i> to add. The <i>serverIndex</i> in the expanded <i>Nodeld</i> must be 0. If the <i>Server</i> cannot use this <i>Nodeld</i> , it rejects this <i>Node</i> and returns the appropriate error code. If the <i>Client</i> does not want to request a <i>Nodeld</i> , then it sets the value of this parameter to the null expanded <i>Nodeld</i> . If the <i>Node</i> to add is a <i>ReferenceType Node</i> , its <i>Nodeld</i> must be a numeric id. See [UA Part 3] for a description of <i>ReferenceType Nodelds</i> .
browseName	QualifiedName	The browse name of the <i>Node</i> to add.
nodeClass	NodeClass	<i>NodeClass</i> of the <i>Node</i> to add.
nodeAttributes	Extensible Parameter NodeAttributes	The <i>Attributes</i> that are specific to the <i>NodeClass</i> . The <i>NodeAttributes</i> parameter type is an extensible parameter type specified in Clause 8.6. The <i>Client</i> must provide a value for each mandatory attribute. If an optional attribute is not set by the <i>Client</i> , the <i>Server</i> behaviour is vendor specific.
typeDefinition	Expanded Nodeld	<i>Nodeld</i> of the <i>TypeDefinitionNode</i> for the <i>Node</i> to add. This parameter must be null for all <i>NodeClasses</i> other than <i>Object</i> and <i>Variable</i> in which case it must be provided.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	AddNodesResult	List of results for the <i>Nodes</i> to add. The size and order of the list matches the size and order of the <i>nodesToAdd</i> request parameter.
statusCode	StatusCode	<i>StatusCodes</i> for the <i>Node</i> to add (see Clause 7.28 for <i>StatusCodes</i> definition).
addedNodeld	Nodeld	<i>Server</i> assigned <i>Nodeld</i> of the added <i>Node</i> . Null <i>Nodeld</i> if the operation failed.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Nodes</i> to add (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>nodesToAdd</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.7.2.3 Service results

Table 20 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 20 – AddNodes Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.7.2.4 StatusCodes

Table 21 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 21 – AddNodes Operation Level Result Codes**

Symbolic Id	Description
Bad_ParentNodeIdInvalid	The parent node id does not refer to a valid node.
Bad_ReferenceTypeNodeIdInvalid	The reference type id does not refer to a valid reference type node.
Bad_ReferenceNotAllowed	The reference could not be created because it violates constraints imposed by the data model.
Bad_NodeIdRejected	The requested node id was rejected either because it was invalid or because the server does not allow node ids to be specified by the client.
Bad_NodeIdExists	The requested node id is already used by another node.
Bad_NodeClassInvalid	The node class is not valid.
Bad_BrowseNameInvalid	The browse name is invalid.
Bad_BrowseNameDuplicated	The browse name is not unique among nodes that share the same relationship with the parent.
Bad_NodeAttributesInvalid	The node <i>Attributes</i> are not valid for the node class.
Bad_TypeDefinitionInvalid	The type definition node id does not reference an appropriate type node.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.

### 5.7.3 AddReferences

#### 5.7.3.1 Description

This Service is used to add one or more *References* to one or more *Nodes*. The *NodeClass* is an input parameter that is used to validate that the *Reference* to be added matches the *NodeClass* of the *TargetNode*. This parameter is not validated if the *Reference* refers to a *TargetNode* in a remote Server.

In certain cases, adding new *References* to the *AddressSpace* will require that the Server add new Server ids to the Server's *ServerTable Variable*. For this reason, remote Servers are identified by their URI and not by their *ServerTable* index. This allows the Server to add the remote Server URIs to its *ServerTable*.

### 5.7.3.2 Parameters

Table 22 defines the parameters for the Service.

**Table 22 – AddReferences Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
referencesToAdd []	AddReferences Item	List of <i>Reference</i> instances to add to the <i>SourceNode</i> . The <i>targetNodeClass</i> of each <i>Reference</i> in the list must match the <i>NodeClass</i> of the <i>TargetNode</i> .
sourceNodeId	NodeId	<i>NodeId</i> of the <i>Node</i> to which the <i>Reference</i> is to be added. The source <i>Node</i> must always exist in the <i>Server</i> to add the <i>Reference</i> .
referenceTypeld	NodeId	<i>NodeId</i> of the <i>ReferenceType</i> that defines the <i>Reference</i> .
isForward	Boolean	If the value is TRUE, the <i>Server</i> creates a forward <i>Reference</i> . If the value is FALSE, the <i>Server</i> creates an inverse <i>Reference</i> .
targetServerUri	String	URI of the remote <i>Server</i> .
targetNodeId	Expanded NodeId	Expanded <i>NodeId</i> of the <i>TargetNode</i> . The <i>ExpandedNodeId</i> type is defined in Clause 7.7.
targetNodeClass	NodeClass	<i>NodeClass</i> of the <i>TargetNode</i> . The <i>Client</i> must specify this since the <i>TargetNode</i> might not be accessible directly by the <i>Server</i> .
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>References</i> to add (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>referencesToAdd</i> request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the <i>References</i> to add (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>referencesToAdd</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.7.3.3 Service results

Table 23 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 23 – AddReferences Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.7.3.4 StatusCodes

Table 24 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 24 – AddReferences Operation Level Result Codes**

Symbolic Id	Description
Bad_SourceNodeIdInvalid	The source node id does not refer to a valid node.
Bad_ReferenceTypeldInvalid	The reference type id does not refer to a valid reference type node.
Bad_ServerUriInvalid	The <i>Server URI</i> is not valid.
Bad_TargetNodeIdInvalid	The target node id does not refer to a valid node.
Bad_NodeClassInvalid	The node class is not valid.
Bad_ReferenceNotAllowed	The reference could not be created because it violates constraints imposed by the data model on this server.
Bad_ReferenceLocalOnly	The reference type is not valid for a reference to a remote <i>Server</i> .
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_DuplicateReferenceNotAllowed	The reference type between the nodes is already defined.
Bad_InvalidSelfReference	The server does not allow this type of self reference on this node.

## 5.7.4 DeleteNodes

### 5.7.4.1 Description

This Service is used to delete one or more *Nodes* from the *AddressSpace*. If the *Node* to be deleted is the owner of another *Node* then this *Node* is deleted too. The ownership is defined by the *ModellingRules* specified in [UA Part 3].

When any of the *Nodes* deleted by an invocation of this Service is the *TargetNode* of a *Reference*, then those *References* are left unresolved based on the *deleteTargetReference* parameter.

When any of the *Nodes* deleted by an invocation of this Service is contained in a *View*, then the *ViewVersion Property* is updated if this *Property* is supported.

When any of the *Nodes* deleted by an invocation of this Service is being monitored, then a *Notification* is sent to the monitoring *Client* indicating that the *Node* has been deleted.

### 5.7.4.2 Parameters

Table 25 defines the parameters for the Service.

**Table 25 – DeleteNodes Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
nodesToDelete []	DeleteNodes Item	List of <i>Nodes</i> to delete
nodeId	NodeId	<i>NodeId</i> of the <i>Node</i> to delete.
deleteTargetReference	Boolean	A Boolean parameter with the following values : TRUE delete <i>References</i> in <i>TargetNodes</i> that Reference the <i>Node</i> to delete. FALSE delete only the <i>References</i> for which the <i>Node</i> to delete is the source.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>Nodes</i> to delete (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the list of the <i>nodesToDelete</i> request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the <i>Nodes</i> to delete (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>nodesToDelete</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.7.4.3 Service results

Table 26 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 26 – DeleteNodes Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.7.4.4 StatusCodes

Table 27 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 27 – DeleteNodes Operation Level Result Codes**

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_NoDeleteRights	The Server will not allow the node to be deleted.

### 5.7.5 DeleteReferences

#### 5.7.5.1 Description

This Service is used to delete one or more *References* of a *Node*.

When any of the *References* deleted by an invocation of this Service is contained in a *View*, then the *ViewVersion Property* is updated if this *Property* is supported.

The deletion of a Reference will trigger a *ModelChange Event*.

#### 5.7.5.2 Parameters

Table 25 defines the parameters for the Service.

**Table 28 – DeleteReferences Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
referencesToDelete []	DeleteReferences Item	List of <i>References</i> to delete.
sourceNodeId	NodeId	<i>NodeId</i> of the <i>Node</i> that contains the <i>Reference</i> to delete.
referenceTypeId	NodeId	<i>NodeId</i> of the <i>ReferenceType</i> that defines the <i>Reference</i> to delete.
serverIndex	Index	Index that identifies the <i>Server</i> that contains the <i>TargetNode</i> . This <i>Server</i> may be the local <i>Server</i> or a remote <i>Server</i> . This index is the index of that <i>Server</i> in the local <i>Server's Server table</i> . The index of the local <i>Server</i> in the <i>Server table</i> is always 0. All remote <i>Servers</i> have indexes greater than 0. The <i>Server table</i> is contained in the <i>Server Object</i> in the <i>AddressSpace</i> (see [UA Part 5]). The <i>Client</i> may read the <i>Server table Variable</i> to access the description of the target <i>Server</i>
targetNodeId	ExpandedNodeId	<i>NodeId</i> of the <i>TargetNode</i> of the <i>Reference</i> . If the <i>Server</i> index indicates that the <i>TargetNode</i> is a remote <i>Node</i> , then the <i>nodeId</i> must contain the absolute namespace URI. If the <i>TargetNode</i> is a local <i>Node</i> the <i>nodeId</i> must contain the namespace index.
deleteBidirectional	Boolean	A Boolean parameter with the following values : TRUE delete the specified <i>Reference</i> and the opposite <i>Reference</i> from the <i>TargetNode</i> . If the <i>TargetNode</i> is located in a remote <i>Server</i> , the <i>Server</i> is permitted to delete the specified <i>Reference</i> only. FALSE delete only the specified <i>Reference</i> .
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>References</i> to delete (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>referencesToDelete</i> request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the <i>References</i> to delete (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>referencesToDelete</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.7.5.3 Service results

Table 29 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 29 – DeleteReferences Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

#### 5.7.5.4 StatusCodes

Table 30 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 30 – DeleteReferences Operation Level Result Codes**

Symbolic Id	Description
Bad_SourceNodeidInvalid	The source node id does not refer to a valid node.
Bad_ReferenceTypeidInvalid	The reference type id does not refer to a valid reference type node.
Bad_ServerIndexInvalid	The server index is not valid.
Bad_TargetNodeidInvalid	The target node id does not refer to a valid node.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_NoDeleteRights	The server will not allow the reference to be deleted.

### 5.8 View Service Set

#### 5.8.1 Overview

A *View* is a subset of the *AddressSpace* created by the *Server*. Future versions of this specification may also define services to create *Client-defined Views*. *Views* are represented in the *AddressSpace* by *View Nodes* that are referenced by the standard *Views Folder* using *Organizes References*. *Clients* can use the *browse Service* to identify the structure of the *View* and the *Nodes* contained in it. See [UA Part 5] for a description of the organisation of views in the *AddressSpace*.

*Clients* use the *browse Services* of the *View Service Set* to navigate through the *AddressSpace* or through a *View* as subset of the *AddressSpace*.

#### 5.8.2 BrowseProperties

##### 5.8.2.1 Description

This Service is used to discover the *Properties* supported by one or more specified *Nodes*. The *read Service* can be used to access the *Property* values.

### 5.8.2.2 Parameters

Table 31 defines the parameters for the Service.

**Table 31 – BrowseProperties Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
nodesToAccess []	NodeId	List of <i>NodeIds</i> of the <i>Nodes</i> whose <i>Properties</i> are to be accessed.
properties[]	QualifiedName	List of <i>Node Properties</i> to return. If set to null, then all <i>Properties</i> are returned.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for the type definition).
results []	BrowsePropertiesResult	List of results for the <i>Nodes</i> to access. The size and order of the list matches the size and order of the <i>nodesToAccess</i> request parameter.
statusCode	StatusCode	<i>StatusCodes</i> for the <i>Node</i> to access (see Clause 7.28 for <i>StatusCodes</i> definition).
propertyResults []	BrowsePropertiesPropertyResult	List of results for the <i>Node</i> to access. The size and order of the list matches the size and order of the <i>properties</i> parameter if the requested list was not empty. Otherwise the list contains all <i>Properties</i> of the browsed <i>Node</i> .
propertyName	QualifiedName	The name of the returned <i>Property</i> .
propertyDisplayName	LocalizedText	The <i>DisplayName</i> of the <i>Property</i> .
propertyNodeId	NodeId	The <i>NodeId</i> of the returned <i>Property</i> .
propertyStatusCode	StatusCodes	<i>StatusCodes</i> for the returned <i>Property</i> .
diagnosticInfos []	DiagnosticInfo	A list of diagnostic information for a <i>Property</i> being read. This list is omitted if diagnostics information was not requested. If present, this list must have the same length as the list of <i>Property</i> results.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Nodes</i> to access. The size and order of the list matches the size and order of the <i>nodesToAccess</i> request parameter. This list is empty if diagnostics information was not requested in the request header (see Clause 7.5 for <i>DiagnosticInfo</i> definition).

### 5.8.2.3 Service results

Table 32 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 32 – BrowseProperties Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.8.2.4 StatusCodes

Table 33 defines values for the operation level *statusCode* and *propertyStatusCode* parameters that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 33 – BrowseProperties Operation Level Result Codes**

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_PropertyNameUnknown	The property name is not known by the server.

## 5.8.3 Browse

### 5.8.3.1 Description

This Service is used to discover the *References* of a specified *Node*. The browse can be further limited by the use of a *View*. This Browse Service also supports a primitive filtering capability.

### 5.8.3.2 Parameters

Table 34 defines the parameters for the Service.

**Table 34 – Browse Service Parameters**

Name	Type	Description																		
<b>Request</b>																				
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).																		
view	ViewDescription	Description of the <i>View</i> to browse (see Clause 7.31 for <i>ViewDescription</i> definition). A null <i>ViewDescription</i> value indicates the entire <i>AddressSpace</i> . Use of the null <i>ViewDescription</i> value causes all <i>References</i> of the <i>nodeToBrowse</i> to be returned. Use of any other <i>View</i> causes only the <i>References</i> of the <i>nodeToBrowse</i> that are defined for that <i>View</i> to be returned.																		
nodeToBrowse	NodeID	<i>NodeID</i> of the <i>Node</i> to be browsed. The passed <i>nodeToBrowse</i> must be part of the passed <i>view</i> .																		
maxReferences ToReturn	Counter	Indicates the maximum number of references to return. The value 0 indicates that the <i>Client</i> is imposing no limitation (see Clause 7.3 for <i>Counter</i> definition).																		
browseDirection	enum BrowseDirection	An enumeration that specifies the direction of <i>References</i> to follow. It has the following values : FORWARD_0 select only forward <i>References</i> . INVERSE_1 select only inverse <i>References</i> . BOTH_2 select forward and inverse <i>References</i> . The returned <i>References</i> do indicate the direction the <i>Server</i> followed in the <i>isForward</i> parameter of the <i>ReferenceDescription</i> .																		
referenceTypeld	NodeID	Specifies the <i>NodeID</i> of the <i>ReferenceType</i> to follow. Only instances of this <i>ReferenceType</i> or its subtypes are returned. If not specified then all <i>References</i> are returned.																		
includeSubtypes	Boolean	Indicates whether subtypes of the <i>ReferenceType</i> should be included in the browse. If TRUE, then instances of <i>referenceTypeld</i> and all of its subtypes are returned.																		
nodeClassMask	UInt32	Specifies the <i>NodeClasses</i> of the <i>TargetNodes</i> . Only <i>TargetNodes</i> with the selected <i>NodeClasses</i> are returned. The <i>NodeClasses</i> are assigned the following bits: <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>Bit</th> <th>NodeClass</th> </tr> <tr> <td>0</td> <td>Object</td> </tr> <tr> <td>1</td> <td>Variable</td> </tr> <tr> <td>2</td> <td>Method</td> </tr> <tr> <td>3</td> <td>ObjectType</td> </tr> <tr> <td>4</td> <td>VariableType</td> </tr> <tr> <td>5</td> <td>ReferenceType</td> </tr> <tr> <td>6</td> <td>DataType</td> </tr> <tr> <td>7</td> <td>View</td> </tr> </table> If set to zero, then all <i>NodeClasses</i> are returned.	Bit	NodeClass	0	Object	1	Variable	2	Method	3	ObjectType	4	VariableType	5	ReferenceType	6	DataType	7	View
Bit	NodeClass																			
0	Object																			
1	Variable																			
2	Method																			
3	ObjectType																			
4	VariableType																			
5	ReferenceType																			
6	DataType																			
7	View																			
<b>Response</b>																				
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).																		
continuationPoint	ByteString	Server defined opaque value that identifies the continuation point. The continuation point is used only when the browse results are too large to be contained in a single response. "Too large" in this context means that the <i>Server</i> is not able to return a larger response, or the number of <i>References</i> to return exceeds the maximum number of <i>References</i> to return that was specified by the <i>Client</i> in the request. The continuation point is used in the <i>BrowseNext Service</i> . When not used, the value of this parameter is null. If a continuation point is returned the <i>Client</i> must call <i>BrowseNext</i> to get the next set of browse information or to free the resources for the continuation point in the <i>Server</i> . Servers must support at least one continuation point per <i>Session</i> . Servers specify a max continuation points per <i>Session</i> in the <i>ServerCapabilities Object</i> defined in [UA Part 5]. A continuation point will remain active until the <i>Client</i> passes the continuation point to <i>BrowseNext</i> or the <i>Session</i> is closed. If the max continuation points have been reached the least recently used continuation point will be reset.																		
references []	Reference Description	List of <i>References</i> selected for the browsed <i>Node</i> . Empty, if no <i>References</i> met the browse direction or <i>Reference</i> filter criteria. The <i>Reference Description</i> type is defined in Clause 7.17.																		

### 5.8.3.3 Service results

Table 35 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 35 – Browse Service Result Codes**

Symbolic Id	Description
Bad_ViewIdUnknown	The view id does not refer to a valid view <i>Node</i> .
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_ReferenceTypeIdInvalid	The reference type id does not refer to a valid reference type node.
Bad_BrowseDirectionInvalid	The browse direction is not valid.
Bad_NodeNotInView	The nodeToBrowse is not part of the view.

### 5.8.4 BrowseNext

#### 5.8.4.1 Description

This Service is used to request the next set of *Browse* or *BrowseNext* response information that is too large to be sent in a single response. “Too large” in this context means that the *Server* is not able to return a larger response or that the number of results to return exceeds the maximum number of results to return that was specified by the *Client* in the original *Browse* request. The *BrowseNext* must be submitted on the same *Session* that was used to submit the *Browse* or *BrowseNext* that is being continued.

#### 5.8.4.2 Parameters

Table 36 defines the parameters for the Service.

**Table 36 – BrowseNext Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
releaseContinuationPoint	Boolean	A <i>Boolean</i> parameter with the following values : TRUE      passed <i>continuationPoint</i> will be reset to free resources for the continuation point in the <i>Server</i> . FALSE     passed <i>continuationPoint</i> will be used to get the next set of browse information. A <i>Client</i> must always use the continuation point returned by a <i>Browse</i> or <i>BrowseNext</i> response to free the resources for the continuation point in the <i>Server</i> . If the <i>Client</i> does not want to get the next set of browse information, <i>BrowseNext</i> must be called with this parameter set to TRUE.
continuationPoint	ByteString	Server-defined opaque value that represents the continuation point. The value of the continuation point was returned to the <i>Client</i> in a previous <i>Browse</i> or <i>BrowseNext</i> response. This value is used to identify the previously processed <i>Browse</i> or <i>BrowseNext</i> request that is being continued and the point in the result set from which the browse response is to continue.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
revisedContinuationPoint	ByteString	Server-defined opaque value that represents the continuation point. It is used only if the information to be returned is too large to be contained in a single response. When not used or when <i>releaseContinuationPoint</i> is set, the value of this parameter is null.
references []	Reference Description	List of <i>References</i> selected for the browsed <i>Node</i> . Empty, if no <i>References</i> met the browse direction or Reference filter criteria. The Reference Description type is defined in Clause 7.17. When <i>releaseContinuationPoint</i> is set this list is empty.

### 5.8.4.3 Service results

Table 37 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 37 – BrowseNext Service Result Codes**

Symbolic Id	Description
Bad_ContinuationPointInvalid	See Table 156 for the description of this result code.

### 5.8.5 TranslateBrowsePathsToNodeIds

#### 5.8.5.1 Description

This Service is used to request the Server to translate one or more browse paths to *NodeIds*. Each browse path is constructed of a starting *Node* and a *RelativePath*. The specified starting *Node* identifies the *Node* from which the *RelativePath* is based. The *RelativePath* contains a sequence of *BrowseNames*. The final *BrowseName* in the path is either the *BrowseName* for a *ReferenceType* or a *BrowseName* for a *Node*. If the final *BrowseName* refers to a *ReferenceType* then the path matches all targets of that *Reference*. If the final *BrowseName* refers to a *Node* then the path matches all targets with the specified *BrowseName*.

The syntax of the *RelativePath* is described in Clause 7.18.

One purpose of this Service is to allow programming against type definitions. Since *BrowseNames* must be unique in the context of type definitions, a *Client* may create a browse path that is valid for a type definition and use this path on instances of the type. For example, an *ObjectType* “Boiler” may have a “HeatSensor” *Variable* as *InstanceDeclaration*. A graphical element programmed against the “Boiler” may need to display the *Value* of the “HeatSensor”. If the graphical element would be called on “Boiler1”, an instance of “Boiler”, it would need to call this Service specifying the *NodeId* of “Boiler1” as starting *Node* and the *BrowseName* of the “HeatSensor” as browse path. The Service would return the *NodeId* of the “HeatSensor” of “Boiler1” and the graphical element could subscribe to its *Value Attribute*.

If a *Node* has multiple targets with the same *BrowseName*, the Server will return a list of *NodeIds*. However, since one purpose of this Service is to support programming against type definitions, the *NodeId* of the *Node* based on the type definition of the starting *Node* is returned first in the list. If no *NameSpaceId* is provided for a *BrowseName* in the browse path, the *NodeId* might not be unique in the context of the type definition. In that case, all applicable *NodeIds* based on the type definition are returned first in the list. In that case, the *NameSpaceId* describes the order of those *Nodes*, the *NodeId* with the smallest *NameSpaceId* in its *BrowseName* occurs first in the list.

### 5.8.5.2 Parameters

Table 38 defines the parameters for the Service.

**Table 38 – TranslateBrowsePathsToNodeIds Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
browsePaths []	BrowsePath	List of browse paths for which <i>NodeIds</i> are being requested.
startingNode	NodeId	<i>NodeId</i> of the starting <i>Node</i> for the browse path.
relativePath	RelativePath	Browse path relative to the starting <i>Node</i> identifies the <i>Node</i> for which the <i>NodeId</i> is being requested. See Clause 7.18 for the definition of <i>RelativePath</i> .
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	TranslateBrowsePathResult	List of results for the list of browse paths. The size and order of the list matches the size and order of the <i>browsePaths</i> request parameter.
statusCode	StatusCodes	<i>StatusCodes</i> for the browse path (see Clause 7.28 for <i>StatusCodes</i> definition).
matchingNodeIds []	NodeId	List of <i>NodeIds</i> of the <i>Nodes</i> which can be accessed via the browse path specified in the request. Servers that have hierarchies without duplicate <i>BrowseNames</i> will always return an array with one <i>NodeId</i> in this parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the list of browse paths (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>browsePaths</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.8.5.3 Service results

Table 39 defines the Service results specific to this Service. Common *StatusCodes* are defined in Clause 7.28.

**Table 39 – TranslateBrowsePathsToNodeIds Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.8.5.4 StatusCodes

Table 40 defines values for the operation level *statusCode* parameters that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 40 – TranslateBrowsePathsToNodeIds Operation Level Result Codes**

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_ReferenceOutOfServer	One of the references to follow in the relative path refers to a node in the address space in another server.
Bad_BrowsePathInvalid	The specified browse path is invalid, e.g. no browse name is specified after specifying which reference to follow.
Bad_TooManyMatches	The requested operation has too many matches to return. Users should use queries for large result sets. Servers should allow at least 10 matches before returning this error code.
Bad_QueryTooComplex	The requested operation requires too many resources in the server.
Bad_NoMatch	The requested operation has no match to return.

## 5.9 Query Service Set

### 5.9.1 Overview

This Service Set is used to issue a *Query* to a Server. UA *Query* is generic in that it provides an underlying storage mechanism independent *Query* capability that can be used to access a wide variety of UA data stores and information management systems. UA *Query* permits a *Client* to

access data maintained by a *Server* without any knowledge of the logical schema used for internal storage of the data. Knowledge of the *AddressSpace* is sufficient.

An *OPC UA Application* is expected to use the *UA Query Services* as part of an initialization process or an occasional information synchronization step. For example, *UA Query* would be used for bulk data access of a persistent store to initialise an analysis application with the current state of a system configuration. A *Query* may also be used to initialise or populate data for a report.

A *Query* defines what instances of one or more *TypeDefinitionNodes* in the *AddressSpace* should supply a set of *Attributes*. Results returned by a *Server* are in the form of an array of *QueryDataSets*. The selected *Attribute* values in each *QueryDataSet* come from the definition of the selected *TypeDefinitionNodes* or related *TypeDefinitionNodes* and appear in results in the same order as the *Attributes* that were passed into the *Query*. *Query* also supports *Node* filtering on the basis of *Attribute* values, as well as relationships between *TypeDefinitionNodes*.

### 5.9.2 Querying Views

A *View* is a subset of the *AddressSpace* available in the *Server*. See [UA Part 5] for a description of the organisation of *Views* in the *AddressSpace*.

For any existing *View*, a *Query* may be used to return a subset of data from the *View*. When an application issues a *Query* against a *View*, only data defined by the *View* is returned. Data not included in the *View* but included in the original *AddressSpace* is not returned.

The *Query Services* supports access to current and historical data. The *Service* supports a *Client* querying a past version of the *AddressSpace*. Clients may specify a *ViewVersion* or a *Timestamp* in a *Query* to access past versions of the *AddressSpace*. *UA Query* is complementary to Historical Access in that the former is used to *Query* an *AddressSpace* that existed at a time and the latter is used to *Query* for the value of *Attributes* over time. In this way, a *Query* can be used to retrieve a portion of a past *AddressSpace* so that *Attribute* value history may be accessed using Historical Access even if the *Node* is no longer in the current *AddressSpace*.

Servers that support *Query* are expect to be able to access the address space that is associated with the local *Server* and any *Views* that are available on the local *Server*. If a *View* or the address space also references a remote *Server*, query may be able to access the address space of remote *Server*, but it is not required. If a *Server* does access a remote *Server* the access must be accomplished using the user identity of the *Client* as described in Clause 5.5.1.

### 5.9.3 QueryFirst

#### 5.9.3.1 Description

This *Service* is used to issue a *Query* request to the *Server*. The complexity of the *Query* can range from very simple to highly sophisticated. The *Query* can simply request data from instances of a *TypeDefinitionNode* or *TypeDefinitionNode* subject to restrictions specified by the filter. On the other hand, the *Query* can request data from instances of related *Node* types by specifying a *RelativePath* from an originating *TypeDefinitionNode*. In the filter, a separate set of paths can be constructed for limiting the instances that supply data. A filtering path can include multiple *RelatedTo* operators to define a multi-hop path between source instances and target instances. For example, one could filter on students that attend a particular school, but return information about students and their families. In this case, the student school relationship is traversed for filtering, but the student family relationship is traversed to select data. For a complete description of *ContentFilter* see Clause 7.2, also see Clause 5.9.5 for additional examples of content filter and queries.

The *Client* provides an array of *NodeTypeDescription* which specify the *NodeId* of a *TypeDefinitionNode* and selects what *Attributes* are to be returned in the response. A client can also provide a set of *RelativePaths* through the type system starting from an originating *TypeDefinitionNode*. Using these paths, the client selects a set of *Attributes* from *Nodes* that are related to instances of the originating *TypeDefinitionNode*. Additionally, the *Client* can request the

Server return instances of subtypes of *TypeDefinitionNodes*. If a selected *Attribute* does not exist in a *TypeDefinitionNode* but does exist in a subtype, it is assumed to have a null value in the *TypeDefinitionNode* in question. Therefore, this does not constitute an error condition and a null value is returned for the *Attribute*.

The *Client* can use the filter parameter to limit the result set by restricting *Attributes* and *Properties* to certain values. Another way the *Client* can use a filter to limit the result set is by specifying how instances should be related, using *RelatedTo* operators. In this case, if an instance at the top of the *RelatedTo* path cannot be followed to the bottom of the path via specified hops, no *QueryDataSets* are returned for the starting instance or any of the intermediate instances.

When querying for related instances in the *RelativePath*, the *Client* can optionally ask for *References*. A *Reference* is requested via a *RelativePath* that only includes a *ReferenceType*. If all *References* are desired then the root *ReferenceType* would be listed. These *Reference* are returned as part of the *QueryDataSets*.

### 5.9.3.2 Parameters

Table 41 defines the request parameters and Table 42 the response parameters for the *QueryFirst Service*.

**Table 41 – QueryFirst Request Parameters**

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
view	ViewDescription	Specifies a <i>View</i> and temporal context to a <i>Server</i> (see Clause 7.31 for <i>ViewDescription</i> definition).
nodeTypes[]	NodeTypeDescription	This is the <i>Node</i> type description.
typeDefinitionNode	ExpandedNodeID	<i>NodeID</i> of the originating <i>TypeDefinitionNode</i> of the instances for which data is to be returned.
includeSubtypes	Boolean	A flag that indicates whether the <i>Server</i> should include instances of subtypes of the <i>TypeDefinitionNode</i> in the list of instances of the <i>Node</i> type.
dataToReturn[]	DataDescription	Specifies an <i>Attribute</i> or <i>Reference</i> from the originating <i>typeDefinitionNode</i> along a given <i>relativePath</i> for which to return data.
relativePath	RelativePath	Browse path relative to the originating <i>Node</i> that identifies the <i>Node</i> which contains the data that is being requested, where the originating <i>Node</i> is an instance <i>Node</i> of the type defined by the type definition <i>Node</i> . The instance <i>Nodes</i> are further limited by the filter provided as part of this call. For a definition of <i>relativePath</i> see Clause 7.18. This relative path could end on a <i>Reference</i> , in which case the <i>ReferenceDescription</i> of the <i>Reference</i> would be returned as its value.
attributeld	IntegerId	<i>Id</i> of the <i>Attribute</i> . This must be a valid <i>Attribute Id</i> . The <i>IntegerId</i> is defined in Clause 7.9. The <i>IntegerIds</i> for the <i>Attributes</i> are defined in [UA Part 6]. If the <i>RelativePath</i> ended in a <i>Reference</i> then this parameter is 0 and ignored by the server.
indexRange	NumericRange	This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. If a range of elements are specified, the values are returned as a composite. The first element is identified by index 0 (zero). The <i>NumericRange</i> type is defined in Clause 7.14. This parameter is null if the specified <i>Attribute</i> is not an array or a structure. However, if the specified <i>Attribute</i> is an array or a structure, and this parameter is null, then all elements are to be included in the range.
filter	ContentFilter	Resulting <i>Nodes</i> will be limited to the <i>Nodes</i> matching the criteria defined by the filter. ContentFilter is discussed in Clause 7.2 and Clause 8.4. If an empty filter is provided then the entire address space will be examined and all <i>Nodes</i> that contain a matching requested <i>Attribute</i> or <i>Reference</i> are returned. A value of 0 indicates that the <i>Client</i> is imposing no limitation.
maxDataSetsToReturn	Counter	The number of <i>QueryDataSets</i> that the <i>Client</i> wants the <i>Server</i> to return in the response and on each subsequent continuation call response. The <i>Server</i> is allowed to further limit the response, but must not exceed this limit. A value of 0 indicates that the <i>Client</i> is imposing no limitation.
maxReferencesToReturn	Counter	The number of <i>References</i> that the <i>Client</i> wants the <i>Server</i> to return in the response for each <i>QueryDataSet</i> and on each subsequent continuation call response. The <i>Server</i> is allowed to further limit the response, but must not exceed this limit. A value of 0 indicates that the <i>Client</i> is imposing no limitation. For example a result where 4 <i>Nodes</i> are being returned, but each has 100 <i>References</i> , if this limit were set to 50 then only the first 50 <i>References</i> for each <i>Node</i> would be returned on the initial call and a continuation point would be set indicating additional data.

**Table 42 – QueryFirst Response Parameters**

Name	Type	Description
Response		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
queryDataSets []	QueryDataSet	The array of <i>QueryDataSet</i> . This array is empty if no <i>Nodes</i> met the <i>nodeTypes</i> criteria. In this case the <i>continuationPoint</i> parameter must be empty. The <i>QueryDataSet</i> type is defined in Clause 7.15.
continuationPoint	ByteString	Server-defined opaque value that identifies the continuation point. The continuation point is used only when the <i>Query</i> results are too large to be returned in a single response. "Too large" in this context means that the Server is not able to return a larger response or that the number of <i>QueryDataSets</i> to return exceeds the maximum number of <i>QueryDataSets</i> to return that was specified by the <i>Client</i> in the request. The continuation point is used in the <i>QueryNext Service</i> . When not used, the value of this parameter is null. If a continuation point is returned, the <i>Client</i> must call <i>QueryNext</i> to get the next set of <i>QueryDataSets</i> or to free the resources for the continuation point in the Server. Servers must support at least one <i>Query</i> continuation point per session. Servers specify a max continuation points per session in <i>Server capabilities Object</i> defined in [UA Part 5]. A continuation point will remain active until the <i>Client</i> passes the continuation point to <i>QueryNext</i> or the session is closed. If the max continuation points have been reached the oldest continuation point will be reset.
results[]	QueryResult	List of results for <i>QueryFirst</i> . The size and order of the list matches the size and order of the <i>NodeTypes</i> request parameter.
statusCode	StatusCodes	Result for the requested <i>NodeTypeDescription</i> .
dataStatusCodes []	StatusCodes	List of results for <i>dataToReturn</i> . The size and order of the list matches the size and order of the <i>dataToReturn</i> request parameter.
dataDiagnosticInfos []	DiagnosticInfo	List of diagnostic information <i>dataToReturn</i> (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>dataToReturn</i> request parameter. This list is empty if diagnostics information was not requested in the request header.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the requested <i>NodeTypeDescription</i> .

### 5.9.3.3 Service results

If the *Query* is invalid or cannot be processed, then *QueryDataSets* are not returned and only a *Service result* and optional *DiagnosticInfo* is returned. Table 43 defines the *Service results* specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 43 – QueryFirst Service Result Codes**

Symbolic Id	Description
Bad_InvalidPath	If the <i>relativePath</i> string is invalid, this result code is returned.
Bad_InvalidFilter	If the filter specification is invalid, this result code is returned.
Bad_QueryError	If the query cannot be processed for any reason, this result code is returned.
Bad_ClientTimeOut	If the query cannot be processed in the time specified by the client or imposed by the server
Good_ResultsMayBeIncomplete	The server should have followed a reference to a node in a remote server but did not. The result set may be incomplete.

## 5.9.4 QueryNext

### 5.9.4.1 Descriptions

This Service is used to request the next set of *QueryFirst* or *QueryNext* response information that is too large to be sent in a single response. “Too large” in this context means that the *Server* is not able to return a larger response or that the number of *QueryDataSets* to return exceeds the maximum number of *QueryDataSets* to return that was specified by the *Client* in the original request. The *QueryNext* must be submitted on the same session that was used to submit the *QueryFirst* or *QueryNext* that is being continued.

### 5.9.4.2 Parameters

Table 44 defines the parameters for the Service.

**Table 44 – QueryNext Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
releaseContinuationPoint	Boolean	A Boolean parameter with the following values : TRUE      passed <i>continuationPoint</i> will be reset to free resources for the continuation point in the <i>Server</i> . FALSE     passed <i>continuationPoint</i> will be used to get the next set of browse information. A <i>Client</i> must always use the continuation point returned by a <i>QueryFirst</i> or <i>QueryNext</i> response to free the resources for the continuation point in the <i>Server</i> . If the <i>Client</i> does not want to get the next set of <i>Query</i> information, <i>QueryNext</i> must be called with this parameter set to TRUE. If the parameter is set to TRUE all array parameters in the response must contain empty arrays.
continuationPoint	ByteString	Server defined opaque value that represents the continuation point. The value of the continuation point was returned to the <i>Client</i> in a previous <i>QueryFirst</i> or <i>QueryNext</i> response. This value is used to identify the previously processed <i>QueryFirst</i> or <i>QueryNext</i> request that is being continued, and the point in the result set from which the browse response is to continue.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
queryDataSets []	QueryDataSet	The array of <i>QueryDataSets</i> . The <i>QueryDataSet</i> type is defined in Clause 7.15.
revisedContinuationPoint	ByteString	Server-defined opaque value that represents the continuation point. It is used only if the information to be returned is too large to be contained in a single response. When not used or when <i>releaseContinuationPoint</i> is set, the value of this parameter is null.

### 5.9.4.3 Service results

Table 45 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 45 – QueryNext Service Result Codes**

Symbolic Id	Description
Bad_ContinuationPointInvalid	See Table 156 for the description of this result code.

## 5.9.5 Example for Queries and ContentFilters

### 5.9.5.1 Overview

As discussed in Clause 7.2 and Clause 8.4, a *ContentFilter* structure defines a collection of elements that make up a filtering criteria. Each element in the collection consists of an operator and an array of operands to be used by the operator.

These query examples illustrate *ContentFilters*. The following of operands are used (for a definition of these operand see Clause 8.4):

- Attribute: Used to refer to a *Node* or an *Attribute* of a *Node*.
- Property: Used to refer to the *Value Attribute* of a *Property* associated with a *Node*.
- Literal: Used to hold a constant.

These examples illustrate the use of a subset of allowed operators (for definition of the Operators see Clause 7.2):

- Equals (=): Evaluates to TRUE if two operands are equal.
- And: Evaluates to TRUE if both of two content filter sub-trees evaluate to TRUE.
- Or: Evaluates to TRUE if either of two content filter sub-trees evaluate to TRUE.
- RelatedTo: Used to specify the *References* to follow (hop), between two *Nodes*, evaluates to TRUE if the hop exist as specified. The first two operands specify the source and destination node types respectively, the third operand the type of relationship, the last operand specifies the number of hops the relationship should be followed before the destination node type is encountered.

### 5.9.5.2 Used type model

The following examples use the type model described below:

New Reference types:

- “HasChild” derived from *HierarchicalReference*.
- “HasAnimal” derived from *HierarchicalReference*.
- “HasPet” derived from *HasAnimal*.
- “HasFarmAnimal” derived from *HasAnimal*.
- “HasSchedule” derived from *HierarchicalReference*.

*PersonType* derived from *BaseObjectType* adds

- HasProperty "LastName"
- HasProperty "FirstName"
- HasProperty "StreetAddress""
- HasProperty "City"
- HasProperty "ZipCode"
- May have HasChild reference to a node of type *PersonType*
- May have HasAnimal reference to a node of type *AnimalType* (or a sub type of this *Reference* type)

*AnimalType* derived from *BaseObjectType* adds

- May have HasSchedule reference to a node of type *FeedingScheduleType*
- HasProperty "Name"

*DogType* derived from *AnimalType* adds

- HasProperty "NickName"
- HasProperty "DogBreed"
- HasProperty "License"

*CatType* derived from *AnimalType* adds

- HasProperty "NickName"
- HasProperty "CatBreed"

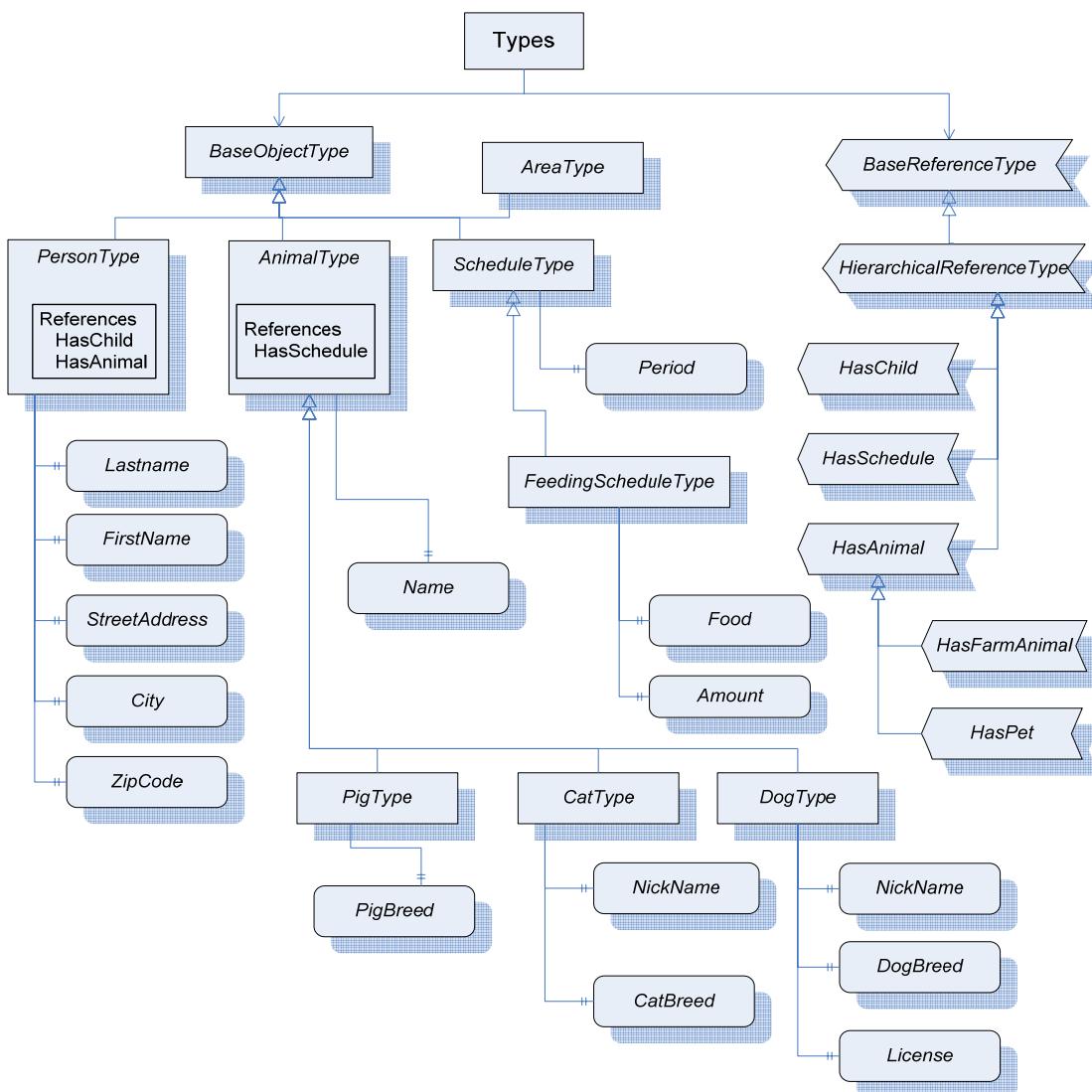
PigType derived from AnimalType adds  
HasProperty "PigBreed"

ScheduleType derived from BaseObjectType adds  
HasProperty "Period"

FeedingScheduleType derived from ScheduleType adds  
HasProperty "Food"  
HasProperty "Amount"

AreaType derived from *BaseObjectType* is just a simple *Folder* and contains no *Properties*.

This example type system is shown in Figure 14. In this Figure, the standard notation is used for all References to *Object* types, *Properties* and sub-types. Additionally supported *References* are contained in an inner box. The actual references only exist in the instances thus no connections to other *Objects* are shown in the Figure and they may be sub-types of the listed *Reference*.



**Figure 14 – Example Type Nodes**

A corresponding example set of instances is shown in Figure 15. These instances include a type *Reference* for *Objects*. Properties also have type *References*, but the *References* are omitted for simplicity. The name of the *Object* is provided in the box and a numeric instance *NodeId* in brackets. Standard *Reference* types use the standard notation, custom *Reference* types are listed with a named *Reference*. For *Properties*, the *BrowseName*, *NodeId*, and *Value* are shown. The *Nodes* that are included in a *View* (*View1*) are enclosed in the colored box. Two *Area* nodes are included for grouping of the existing person nodes.

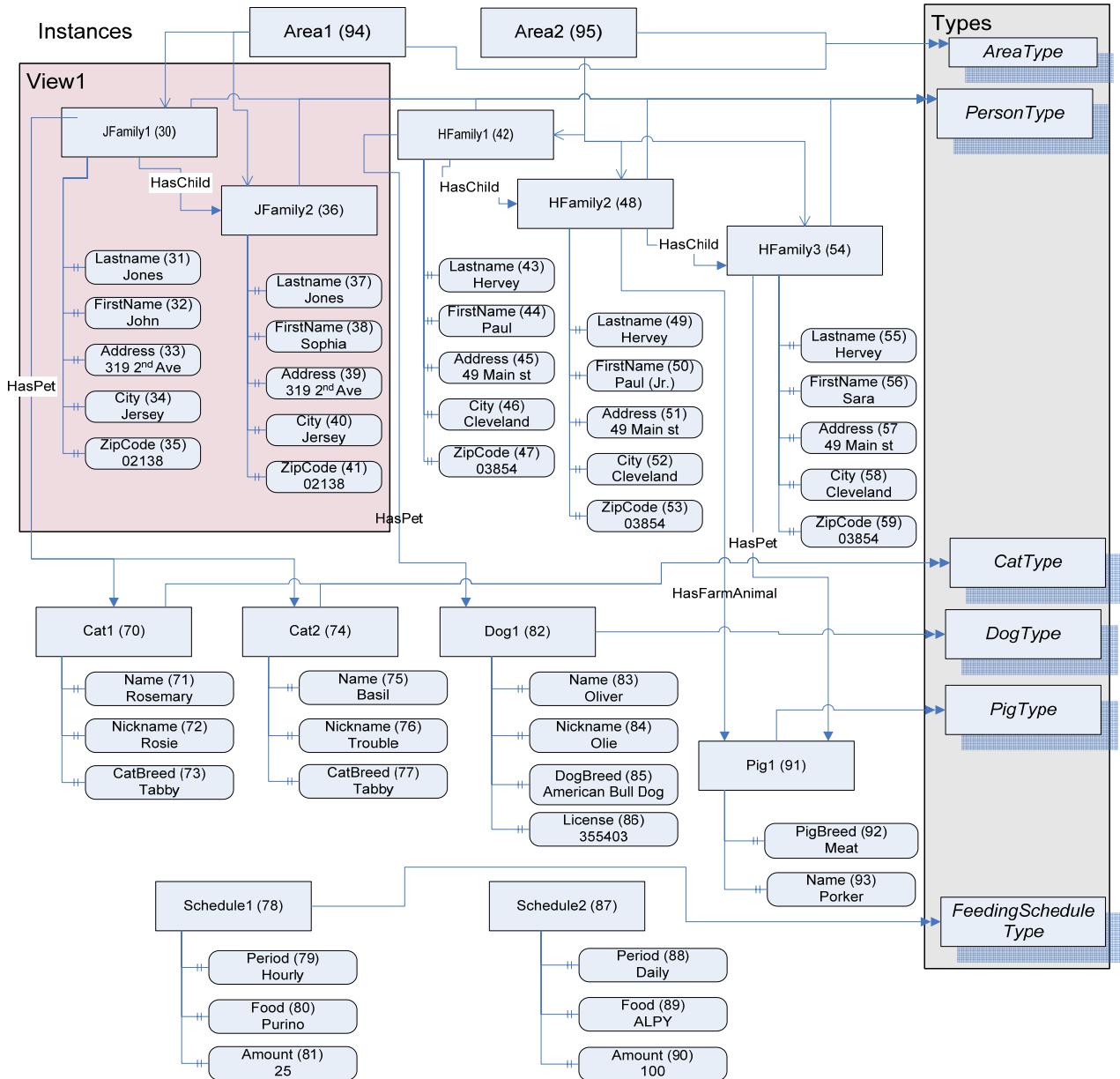


Figure 15 - Example Instance Nodes

### 5.9.5.3 Example Notes

For all of the examples in Clause 5.9.5, the type definition *Node* is listed in its symbolic form, in the actual call it would be the *NodeID* assigned to the *Node*. The *AttributetID* is also the symbolic name of the *Attribute*, in the actual call they would be translated to the *IntegerID* of the *Attribute*. Also in all of the examples the *BrowseName* is included in the result table for clarity, normally this would not be returned.

### 5.9.5.4 Example 1

This example requests a simple layered filter, a person has a pet and the pet has a schedule.

**Example 1: Get PersonType.lastName, AnimalType.name, ScheduleType.period where the Person Has a Pet and that Pet Has a Schedule.**

The *NodeTypeDescription* parameters used in the example are described in Table 46.

**Table 46 – Example 1 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".LastName"	value	N/A
		"<HasPet>AnimalType.name"	value	N/A
		"<HasPet>AnimalType<HasSchedule> Schedule.period"	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 16.

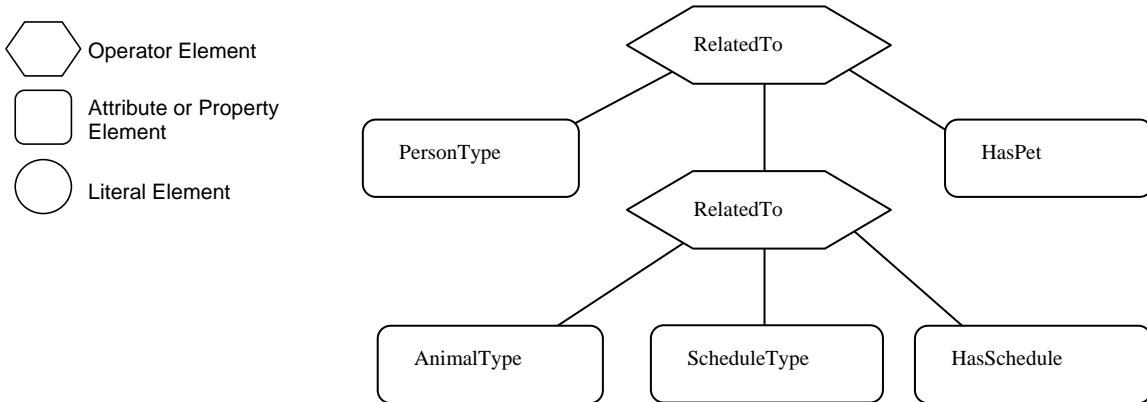
**Figure 16 - Example 1 Filter**

Table 47 describes the *ContentFilter* elements, operators and operands used in the example.

**Table 47 – Example 1 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	ElementOperand = 2	AttributeOperand = Nodeld: HasPet, Attributeld: Nodeld	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = Nodeld: AnimalType, Attributeld: Nodeld	AttributeOperand = Nodeld: ScheduleType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasSchedule, Attributeld: Nodeld	LiteralOperand= '1'

Table 48 describes the *QueryDataSet* that results from this query if it were executed against the instances described in Figure 15.

**Table 48 – Example 1 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
30 (JFamily1)	PersonType	".lastName"	Jones
		"<HasPet>AnimalType.name"	Rosemary
			Basil
		"<HasPet>AnimalType<HasSchedule> Schedule.period"	Hourly
42(HFamily1)	PersonType	".lastName"	Hervey
		"<HasPet>AnimalType..name"	Olive
		"<HasPet>AnimalType<HasSchedule> Schedule.period"	Daily

The Value column is returned as an array for each *Node* description, where the order of the items in the array would correspond to the order of the items that were requested for the given Node Type. In Addition if a single *Attribute* has multiple values then it would be returned as an array within the larger array, for example in this table Rosemary and Basil would be returned in a array the .<hasPet>.AnimalType.name item. They are show as separate rows for ease of viewing.

[Note: that the relative path column and browse name (in parentheses in the *NodeID* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinition NodeID* would be an integer not the symbolic name that is included in the table].

### 5.9.5.5 Example 2

The second example illustrates receiving a list of disjoint *Nodes* and also illustrates that an array of results can be received.

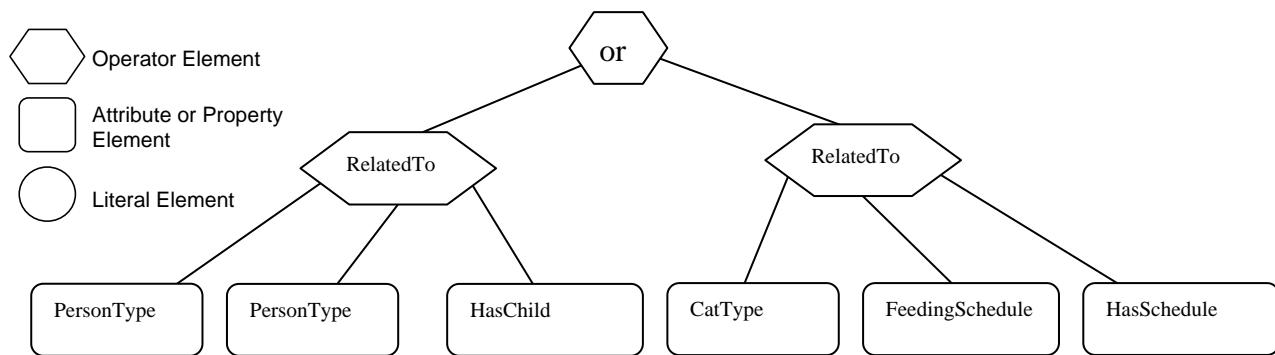
**Example 2: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule).**

The *NodeTypeDescription* parameters used in the example are described in Table 49.

**Table 49 – Example 2 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".LastName"	value	N/A
AnimalType	TRUE	".name"	value	N/A

The corresponding ContentFilter is illustrated in Figure 17.



**Figure 17 – Example 2 Filter Logic Tree**

Table 50 describes the elements, operators and operands used in the example. It is worth noting that a Cattype is a subtype of Animaltype.

**Table 50 – Example 2 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = Nodeld: CatType, Attributeld: Nodeld	AttributeOperand = Nodeld: FeedingScheduleType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasSchedule, Attributeld: Nodeld	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 51.

**Table 51 – Example 2 QueryDataSets**

NodeId	TypeDefinition NodeId	RelativePath	Value
30 (Jfamily1)	PersonType	.LastName	Jones
42 (HFamily1)	PersonType	.LastName	Hervey
48 (HFamily2)	PersonType	.LastName	Hervey
70 (Cat1)	CatType	.name	Rosemary
74 (Cat2)	CatType	.name	Basil

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table].

#### 5.9.5.6 Example 3

The third example provides a more complex *Query* in which the results are filtered on multiple criteria.

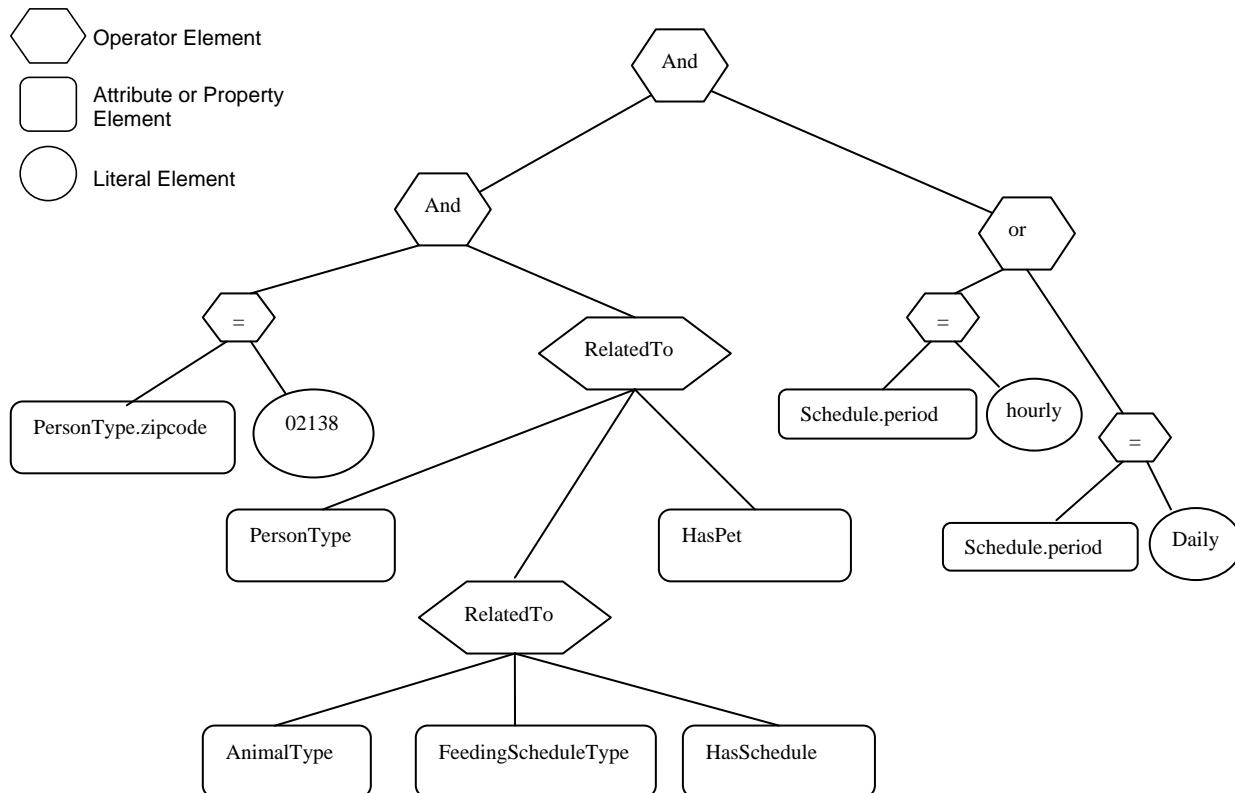
**Example 3: Get PersonType.lastName, AnimalType.name, ScheduleType.period where a person has a pet and the animal has a feeding schedule and the person has a zipcode = '02138' and the schedule.period is daily or hourly.**

Table 52 describes the NodeTypeDescription parameters used in the example.

**Table 52 – Example 3 - NodeTypeDescriptions**

Type Definition Node	Include Subtypes	RelativePath	Attribute Id	Index Range
PersonType	FALSE	"PersonType.lastName"	Value	N/A
		"PersonType<HasPet>AnimalType.name"	Value	N/A
		"PersonType<HasPet>AnimalType<HasSchedule> FeedingSchedule.period"	Value	N/A

The corresponding *ContentFilter* is illustrated in Figure 18.



**Figure 18 – Example 3 Filter Logic Tree**

Table 53 describes the elements, operators and operands used in the example.

**Table 53 – Example 3 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	And	Element Operand= 1	ElementOperand = 2		
1	And	ElementOperand = 3	ElementOperand = 5		
2	Or	ElementOperand = 6	ElementOperand = 7		
3	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	ElementOperand = 4	AttributeOperand = Nodeld: HasPet, Attributeld: Nodeld	LiteralOperand = '1'
4	RelatedTo	AttributeOperand = Node: AnimalType, Attributeld: Nodeld	AttributeOperand = Nodeld: FeedingScheduleType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasSchedule, Attributeld: Nodeld	LiteralOperand = '1'
5	Equals	PropertyOperand = Nodeld: PersonType Property: zipcode	LiteralOperand = '02138'		
6	Equals	PropertyOperand = Nodeld: ScheduleType Property: Period	LiteralOperand = 'Daily'		
7	Equals	PropertyOperand = Nodeld: ScheduleType Property: Period	LiteralOperand = 'Hourly'		

The results from this query would contain the *QueryDataSets* shown in Table 54.

**Table 54 – Example 3 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
30 (JFamily1)	PersonType	".lastName"	Jones
		"<hasPet>PersonType.name"	Rosemary
			Basil
		"<hasPet>AnimalType<hasSchedule>FeedingSchedule.period"	Hourly
			Hourly

[Note: that the relative path column and browse name (in parentheses in the *Nodeld* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinitionNodeld* would be an integer not the symbolic name that is included in the table].

#### 5.9.5.7 Example 4

The fourth example provides an illustration of the Hop parameter that is part of the RelatedTo Operator.

**Example 4: Get PersonType.lastName where a person has a child who has a child who has a pet.**

Table 55 describes the *NodeTypeDescription* parameters used in the example.

**Table 55 – Example 4 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".lastName"	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 19.

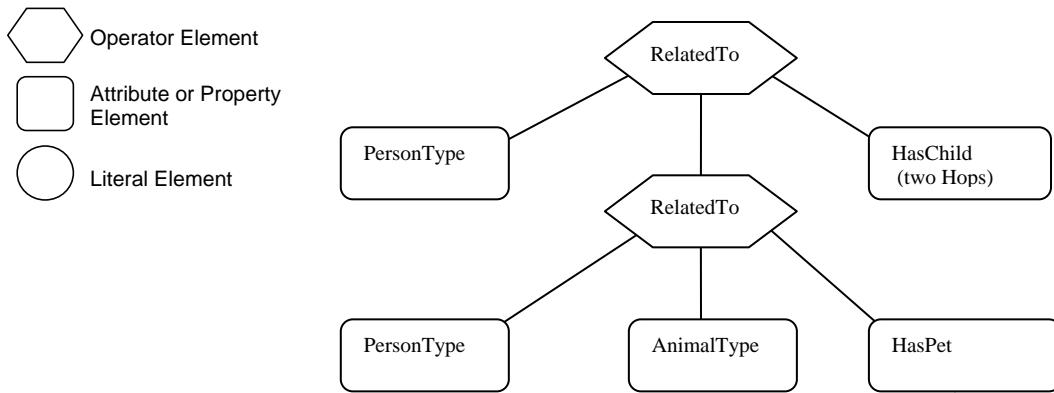
**Figure 19 – Example 4 Filter Logic Tree**

Table 56 describes the elements, operators and operands used in the example.

**Table 56 – Example 4 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	Element Operand = 1	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = '2'
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: AnimalType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasPet, Attributeld: Nodeld	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 57. It is worth noting that the pig “Pig1” is referenced as a pet by Sara, but is referenced as a farmanimal by Sara’s parent Paul.

**Table 57 – Example 4 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
42 (HFamily1)	PersonType	“.lastName”	Hervey

[Note: that the relative path column and browse name (in parentheses in the *Nodeld* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinitionNodeld* would be an integer not the symbolic name that is included in the table].

### 5.9.5.8 Example 5

The fifth example provides an illustration of the use of alias.

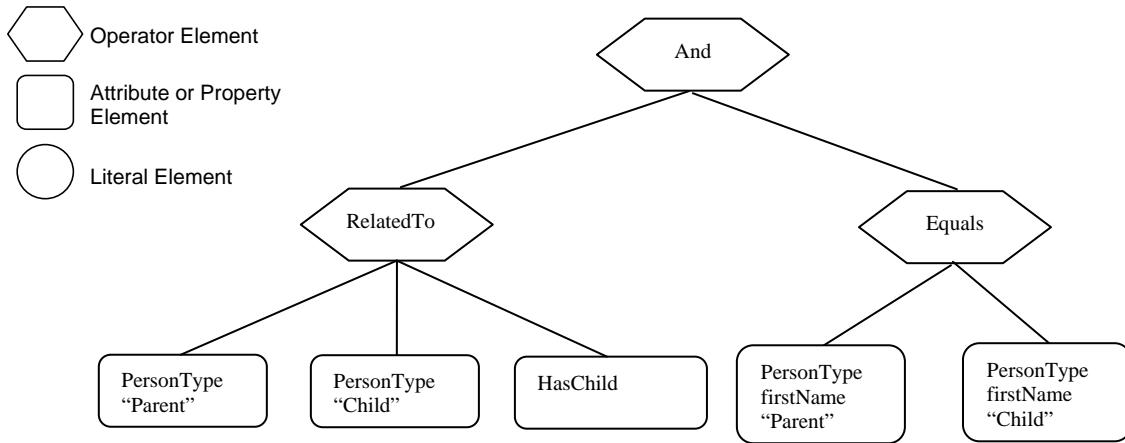
#### Example 5: Get the last names of children that have the same first name as a parent of theirs

Table 58 describes the *NodeTypeDescription* parameters used in the example.

**Table 58 – Example 5 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	“<HasChild>PersonType.lastName”	Value	N/A

The corresponding *ContentFilter* is illustrated in Figure 20.

**Figure 20 – Example 5 Filter Logic Tree**

In this example, one *Reference* to *PersonType* is aliased to “Parent” and another *Reference* to *PersonType* is aliased to “Child”. The value of *Parent.firstName* and *Child.firstName* are then compared. Table 59 describes the elements, operators and operands used in the example.

**Table 59 – Example 5 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	And	ElementOperand = 1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld, Alias: “Parent”	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld, Alias: “Child”	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = “1”
2	Equals	PropertyOperand = Nodeld: PersonType, Property: FirstName, Alias: “Parent”	PropertyOperand = Nodeld: PersonType, Property: FirstName, Alias: “Child”		

The results from this query would contain the *QueryDataSets* shown in Table 60.

**Table 60 – Example 5 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
42 (HFamily1)	PersonType	“<HasChild>PersonType.lastName”	Hervey

### 5.9.5.9 Example 6

The sixth example provides an illustration a different type of request, one in which the *Client* is interested in displaying part of the address space of the server. This request includes listing a *Reference* as something that is to be returned.

**Example 6: Get PersonType.Nodeld, AnimalType.Nodeld, PersonType.HasChild Reference, PersonType.HasAnimal Reference where a person has a child who has a Animal.**

Table 61 describes the *NodeTypeDescription* parameters used in the example.

**Table 61 – Example 6 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".Nodeld"	value	N/A
		<HasChild>PersonType<HasAnimal>AnimalType.Nodeld	value	N/A
		<HasChild>	value	N/A
		<HasChild>PersonType<HasAnimal>	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 21.

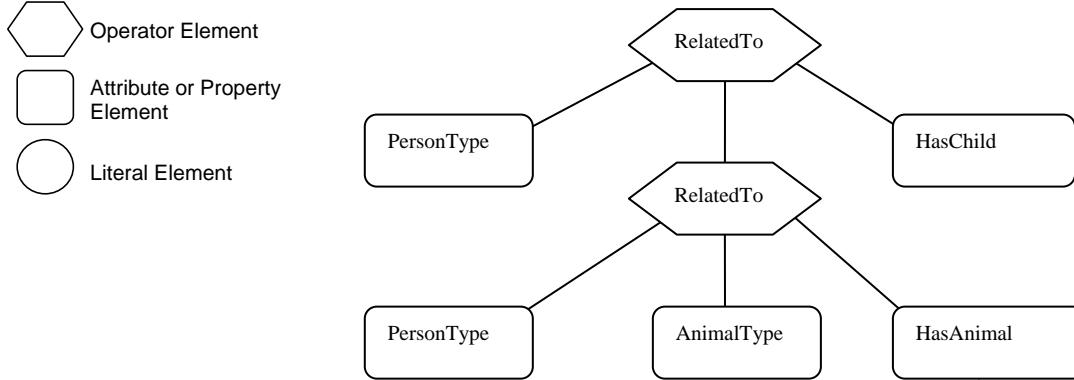
**Figure 21 – Example 6 Filter Logic Tree**

Table 62 describes the elements, operators and operands used in the example.

**Table 62 – Example 6 ContentFilter**

Element[0]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	ElementOperand = 1	AttributeOperand = Node:HasChild, Attr:Nodeld	LiteralOperand = '1'
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: AnimalType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasAnimal, Attributeld: Nodeld	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 63.

**Table 63 – Example 6 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
42 (HFamily1)	PersonType	".Nodeld"	42 (HFamily1)
		<HasChild>PersonType<HasAnimal>AnimalType.Nodeld	91 (Pig1)
		<HasChild>	HasChild ReferenceDescription
		<HasChild>PersonType<HasAnimal>	HasFarmAnimal ReferenceDescription
48 (HFamily2)	PersonType	".Nodeld"	48 (HFamily2)
		<HasChild>PersonType<HasAnimal>AnimalType.Nodeld	91 (Pig1)
		<HasChild>	HasChild ReferenceDescription
		<HasChild>PersonType<HasAnimal>	HasPet ReferenceDescription

[Note: that the relative path and browse name (in parentheses) is not in the *QueryDataSet* and is only shown here for clarity and the *TypeDefinitionNodId* would be an integer not the symbolic name that is included in the table. The value field would in this case be the *NodId* where it was requested, but for the example the browse name is provided in parentheses and in the case of *Reference* types on the browse name is provided. For the *References* listed in Table 63, the value would be a *ReferenceDescription* which are described in Clause 7.17].

Table 64 provides an example of the same *QueryDataSet* as shown in Table 63 without any additional fields and minimal symbolic Ids. There is an entry for each requested *Attribute*, in the cases where an *Attribute* would return multiple entries the entries are separated by commas. If a structure is being returned then the structure is enclosed in square brackets. In the case of a *ReferenceDescription* the structure contains a structure and *DisplayName* and *BrowseName* are assumed to be the same and defined in Figure 15.

**Table 64 – Example 6 QueryDataSets without Additional Information**

NodId	TypeDefinition NodId	Value
42	PersonType	42
		91
		[HasChild,TRUE,[48,HFamily2,HFamily2,PersonType]],
		[HasFarmAnimal,TRUE[91,Pig1,Pig1,PigType]]
48	PersonType	54
		91
		[HasChild,TRUE,[ 54,HFamily3,HFamily3,PersonType]]
		[HasPet, TRUE,[ 91,Pig1,Pig1,PigType]]

The PersonType, HasChild, PigType, HasPet, HasFarmAnimal identifiers used in the above table would be translated to actual *ExtendedNodIds*.

#### 5.9.5.10 Example 7

The seventh example provides an illustration a request in which a *Client* wants to display part of the address space based on a starting point that was obtained via standard browsing. This request includes listing *References* as something that is to be returned. In this case the Person Browsed to Area2 and wanted to *Query* for information below this starting point.

**Example 7: Get PersonType.NodId, AnimalType.NodId, PersonType.HasChild Reference, PersonType.HasAnimal Reference where the person is in Area2 (Cleveland nodes) and the person has a child.**

Table 65 describes the *NodeTypeDescription* parameters used in the example.

**Table 65 – Example 7 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".NodId"	value	N/A
		<HasChild>	value	N/A
		<HasAnimal>NodId	value	N/A
		<HasAnimal>	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 22. Note the *Browse* call would typically return a *NodId*, thus the first filter is for the *BaseObjectType* with a *NodId* of 95 where 95 is the *NodId* associated with the Area2 node, all Nodes descend from *BaseObjectType*, and *NodId* is a base *Property* so this filter will work for all *Queries* of this nature.

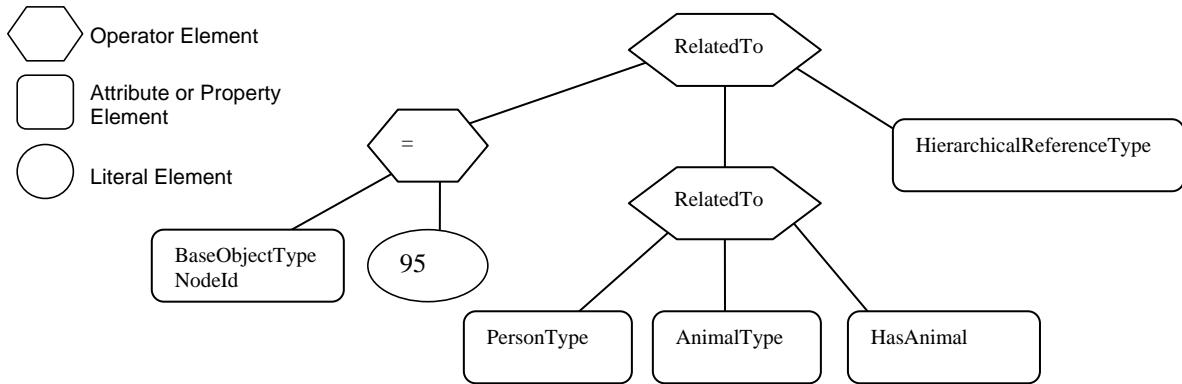
**Figure 22 – Example 7 Filter Logic Tree**

Table 66 describes the elements, operators and operands used in the example.

**Table 66 – Example 7 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = Nodeld: BaseObjectType, Attributeld: Nodeld	ElementOperand = 1	AttributeOperand = Node:HierachicalReference, Attr:Nodeld	LiteralOperand = '1'
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = '1'
2	Equals	PropertyOperand = Nodeld: BaseObjectType, Property: Nodeld,	LiteralOperand = '95'		

The results from this *Query* would contain the *QueryDataSets* shown in Table 67.

**Table 67 – Example 7 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
42 (HFamily1)	PersonType	".Nodeld"	42 (HFamily1)
		<HasChild>	HasChild <i>ReferenceDescription</i>
		<HasAnimal>AnimalType.Nodeld	NULL
		<HasAnimal>	HasFarmAnimal <i>ReferenceDescription</i>
48 (HFamily2)	PersonType	".Nodeld"	48 (HFamily2)
		<HasChild>	HasChild <i>ReferenceDescription</i>
		<HasAnimal>AnimalType.Nodeld	91 (Pig1)
		<HasAnimal>	HasFarmAnimal <i>ReferenceDescription</i>

[Note: that the relative path and browse name (in parentheses) is not in the *QueryDataSet* and is only shown here for clarity and the *TypeDefinitionNodeld* would be an integer not the symbolic name that is included in the table. The value field would in this case be the *Nodeld* where it was requested, but for the example the browse name is provided in parentheses and in the case of *Reference* types on the browse name is provided. For the *References* listed in Table 67, the value would be a *ReferenceDescription* which are described in Clause 7.17].

### 5.9.5.11 Example 8

The eighth example provides an illustration of a request in which the address space is restricted by a *Server defined View*. This request is the same as in the second example which illustrates receiving a list of disjoint *Nodes* and also illustrates that an array of results can be received. It is **important** to note that all of the parameters and the *contentFilter* are the same, only the View description would be specified as "View1"

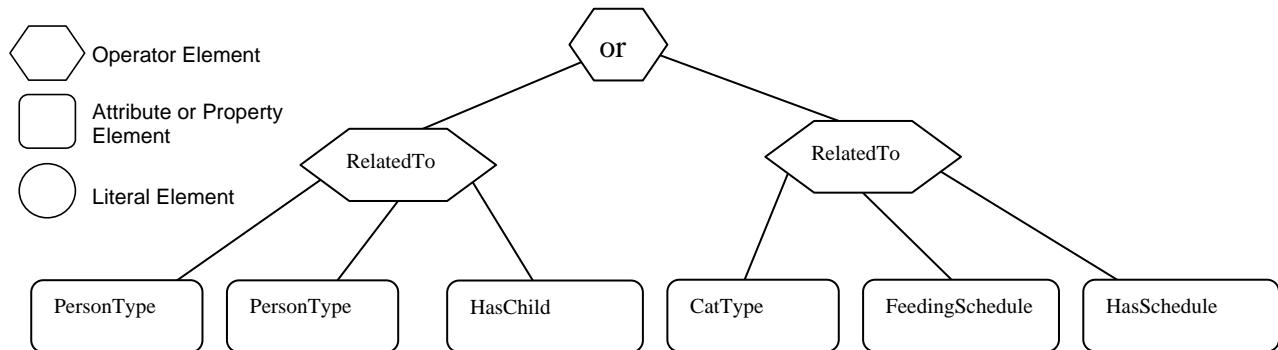
**Example 8: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule) limited by the address space in View1.**

The NodeTypeDescription parameters used in the example are described in Table 68

**Table 68 – Example 8 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".LastName"	value	N/A
AnimalType	TRUE	".name"	value	N/A

The corresponding ContentFilter is illustrated in Figure 23.



**Figure 23 – Example 8 Filter Logic Tree**

Table 69 describes the elements, operators and operands used in the example. It is worth noting that a CatType is a subtype of AnimalType.

**Table 69 – Example 8 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = Nodeld: CatType, Attributeld: Nodeld	AttributeOperand = Nodeld: FeedingScheduleType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasSchedule, Attributeld: Nodeld	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 70. If this is compared to the result set from example 2, the only difference is the omission of the Cat Nodes. These Nodes are not in the View and thus are not included in the result set

**Table 70 – Example 8 QueryDataSets**

Nodeld	TypeDefinition Nodeld	RelativePath	Value
30 (Jfamily1)	PersonType	.LastName	Jones

[Note: that the relative path column and browse name (in parentheses in the *Nodeld* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinitionNodeld* would be an integer not the symbolic name that is included in the table].

#### 5.9.5.12 Example 9

The ninth example provides a further illustration for a request in which the address space is restricted by a Server defined View. This request is similar to the second example except that some

of the requested nodes are expressed in terms of a relative path. It is important to note that the *contentFilter* is the same, only the View description would be specified as "View1".

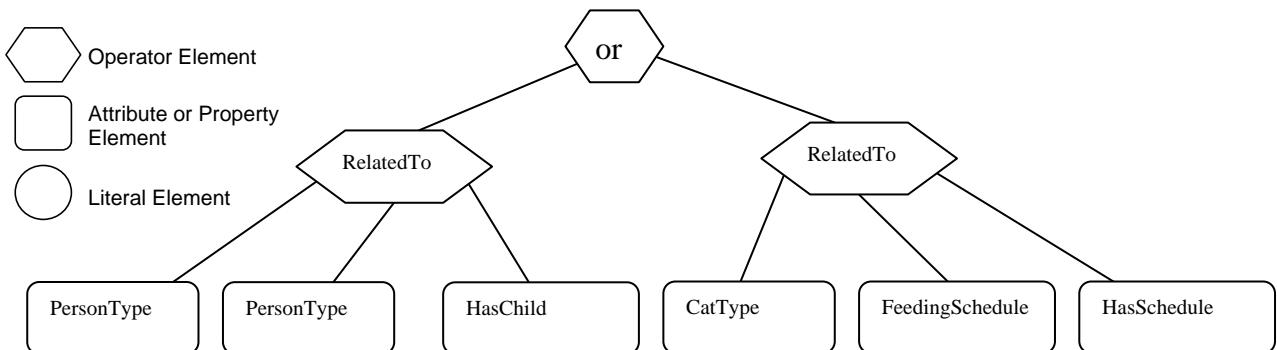
**Example 9: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule) limited by the address space in View1.**

Table 71 describes the NodeTypeDescription parameters used in the example.

**Table 71 – Example 9 NodeTypeDescription**

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".Nodeld"	value	N/A
		<HasChild>PersonType<HasAnimal>AnimalType.Nodeld	value	N/A
		<HasChild>	value	N/A
		<HasChild>PersonType<HasAnimal>	value	N/A
PersonType	FALSE	".LastName"	value	N/A
		<HasAnimal>AnimalType.Name	value	N/A
AnimalType	TRUE	"name"	value	N/A

The corresponding ContentFilter is illustrated in Figure 24.



**Figure 24 – Example 9 Filter Logic Tree**

Table 72 describes the elements, operators and operands used in the example.

**Table 72 – Example 9 ContentFilter**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: PersonType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasChild, Attributeld: Nodeld	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = Nodeld: CatType, Attributeld: Nodeld	AttributeOperand = Nodeld: FeedingScheduleType, Attributeld: Nodeld	AttributeOperand = Nodeld: HasSchedule, Attributeld: Nodeld	LiteralOperand = '1'

The results from this Query would contain the *QueryDataSets* shown in Table 73. If this is compared to the result set from example 2, the Pet Nodes are included in the list, even though they are outside of the View. This is possible since the name referenced via the relative path and the root Node is in the View.

**Table 73 – Example 9 QueryDataSets**

<b>NodeId</b>	<b>TypeDefinition NodeId</b>	<b>RelativePath</b>	<b>Value</b>
30 (Jfamily1)	PersonType	.LastName	Jones
		<HasAnimal>AnimalType.Name	Rosemary
		<HasAnimal>AnimalType.Name	Basil

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table].

## 5.10 Attribute Service Set

### 5.10.1 Overview

This Service Set provides Services to access *Attributes* that are part of *Nodes*.

### 5.10.2 Read

#### 5.10.2.1 Description

This Service is used to read one or more *Attributes* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this Service allows *Clients* to read the entire set of indexed values as a composite, to read individual elements or to read ranges of elements of the composite.

The maxAge parameter is used to direct the Server to access the value from the underlying data source, such as a device, if its copy of the data is older than the maxAge specifies. If the Server cannot meet the requested max age, it returns its “best effort” value rather than rejecting the request.

### 5.10.2.2 Parameters

Table 74 defines the parameters for the Service.

**Table 74 – Read Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
maxAge	Duration	<p>Maximum age of the value to be read in milliseconds (see Clause 7.6 for <i>Duration</i> definition). The age of the value is based on the <i>Server</i> timestamp when the <i>Server</i> starts processing the request. For example if the <i>Client</i> specifies a <i>maxAge</i> of 500 milliseconds and it takes 100 milliseconds until the <i>Server</i> starts processing the request, the age of the returned value could be 600 milliseconds prior to the time it was requested.</p> <p>If the <i>Server</i> has one or more values of an <i>Attribute</i> that are within the maximum age, it can return any one of the values or it can read a new value from the data source. The number of values of an <i>Attribute</i> that a <i>Server</i> has depends on the number of <i>MonitoredItems</i> that are defined for the <i>Attribute</i>. In any case, the <i>Client</i> can make no assumption about which copy of the data will be returned.</p> <p>If the <i>Server</i> does not have a value that is within the maximum age, it must attempt to read a new value from the data source.</p> <p>If the <i>Server</i> cannot meet the requested <i>maxAge</i>, it returns its “best effort” value rather than rejecting the request. This may occur when the time it takes the <i>Server</i> to process and return the new data value after it has been accessed is greater than the specified maximum age.</p> <p>If <i>maxAge</i> is set to 0, the <i>Server</i> must attempt to read a new value from the data source.</p> <p>If <i>maxAge</i> is set to the max Int32 value, the <i>Server</i> must attempt to get a cached value if a <i>MonitoredItem</i> is defined for the <i>Attribute</i>.</p> <p>Negative values are invalid for <i>maxAge</i>.</p>
timestampsToReturn	enum TimestampsToReturn	An enumeration that specifies the <i>Timestamp Attributes</i> to be returned for each requested <i>Variable Value Attribute</i> . The <i>TimestampsToReturn</i> enumeration is defined in Clause 7.29.
nodesToRead []	ReadValueId	List of <i>Nodes</i> and their <i>Attributes</i> to read. For each entry in this list, a <i>StatusCodes</i> is returned, and if it indicates success, the <i>Attribute Value</i> is also returned. The <i>ReadValueId</i> parameter type is defined in Clause 7.16.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	DataValue	List of <i>Attribute</i> values (see Clause 7.4 for <i>DataValue</i> definition). The size and order of this list matches the size and order of the <i>nodesToRead</i> request parameter. There is one entry in this list for each <i>Node</i> contained in the <i>nodesToRead</i> parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of this list matches the size and order of the <i>nodesToRead</i> request parameter. There is one entry in this list for each <i>Node</i> contained in the <i>nodesToRead</i> parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.10.2.3 Service results

Table 75 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 75 – Read Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_MaxAgeInvalid	The max age parameter is invalid.
Bad_TimestampsToReturnInvalid	See Table 156 for the description of this result code.

### 5.10.2.4 StatusCodes

Table 76 defines values for the operation level *statusCode* contained in the *DataValue* structure of each *values* element. Common *StatusCodes* are defined in Table 157.

**Table 76 – Read Operation Level Result Codes**

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_AttributeldInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeNoData	See Table 157 for the description of this result code.
Bad_DataEncodingInvalid	See Table 157 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 157 for the description of this result code.
Bad_NoReadRights	See Table 157 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code.

### 5.10.3 HistoryRead

#### 5.10.3.1 Description

This *Service* is used to read historical values or *Events* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this *Service* allows *Clients* to read the entire set of indexed values as a composite, to read individual elements or to read ranges of elements of the composite. *Servers* may make historical values available to *Clients* using this *Service*, although the historical values themselves are not visible in the *AddressSpace*.

The *AccessLevel Attribute* defined in [UA Part 3] indicates a *Node*'s support for historical values. Several request parameters indicate how the *Server* is to access values from the underlying history data source. The *EventNotifier Attribute* defined in [UA Part 3] indicates a *Node*'s support for historical *Events*.

The *continuationPoint* parameter in the *HistoryRead* is used to mark a point from which to continue the read if not all values could be returned in one response. The value is opaque for the *Client* and is only used to maintain the state information for the *Server* to continue from. A *Server* may use the timestamp of the last returned data item if the timestamp is unique. This can reduce the need in the *Server* to store state information for the continuation point.

For additional details on reading historical data and historical *Events* see [UA Part 11].

#### 5.10.3.2 Parameters

Table 77 defines the parameters for the *Service*.

**Table 77 – HistoryRead ServiceParameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
historyReadDetails	Extensible Parameter HistoryReadDetails	The details define the types of history reads that can be performed. The <i>HistoryReadDetails</i> parameter type is an extensible parameter type formally defined in [UA Part 11]. The <i>ExtensibleParameter</i> type is defined in Clause 8.2.
timestampsToReturn	enum TimestampsToReturn	An enumeration that specifies the timestamp <i>Attributes</i> to be returned for each requested <i>Variable Value Attribute</i> . The <i>TimestampsToReturn</i> enumeration is defined in Clause 7.29. Specifying a <i>TimestampsToReturn</i> of NEITHER is not valid. A <i>Server</i> must return a <i>Bad_InvalidTimestampArgument StatusCode</i> in this case. If the requested timestamp is not stored for a <i>Node</i> , the operation for the <i>Node</i> must return the <i>StatusCode Bad_NoTimestamp</i> . When reading <i>Events</i> this only applies to <i>Event</i> fields that are of type <i>DataValue</i> .
releaseContinuation Points	Boolean	A Boolean parameter with the following values : TRUE      passed <i>continuationPoints</i> will be reset to free resources for the continuation point in the <i>Server</i> . FALSE     passed <i>continuationPoints</i> will be used to get the next set of history information. A <i>Client</i> must always use the continuation point returned by a <i>HistoryRead</i> response to free the resources for the continuation point in the <i>Server</i> . If the <i>Client</i> does not want to get the next set of history information, <i>HistoryRead</i> must be called with this parameter set to TRUE.
nodesToRead []	HistoryReadValueId	This parameter contains the list of items upon which the historical retrieval is to be performed.
nodeId	NodeId	If the <i>parameterTypeld</i> of <i>HistoryReadDetails</i> has the value RAW, PROCESSED, MODIFIED or ATTIME: The <i>nodeId</i> of the <i>Nodes</i> whose historical values are to be read. The value returned must always include a timestamp. If the <i>parameterTypeld</i> of <i>HistoryReadDetails</i> has the value EVENTS: The <i>nodeId</i> of the <i>Node</i> whose <i>Event</i> history is to be read. If the <i>Node</i> does not support the requested access for historical values or historical <i>Events</i> the appropriate error response for the given <i>Node</i> will be generated.
dataEncoding	QualifiedName	A <i>QualifiedName</i> that specifies the data encoding to be returned for the <i>Node</i> to be read (see Clause 7.16 for definition how to specify the data encoding).
continuationPoint	ByteString	For each <i>NodeToRead</i> this parameter specifies a continuation point returned from a previous <i>HistoryRead</i> call, allowing the <i>Client</i> to continue that read from the last value received. The <i>HistoryRead</i> is used to select an ordered sequence of historical values. A continuation point marks a point in that ordered sequence, such that the <i>Server</i> returns the subset of the sequence that follows that point. A null value indicates that this parameter is not used. This continuation point is described in more detail in [UA Part 11].
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> type).
results []	HistoryReadResult	List of read results. The size and order of the list matches the size and order of the <i>nodesToRead</i> request parameter.
statusCode	StatusCodes	<i>StatusCodes</i> for the <i>NodeToRead</i> (see Clause 7.28 for <i>StatusCodes</i> definition).
continuationPoint	ByteString	This parameter is used only if the number of values to be returned is too large to be returned in a single response. In this case the <i>StatusCodes</i> of the read result is set to <i>Good_MoreData</i> . When this parameter is not used, its value is null. <i>Servers</i> must support at least one continuation point per <i>Session</i> . <i>Servers</i> specify a max continuation points per <i>Session</i> in the <i>Server capabilities Object</i> defined in [UA Part 5]. A continuation point will remain active until the <i>Client</i> passes the continuation point to <i>HistoryRead</i> or the <i>Session</i> is closed. If the max continuation points have been reached the oldest continuation point will be reset.
historyData	Extensible Parameter HistoryData	The history data returned for the <i>Node</i> . The <i>HistoryData</i> parameter type is an extensible parameter type formally defined in [UA Part 11]. It specifies the types of history data that can be returned. The <i>ExtensibleParameter</i> base type is defined in Clause 8.2.
diagnosticInfos []	Diagnostic Info	List of diagnostic information. The size and order of the list matches the size and order of the <i>nodesToRead</i> request parameter. There is one entry in this list for each <i>Node</i> contained in the <i>nodesToRead</i> parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.10.3.3 Service results

Table 78 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 78 – HistoryRead Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.

### 5.10.3.4 StatusCodes

Table 79 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 79 – HistoryRead Operation Level Result Codes**

Symbolic Id	Description
Good_MoreData	More data is available in the time range beyond the number of values requested. This is an indication to reissue the read request using the <i>continuationPoint</i> parameter.
Good_NoData	No data was found in the specified time range.
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_DataEncodingInvalid	See Table 157 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 157 for the description of this result code.
Bad_NoTimestamp	The requested timestamp is not available for the <i>Node</i> .
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_ContinuationPointInvalid	See Table 156 for the description of this result code.

## 5.10.4 Write

### 5.10.4.1 Description

This Service is used to write values to one or more *Attributes* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this Service allows *Clients* to write the entire set of indexed values as a composite, to write individual elements or to write ranges of elements of the composite.

The sequence number in the Service request header is used by this Service to detect duplicate requests. *Servers* are responsible for tracking the sequence numbers used by the Service and for discarding requests that contain a sequence number that has already been used. *Servers* are expected to manage the rollover appropriately.

The values are written to the data source, such as a device, and the Service does not return until it writes the values or determines that the value cannot be written. In certain cases, the Server will successfully write to an intermediate system or Server, and will not know if the data source was updated properly. In these cases, the Server will report a success code that indicates that the write was not verified. In the cases where the Server is able to verify that it has successfully written to the data source, it reports an unconditional success.

It is possible that the Server may successfully write some *Attributes*, but not others. Rollback is the responsibility of the *Client*.

#### 5.10.4.2 Parameters

Table 80 defines the parameters for the Service.

**Table 80 – Write Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
nodesToWrite []	WriteValue	List of <i>Nodes</i> and their <i>Attributes</i> to write.
nodeId	NodeId	<i>NodeId</i> of the <i>Node</i> that contains the <i>Attributes</i> .
attributeId	IntegerId	Id of the <i>Attribute</i> . This must be a valid <i>Attribute</i> id. The <i>IntegerId</i> is defined in Clause 7.9. The <i>IntegerIds</i> for the <i>Attributes</i> are defined in [UA Part 6].
indexRange	NumericRange	This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. The first element is identified by index 0 (zero). The <i>NumericRange</i> type is defined in Clause 7.14. This parameter is not used if the specified <i>Attribute</i> is not an array or a structure. However, if the specified <i>Attribute</i> is an array or a structure, and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used.
value	DataValue	The <i>Node's Attribute</i> value (see Clause 7.4 for <i>DataValue</i> definition). If the <i>nodeAttributeId</i> parameter specifies a structure, an array or a range of array elements, then this parameter contains a composite value. If a null timestamp is supplied for the source timestamp or the <i>Server</i> timestamp with the value, the <i>Server</i> replaces these nulls with the time that the request was received. A Server must reject values that are not of the same type as the <i>Attribute's</i> value type.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of results for the <i>Nodes</i> to write (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>nodesToWrite</i> request parameter. There is one entry in this list for each <i>Node</i> contained in the <i>nodesToWrite</i> parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Nodes</i> to write (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>nodesToWrite</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.10.4.3 Service results

Table 81 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 81 – Write Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

#### 5.10.4.4 StatusCodes

Table 82 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 82 – Write Operation Level Result Codes**

Symbolic Id	Description
Good_CompletesAsynchronously	See Table 156 for the description of this result code. The value was successfully written to an intermediate system but the <i>Server</i> does not know if the data source was updated properly.
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_AttributeIdInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeNoData	See Table 157 for the description of this result code.
Bad_WriteNotSupported	The requested write operation is not supported. If a <i>Client</i> attempts to write any value, quality, timestamp combination and the <i>Server</i> does not support the requested combination (which could be a single quantity such as just timestamp), then the <i>Server</i> will not perform any write on this <i>Node</i> and will return this <i>StatusCode</i> for this <i>Node</i> .
Bad_NoWriteRights	See Table 157 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code. The current user does not have permission to write the attribute.
Bad_OutOfRange	See Table 157 for the description of this result code.
Bad_TypeMismatch	The value supplied for the attribute is not of the same type as the attribute's value.

#### 5.10.5 HistoryUpdate

##### 5.10.5.1 Description

This Service is used to update historical values or *Events* of one or more *Nodes*. Several request parameters indicate how the *Server* is to update the historical value or *Event*. Valid actions are Insert, Replace or Delete.

##### 5.10.5.2 Parameters

Table 83 defines the parameters for the Service.

**Table 83 – HistoryUpdate Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
historyUpdateDetails []	Extensible Parameter HistoryUpdate Details	The details defined for this update. The <i>HistoryUpdateDetails</i> parameter type is an extensible parameter type formally defined in [UA Part 11]. It specifies the types of history updates that can be performed. The <i>ExtensibleParameter</i> type is defined in Clause 8.2
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	HistoryUpdate Result	List of update results for the history update details. The size and order of the list matches the size and order of the details element of the <i>historyUpdateDetails</i> parameter specified in the request.
statusCode	StatusCodes	<i>StatusCodes</i> for the update of the <i>Node</i> (see Clause 7.28 for <i>StatusCodes</i> definition).
operationResults []	StatusCodes	List of <i>StatusCodes</i> for the operations to be performed on a <i>Node</i> . The size and order of the list matches the size and order of any list defined by the details element being reported by this <i>updateResults</i> entry.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the operations to be performed on a <i>Node</i> (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of any list defined by the details element being reported by this <i>updateResults</i> entry.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the history update details. The size and order of the list matches the size and order of the details element of the <i>historyUpdateDetails</i> parameter specified in the request.

### 5.10.5.3 Service results

Table 84 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 84 – HistoryUpdate Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

### 5.10.5.4 StatusCodes

Table 85 defines values for the *statusCode* and *operationResult* parameters that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 85 – HistoryUpdate Operation Level Result Codes**

Symbolic Id	Description
Good_ValueInserted	The value successfully inserted into history
Good_ValueReplaced	The value successfully replaced an existing value in history
Bad_ValueExists	A value already exists at the specified timestamp.
Bad_NoWriteRights	See Table 157 for the description of this result code.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.

## 5.11 Method Service Set

### 5.11.1 Overview

*Methods* represent the function calls of *Objects*. They are defined in [UA Part 3]. *Methods* are invoked and return only after completion (successful or unsuccessful). Execution times for methods may vary, depending on the function that they perform.

The *Method Service Set* defines the means to invoke methods. A *method* must be a *component* of an *Object*. Discovery is provided through the *browse* and *Query Services*. *Clients* discover the *methods* supported by a *Server* by browsing for the owning *Objects References* that identify their supported *methods*.

Because *Methods* may control some aspect of plant operations, method invocation may depend on environmental or other conditions. This may be especially true when attempting to re-invoke a method immediately after it has completed execution. Conditions that are required to invoke the method might not yet have returned to the state that permits the method to start again. In addition, some methods may support concurrent invocations, while others may have a single invocation executing at a given time. *Method Attributes* specify these behaviours.

### 5.11.2 Call

#### 5.11.2.1 Description

This Service is used to call (invoke) a *method*. Each *method* call is invoked within the context of an existing *Session*. If the *Session* is terminated, the results of the *method*'s execution cannot be returned to the *Client* and are discarded. This is independent of the task actually performed at the *Server*.

This Service provides for passing input and output arguments to/from a method. These arguments are defined by *Properties* of the method.

The sequence number in the Service request header is used by this Service to detect duplicate requests. *Servers* are responsible for tracking the sequence numbers used by this Service and for discarding requests that contain a sequence number that has already been used. *Clients* and *Servers* are expected to manage the rollover appropriately.

### 5.11.2.2 Parameters

Table 86 defines the parameters for the Service.

**Table 86 – Call Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
objectId	NodeId	<i>NodeId</i> of the <i>Object</i> that defines the <i>Method</i> . See [UA Part 3] for a description of <i>Objects</i> and their <i>Methods</i> .
methodId	NodeId	<i>NodeId</i> of the <i>Method</i> to invoke.
inputArguments []	BaseDataType	List of input argument values. An empty list indicates that there are no input arguments. The size and order of this list matches the size and order of the input arguments defined by the input <i>InputArguments Property</i> of the <i>Method</i> . The name, a description and the data type of each argument are defined by the <i>Argument</i> structure in each element of the method's <i>InputArguments Property</i> .
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
callResult	CallResult	Result of the <i>Method</i> call.
statusCode	StatusCodes	<i>StatusCodes</i> of the <i>Method</i> executed in the server. This <i>StatusCodes</i> is set to the <i>Bad_InvalidArgument StatusCode</i> if at least one input argument broke a constraint (e.g. wrong data type, value out of range). This <i>StatusCodes</i> is set to a bad <i>StatusCodes</i> if the <i>Method</i> execution failed in the server, e.g. based on an exception or an <i>HRESULT</i> .
inputArgumentResults []	StatusCodes	List of <i>StatusCodes</i> for each <i>inputArgument</i> .
inputArgumentDiagnosticInfos []	DiagnosticInfo	List of diagnostic information for each <i>inputArgument</i> .
diagnosticInfo	DiagnosticInfo	Diagnostic information for the <i>StatusCodes</i> of the <i>CallResult</i> .
outputArguments []	BaseDataType	List of output argument values. An empty list indicates that there are no output arguments. The size and order of this list matches the size and order of the output arguments defined by the <i>OutputArguments Property</i> of the <i>Method</i> . The name, a description and the data type of each argument are defined by the <i>Argument</i> structure in each element of the methods <i>OutputArguments Property</i> .

### 5.11.2.3 Service results

Table 87 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 87 – Call Service Result Codes**

Symbolic Id	Description
Bad_InvalidArgument	See Table 156 for the description of this result code.
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object.

## 5.12 MonitoredItem Service Set

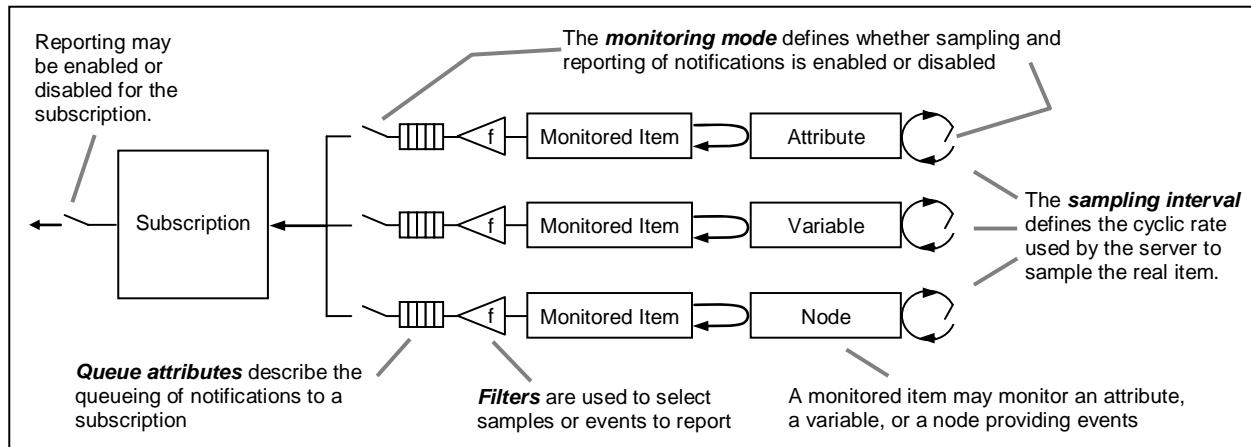
### 5.12.1 MonitoredItem model

#### 5.12.1.1 Overview

Clients define *MonitoredItems* to subscribe to data and *Events*. Each *MonitoredItem* identifies the item to be monitored and the *Subscription* to use to send *Notifications*. The item to be monitored may be an arbitrary *Node Attribute*.

*Notifications* are data structures that describe the occurrence of data changes and *Events*. They are packaged into *NotificationMessages* for transfer to the *Client*. The *Subscription* periodically sends *NotificationMessages* at a user-specified publishing interval, and the cycle during which these messages are sent is called a publishing cycle.

Four primary *Attributes* are defined for *MonitoredItems* that tell the *Server* how the item is to be sampled, evaluated and reported. These *Attributes* are the sampling interval, the monitoring mode, the filter and the queue *Attributes*. Figure 25 illustrates these concepts.



**Figure 25 – MonitoredItem Model**

This specification describes the monitoring of *Attributes* and *Variables* for value or status changes, including the caching of values and the monitoring of *Nodes* for *Events*.

*Attributes*, other than the *Value Attribute*, are monitored for a change in value. The filter is not used for these *Attributes*. Any change in value for these *Attributes* causes a *Notification* to be generated.

The *Value Attribute* is used when monitoring *Variables*. *Variable* values are monitored for a change in value or a change in their status. The filters defined in this specification (see Clause 8.3.2) and in [UA Part 8] are used to determine if the value change is large enough to cause a *Notification* to be generated for the *Variable*.

*Objects* and views can be used to monitor *Events*. *Events* are only available from *Nodes* where the *SubscribeToEvents* bit of the *EventNotifier Attribute* is set. The filter defined in this specification (see Clause 8.3.3) is used to determine if an *Event* received from the *Node* is sent to the *Client*. The filter also allows selecting *Properties* of the *EventType* that will be contained in the *Event* such as *EventId*, *EventType*, *SourceNode*, *Time* and *Description*.

[UA Part 3] describes the *Event* model and the base *EventTypes*.

The *Properties* of the base *EventTypes* and the representation of the base *EventTypes* in the *AddressSpace* are specified in [UA Part 5].

### 5.12.1.2 Sampling interval

Each *MonitoredItem* created by the *Client* is assigned a sampling interval that is either inherited from the publishing interval of the *Subscription* or that is defined specifically to override that rate. The sampling interval indicates the fastest rate at which the *Server* should sample its underlying source for data changes.

The assigned sampling interval defines a “best effort” cyclic rate that the *Server* uses to sample the item from its source. “Best effort” in this context means that the *Server* does its best to sample at this rate. Sampling at rates faster than this rate is acceptable, but not necessary to meet the needs

of the *Client*. How the *Server* deals with the sampling rate and how often it actually polls its data source internally is a *Server* implementation detail.

The *Client* may also specify 0 for the sampling interval, which indicates that the *Server* should use the fastest practical rate. It is expected that *Servers* will support only a limited set of sampling intervals to optimize their operation. If the exact interval requested by the *Client* is not supported by the *Server*, then the *Server* assigns to the *MonitoredItem* the most appropriate interval as determined by the *Server*. It returns this assigned interval to the *Client*. The *Server Capabilities Object* defined in [UA Part 5] identifies the sampling intervals supported by the *Server*.

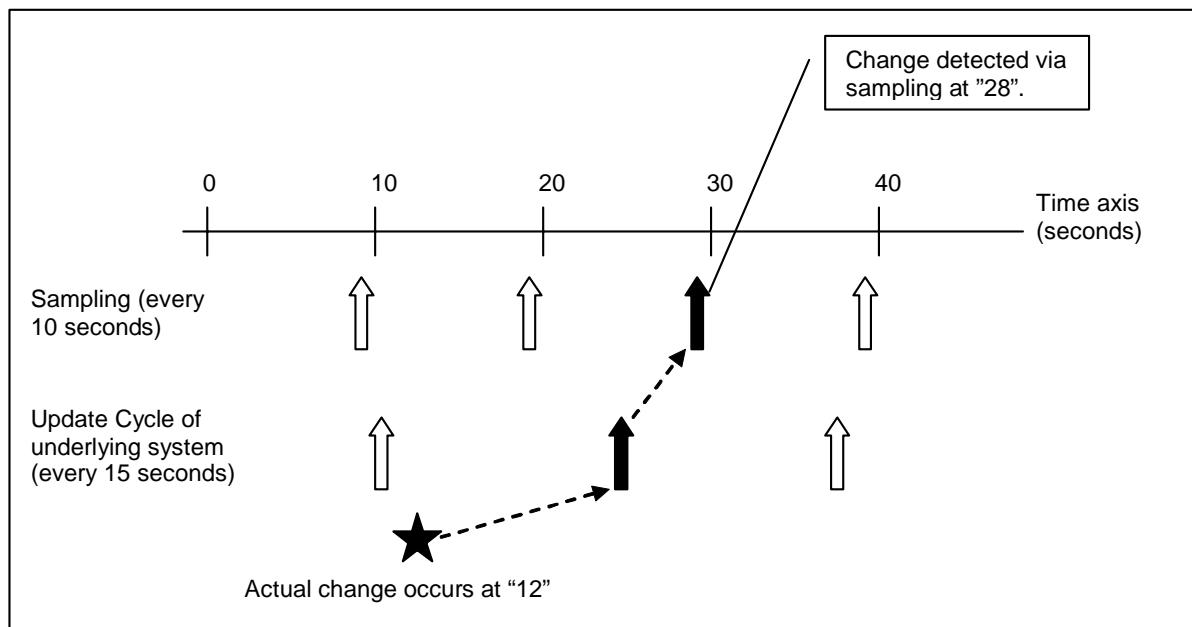
The *Server* may support data that is collected based on a sampling model or generated based on an exception-based model. The fastest supported sampling interval may be equal to 0, which indicates that the data item is exception-based rather than being sampled at some period. Exception-based means that the underlying system reports changes of the data.

The *Client* may use the revised sampling interval values as a hint for setting the publishing interval as well as the keep alive count of a *Subscription*. If, for example, the smallest revised sampling interval of the *MonitoredItems* is 5 seconds, then the time before a keep-alive is sent should be longer than 5 seconds.

Note that, in many cases, the UA *Server* provides access to a decoupled system and therefore has no knowledge of the data update logic. In this case, even though the UA *Server* samples at the negotiated rate, the data might be updated by the underlying system at a much slower rate. In this case, changes can only be detected at this slower rate.

If the behaviour by which the underlying system updates the item is known, it will be available via the *MinSamplingInterval Attribute* defined in [UA Part 3].

*Clients* should also be aware that the sampling by the UA *Server* and the update cycle of the underlying system are usually not synchronized. This can cause additional delays in change detection, as illustrated in Figure 26.



**Figure 26 – Typical delay in change detection.**

#### 5.12.1.3 Monitoring mode

The monitoring mode *Attribute* is used to enable and disable the sampling of a *MonitoredItem*, and also to provide for independently enabling and disabling the reporting of *Notifications*. This capability allows a *MonitoredItem* to be configured to sample, sample and report, or neither.

Disabling sampling does not change the values of any of the other *MonitoredItem Attributes*, such as its sampling interval.

#### 5.12.1.4 Filter

Each time a *MonitoredItem* is sampled, the *Server* evaluates the sample using the filter defined for the *MonitoredItem*. The filter *Attribute* defines the criteria that the *Server* uses to determine if a *Notification* should be generated for the sample. The type of filter is dependent on the type of the item that is being monitored. For example, the *Deadband* filter is used when monitoring *Variables* and the *EventFilter* is used when monitoring *Events*. Sampling and evaluation, including the use of filters, are described in this specification. Additional filters may be defined in other parts of this multi-part specification.

#### 5.12.1.5 Queue Attributes

If the sample passes the filter criteria, a *Notification* is generated and queued for transfer by the *Subscription*. The size of the queue is defined when the *MonitoredItem* is created. When the queue is full and a new *Notification* is received, the *Server* either discards the oldest *Notification* and queues the new one, or it simply discards the new one. The *MonitoredItem* is configured for one of these discard policies when the *MonitoredItem* is created. If a *Notification* is discarded for a *DataValue*, the *Overflow* bit in the *InfoBits* portion of the *DataValue statusCode* is set.

If the queue size is one and if the discard policy is to discard the oldest, the queue becomes a buffer that always contains the newest *Notification*. In this case, if the sampling interval of the *MonitoredItem* is faster than the publishing interval of the *Subscription*, the *MonitoredItem* will be over sampling and the *Client* will always receive the most up-to-date value.

On the other hand, the *Client* may want to subscribe to a continuous stream of *Notifications* without any gaps, but does not want them reported at the sampling interval. In this case, the *MonitoredItem* would be created with a queue size large enough to hold all *Notifications* generated between two consecutive publishing cycles. Then, at each publishing cycle, the *Subscription* would send all *Notifications* queued for the *MonitoredItem* to the *Client*. The *Server* is required to return values for any particular item in chronological order.

The *Server* may be sampling at a faster rate than the sampling interval to support other *Clients*; the *Client* should only expect values at the negotiated sampling interval. The *Server* may deliver fewer values than dictated by the sampling interval, based on the filter and implementation constraints. If a *Deadband* filter is configured for a *MonitoredItem*, it is always applied to the newest value in the queue compared to the current sample.

If, for example, the *AbsoluteDeadband* in the *DataChangeFilter* is “10”, the queue could consist of values in the following order:

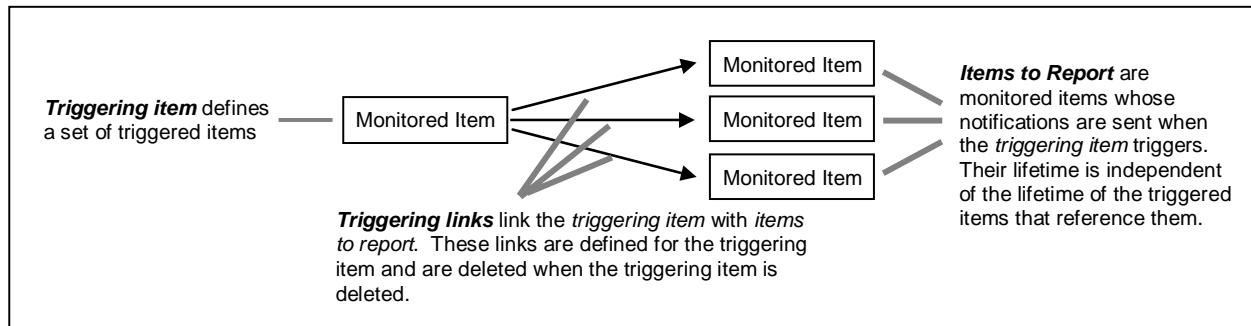
- 100
- 111
- 101
- 89
- 100

Queuing of data may result in unexpected behaviour when using a *Deadband* filter and the number of encountered changes is larger than the number of values that can be maintained. It is realistically possible that, due to the discard policy “*discardOldest=TRUE*”, the new first value in the queue will not exceed the *Deadband* limit of the previous value sent to the *Client*.

The queue size is the maximum value supported by the *Server* when monitoring *Events*. In this case, the *Server* is responsible for the *Event* buffer. If *Events* are lost, an *Event* of the type *EventQueueOverflow* is generated.

### 5.12.1.6 Triggering model

The *MonitoredItems* Service allows adding items that are reported only when some other item (the triggering item) triggers. This is done by creating links between the triggered items and the items to report. The monitoring mode of the items to report is set to sampling-only so that it will sample and queue *Notifications* without reporting them. Figure 27 illustrates this concept.



**Figure 27 – Triggering Model**

The triggering mechanism is a useful feature that allows *Clients* to reduce the data volume on the wire by configuring some items to sample frequently but only report when some other *Event* happens.

The following triggering behaviours are specified:

- The monitoring mode of the triggering item indicates that reporting is disabled. In this case, the triggering item is not reported when the triggering item triggers.
- The monitoring mode of the triggering item indicates that reporting is enabled. In this case, the triggering item is reported when the triggering item triggers.
- The monitoring mode of the item to report indicates that reporting is disabled. In this case, the item to report is reported when the triggering item triggers.
- The monitoring mode of the item to report indicates that reporting is enabled. In this case, the item to report is reported only once (when the item to report triggers), effectively causing the triggering item to be ignored.

*Clients* create and delete triggering links between a triggering item and a set of items to report. If the *MonitoredItem* that represents an item to report is deleted before its associated triggering link is deleted, the triggering link is also deleted, but the triggering item is otherwise unaffected.

Deletion of a *MonitoredItem* should not be confused with the removal of the *Attribute* that it monitors. If the *Node* that contains the *Attribute* being monitored is deleted, the *MonitoredItem* generates a *Notification* with a *StatusCode Bad\_UnknownNodeId* that indicates the deletion, but the *MonitoredItem* is not deleted.

## 5.12.2 CreateMonitoredItems

### 5.12.2.1 Description

This Service is used to create and add one or more *MonitoredItems* to a *Subscription*. A *MonitoredItem* is deleted automatically by the *Server* when the *Subscription* is deleted. Deleting a *MonitoredItem* causes its entire set of triggered item links to be deleted, but has no effect on the *MonitoredItems* referenced by the triggered items.

Calling the *CreateMonitoredItems* Service repetitively to add a small number of *MonitoredItems* each time may adversely affect the performance of the *Server*. *Servers* may introduce delays between repetitive calls to this Service or enforce other measures to discourage this behaviour. *Clients* are cautioned to not use this Service in this manner. Instead, *Clients* should add a complete set of *MonitoredItems* to a *Subscription* whenever possible.

When a *MonitoredItem* is added, the Server performs initialization processing for it. The initialization processing is defined by the *Notification* type of the item being monitored. *Notification* types are specified in this specification and in the Access Types Parts of this multi-part specification, such as [UA Part 8]. See Clause 4 of [UA Part 1] for a description of the Access Type Parts.

When a user adds a monitored item that the user is denied read access to, the add operation for the item must succeed and the bad status *Bad\_NoReadRights* or *Bad\_UserAccessDenied* will be returned in the Publish response. This is the same behaviour for the case where the access rights are changed after the call to *CreateMonitoredItem*. If the access rights change to read rights, the Server must start sending data for the *MonitoredItem*.

### 5.12.2.2 Parameters

Table 88 defines the parameters for the Service.

**Table 88 – CreateMonitoredItems Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> that will report <i>Notifications</i> for this <i>MonitoredItem</i> (see Clause 7.9 for <i>IntegerId</i> definition).
timestampsToReturn	enum Timestamps ToReturn	An enumeration that specifies the timestamp <i>Attributes</i> to be transmitted for each <i>MonitoredItem</i> . The <i>TimestampsToReturn</i> enumeration is defined in Clause 7.29. When monitoring <i>Events</i> , this applies only to <i>Event</i> fields that are of type <i>DataValue</i> .
itemsToCreate []	MonitoredItem CreateRequest	A list of <i>MonitoredItems</i> to be created and assigned to the specified <i>Subscription</i> .
itemToMonitor	ReadValueId	Identifies an item in the <i>AddressSpace</i> to monitor. To monitor for <i>Events</i> , the <i>attributeId</i> element of the <i>ReadValueId</i> structure is the id of the <i>EventNotifier Attribute</i> . The <i>ReadValueId</i> type is defined in Clause 7.16.
monitoringMode	enum MonitoringMode	The monitoring mode to be set for the <i>MonitoredItem</i> . The <i>MonitoringMode</i> enumeration is defined in Clause 7.12.
requestedAttributes	Monitoring Attributes	The requested monitoring <i>Attributes</i> . Servers negotiate the values of these <i>Attributes</i> based on the override policy of the <i>Subscription</i> and the capabilities of the Server. The <i>MonitoringAttributes</i> type is defined in Clause 7.10.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	MonitoredItem CreateResult	List of results for the <i>MonitoredItems</i> to create. The size and order of the list matches the size and order of the <i>itemsToCreate</i> request parameter.
statusCode	StatusCodes	<i>StatusCodes</i> for the <i>MonitoredItem</i> to create (see Clause 7.28 for <i>StatusCodes</i> definition).
monitoredItemId	IntegerId	Server-assigned id for the <i>MonitoredItem</i> (see Clause 7.9 for <i>IntegerId</i> definition). This id is unique within the <i>Subscription</i> , but might not be unique within the Server or Session. This parameter is present only if the <i>statusCode</i> indicates that the <i>MonitoredItem</i> was successfully created.
revisedSampling Interval	Duration	The actual sampling interval that the Server will use (see Clause 7.6 for <i>Duration</i> definition). This value is based on a number of factors, including capabilities of the underlying system.
revisedQueueSize	Counter	The actual queue size that the Server will use.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>MonitoredItems</i> to create (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>itemsToCreate</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.12.2.3 Service results

Table 89 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 89 – CreateMonitoredItems Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 156 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

#### 5.12.2.4 StatusCodes

Table 90 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 90 – CreateMonitoredItems Operation Level Result Codes**

Symbolic Id	Description
Bad_MonitoringModelError	See Table 157 for the description of this result code.
Bad_NodeIdInvalid	See Table 157 for the description of this result code.
Bad_NodeIdUnknown	See Table 157 for the description of this result code.
Bad_AttributeIdInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeInvalid	See Table 157 for the description of this result code.
Bad_IndexRangeNoData	See Table 157 for the description of this result code.
Bad_DataEncodingInvalid	See Table 157 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 157 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.
Bad_FilterNotAllowed	See Table 156 for the description of this result code.

#### 5.12.3 ModifyMonitoredItems

##### 5.12.3.1 Description

This Service is used to modify *MonitoredItems* of a *Subscription*. Changes to the sampling interval and filter take effect at the beginning of the next sampling interval (the next time the sampling timer expires).

### 5.12.3.2 Parameters

Table 91 defines the parameters for the Service.

**Table 91 – ModifyMonitoredItems Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> used to qualify the <i>monitoredItemId</i> (see Clause 7.9 for <i>IntegerId</i> definition).
timestampsToReturn	enum Timestamps ToReturn	An enumeration that specifies the timestamp <i>Attributes</i> to be transmitted for each <i>MonitoredItem</i> to be modified. The <i>TimestampsToReturn</i> enumeration is defined in Clause 7.29. When monitoring <i>Events</i> , this applies only to <i>Event</i> fields that are of type <i>DataValue</i> .
itemsToModify []	MonitoredItemModifyRequest	The list of <i>MonitoredItems</i> to modify.
monitoredItemId	IntegerId	Server-assigned id for the <i>MonitoredItem</i> .
requestedAttributes	Monitoring Attributes	The requested values for the monitoring <i>Attributes</i> . The <i>MonitoringAttributes</i> type is defined in Clause 7.10.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	MonitoredItemModifyResult	List of results for the <i>MonitoredItems</i> to modify. The size and order of the list matches the size and order of the <i>itemsToModify</i> request parameter.
statusCode	StatusCodes	<i>StatusCodes</i> for the <i>MonitoredItem</i> to be modified (see Clause 7.28 for <i>StatusCodes</i> definition).
revisedSamplingInterval	Duration	The actual sampling interval that the Server will use (see Clause 7.6 for <i>Duration</i> definition). The Server returns the value it will actually use for the sampling interval. This value is based on a number of factors, including capabilities of the underlying system.
revisedQueueSize	Counter	The actual queue size that the Server will use.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>MonitoredItems</i> to modify (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>itemsToModify</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.12.3.3 Service results

Table 92 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 92 – ModifyMonitoredItems Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 156 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.12.3.4 StatusCodes

Table 93 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 93 – ModifyMonitoredItems Operation Level Result Codes**

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 157 for the description of this result code.
Bad_ExtensibleParameterInvalid	See Table 156 for the description of this result code.
Bad_ExtensibleParameterUnsupported	See Table 156 for the description of this result code.
Bad_FilterNotAllowed	See Table 156 for the description of this result code.

### 5.12.4 SetMonitoringMode

#### 5.12.4.1 Description

This Service is used to set the monitoring mode for one or more *MonitoredItems* of a *Subscription*. Setting the mode to DISABLED or SAMPLING causes all queued *Notifications* to be deleted.

#### 5.12.4.2 Parameters

Table 94 defines the parameters for the Service.

**Table 94 – SetMonitoringMode Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> used to qualify the <i>monitoredItemIds</i> (see Clause 7.9 for <i>IntegerId</i> definition).
monitoringMode	enum MonitoringMode	The monitoring mode to be set for the <i>MonitoredItems</i> . The <i>MonitoringMode</i> enumeration is defined in Clause 7.12.
monitoredItemIds []	IntegerId	List of Server-assigned ids for the <i>MonitoredItems</i> .
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>MonitoredItems</i> to enable/disable (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>monitoredItemIds</i> request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>MonitoredItems</i> to enable/disable (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>monitoredItemIds</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.12.4.3 Service results

Table 95 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 95 – SetMonitoringMode Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.
Bad_MonitoringModelInvalid	See Table 157 for the description of this result code.

#### 5.12.4.4 StatusCodes

Table 96 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 96 – SetMonitoringMode Operation Level Result Codes**

Value	Description
Bad_MonitoredItemIdInvalid	See Table 157 for the description of this result code.

### 5.12.5 SetTriggering

#### 5.12.5.1 Description

This Service is used to create and delete triggering links for a triggering item. The triggering item and the items to report must belong to the same *Subscription*.

Each triggering link links a triggering item to an item to report. Each link is represented by the *MonitoredItem* id for the item to report. An error code is returned if this id is invalid.

### 5.12.5.2 Parameters

Table 97 defines the parameters for the Service.

**Table 97 – SetTriggering Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> that contains the triggering item and the items to report (see Clause 7.9 for <i>IntegerId</i> definition).
triggeringItemId	IntegerId	Server-assigned id for the <i>MonitoredItem</i> used as the triggering item.
linksToAdd []	IntegerId	The list of Server-assigned ids of the items to report that are to be added as triggering links.
linksToRemove []	IntegerId	The list of Server-assigned ids of the items to report for the triggering links to be deleted.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
addResults []	StatusCodes	List of <i>StatusCodes</i> for the items to add (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>linksToAdd</i> parameter specified in the request.
addDiagnosticInfos []	Diagnostic Info	List of diagnostic information for the links to add (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>linksToAdd</i> request parameter. This list is empty if diagnostics information was not requested in the request header.
removeResults []	StatusCodes	List of <i>StatusCodes</i> for the items to delete. The size and order of the list matches the size and order of the <i>linksToDelete</i> parameter specified in the request.
removeDiagnosticInfos []	Diagnostic Info	List of diagnostic information for the links to delete. The size and order of the list matches the size and order of the <i>linksToDelete</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.12.5.3 Service results

Table 98 defines the Service results specific to this Service. Common *StatusCodes* are defined in Clause 7.28.

**Table 98 – SetTriggering Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.
Bad_MonitoredItemIdInvalid	See Table 157 for the description of this result code.

### 5.12.5.4 StatusCodes

Table 99 defines values for the results parameters that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 99 – SetTriggering Operation Level Result Codes**

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 157 for the description of this result code.

## 5.12.6 DeleteMonitoredItems

### 5.12.6.1 Description

This Service is used to remove one or more *MonitoredItems* of a *Subscription*. When a *MonitoredItem* is deleted, its triggered item links are also deleted.

Successful removal of a *MonitoredItem*, however, might not remove *Notifications* for the *MonitoredItem* that are in the process of being sent by the *Subscription*. Therefore, *Clients* may receive *Notifications* for the *MonitoredItem* after they have received a positive response that the *MonitoredItem* has been deleted.

### 5.12.6.2 Parameters

Table 100 defines the parameters for the Service.

**Table 100 – DeleteMonitoredItems Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> that contains the <i>MonitoredItems</i> to be deleted (see Clause 7.9 for <i>IntegerId</i> definition).
monitoredItemIds []	IntegerId	List of Server-assigned ids for the <i>MonitoredItems</i> to be deleted.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>MonitoredItems</i> to delete (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>monitoredItemIds</i> request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>MonitoredItems</i> to delete (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>monitoredItemIds</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.12.6.3 Service results

Table 101 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 101 – DeleteMonitoredItems Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.12.6.4 StatusCodes

Table 102 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 102 – DeleteMonitoredItems Operation Level Result Codes**

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 157 for the description of this result code.

## 5.13 Subscription Service Set

### 5.13.1 Subscription model

#### 5.13.1.1 Description

*Subscriptions* are used to report *Notifications* to the *Client*. Their general behaviour is summarized below. Their precise behaviour is described in Clause 5.13.1.2.

- a) *Subscriptions* have a set of *MonitoredItems* assigned to them by the *Client*. *MonitoredItems* generate *Notifications* that are to be reported to the *Client* by the *Subscription* (see Clause 5.12.1 for a description of *MonitoredItems*).
- b) *Subscriptions* have a publishing interval. The publishing interval of a *Subscription* defines the cyclic rate at which the *Subscription* executes. Each time it executes, it attempts to send a *NotificationMessage* to the *Client*. *NotificationMessages* contain *Notifications* that have not yet been reported to *Client*.
- c) *NotificationMessages* are sent to the *Client* in response to *Publish* requests. *Publish* requests are normally queued to the *Session* as they are received, and one is dequeued and processed by a subscription related to this *Session* each publishing cycle, if there are *Notifications* to report. When there are not, the *Publish* request is not dequeued from the *Session*, and the *Server* waits until the next cycle and checks again for *Notifications*.
- d) At the beginning of a cycle, if there are *Notifications* to send but there are no *Publish* requests queued, the *Server* enters a wait state for a *Publish* request to be received. When one is received, it is processed immediately without waiting for the next publishing cycle.
- e) *NotificationMessages* are uniquely identified by sequence numbers that enable *Clients* to detect missed *Messages*. The publishing interval also defines the default sampling interval for its *MonitoredItems*.
- f) *Subscriptions* have a keep-alive counter that counts the number of consecutive publishing cycles in which there have been no *Notifications* to report to the *Client*. When the maximum keep-alive count is reached, a *Publish* request is dequeued and used to return a keep-alive *Message*. This keep-alive *Message* informs the *Client* that the *Subscription* is still active. Each keep-alive *Message* is a response to a *Publish* request in which the *notificationMessage* parameter does not contain any *Notifications* and that contains the sequence number of the next *NotificationMessage* that is to be sent. In the sections that follow, the term *NotificationMessage* refers to a response to a *Publish* request in which the *notificationMessage* parameter actually contains one or more *Notifications*, as opposed to a keep-alive *Message* in which this parameter contains no *Notifications*. The maximum keep-alive count is set by the *Client* during *Subscription* creation and may be subsequently modified using the *ModifySubscription* Service. Similar to *Notification* processing described in (c) above, if there are no *Publish* requests queued, the *Server* waits for the next one to be received and sends the keep-alive immediately without waiting for the next publishing cycle.
- g) Publishing by a *Subscription* may be enabled or disabled by the *Client* when created, or subsequently using the *SetPublishingMode* Service. Disabling causes the *Subscription* to cease sending *NotificationMessages* to the *Client*. However, the *Subscription* continues to execute cyclically and continues to send keep-alive *Messages* to the *Client*.
- h) *Subscriptions* have a lifetime counter that counts the number of consecutive publishing cycles in which there have been no *Publish* requests received from the *Client*. When this counter reaches the value calculated for the lifetime of a *Subscription* based on the *MaxKeepAliveCount* parameter in the *CreateSubscription* Service (Clause 5.13.2), the *Subscription* is closed. Closing the *Subscription* causes its *MonitoredItems* to be deleted.
- i) *Subscriptions* maintain a retransmission queue of sent *NotificationMessages*. *NotificationMessages* are retained in this queue until they are acknowledged or until they have been in the queue for a minimum of one keep-alive interval. *Clients* are required to acknowledge *NotificationMessages* as they are received.

The sequence number is an unsigned 32-bit integer that is incremented by one for each *NotificationMessage* sent. The value 0 is never used for the sequence number. The first *NotificationMessage* sent on a *Subscription* has a sequence number of 1. If the sequence number rolls over, it rolls over to 1.

When a *Subscription* is created, the first *Message* is sent at the end of the first publishing cycle to inform the *Client* that the *Subscription* is operational. A *NotificationMessage* is sent if there are *Notifications* ready to be reported. If there are none, a keep-alive *Message* is sent instead that contains a sequence number of 1, indicating that the first *NotificationMessage* has not yet been sent. This is the only time a keep-alive *Message* is sent without waiting for the maximum keep-alive count to be reached, as specified in (f) above.

The value of the sequence number is never reset during the lifetime of a *Subscription*. Therefore, the same sequence number will not be reused on a *Subscription* until over four billion *NotificationMessages* have been sent. At a continuous rate of one thousand *NotificationMessages* per second on a given *Subscription*, it would take roughly fifty days for the same sequence number to be reused. This allows *Clients* to safely treat sequence numbers as unique.

Sequence numbers are also used by *Clients* to acknowledge the receipt of *NotificationMessages*. *Publish* requests allow the *Client* to acknowledge all *Notifications* up to a specific sequence number and to acknowledge the sequence number of the last *NotificationMessage* received. One or more gaps may exist in between. Acknowledgements allow the *Server* to delete *NotificationMessages* from its retransmission queue.

*Clients* may ask for retransmission of selected *NotificationMessages* using the Republish Service. This Service returns the requested *Message*.

#### 5.13.1.2 State table

The state table formally describes the operation of the *Subscription*. The following model of operations is described by this state table. This description applies when publishing is enabled or disabled for the *Subscription*.

After creation of the *Subscription*, the *Server* starts the publishing timer and restarts it whenever it expires. If the timer expires the number of times defined for the *Subscription* lifetime without having received a *Subscription Service* request from the *Client*, the *Subscription* assumes that the *Client* is no longer present, and terminates.

*Clients* send *Publish* requests to *Servers* to receive *Notifications*. *Publish* requests are not directed to any one *Subscription* and, therefore, may be used by any *Subscription*. Each contains acknowledgements for one or more *Subscriptions*. These acknowledgements are processed when the *Publish* request is received. The *Server* then queues the request in a queue shared by all *Subscriptions*, except in the following cases:

- a) The previous *Publish* response indicated that there were still more *Notifications* ready to be transferred and there were no more *Publish* requests queued to transfer them.
- b) The publishing timer of a *Subscription* expired and there were either *Notifications* to be sent or a keep-alive *Message* to be sent.

In these cases, the newly received *Publish* request is processed immediately by the first *Subscription* to encounter either case (a) or case (b).

Each time the publishing timer expires, it is immediately reset. If there are *Notifications* or a keep-alive *Message* to be sent, it dequeues and processes a *Publish* request. When a *Subscription* processes a *Publish* request, it accesses the queues of its *MonitoredItems* and dequeues its *Notifications*, if any. It returns these *Notifications* in the response, setting the *moreNotifications* flag if it was not able to return all available *Notifications* in the response.

If there were *Notifications* or a keep-alive *Message* to be sent but there were no *Publish* requests queued, the *Subscription* assumes that the *Publish* request is late and waits for the next *Publish* request to be received, as described in case (b).

If the *Subscription* is disabled when the publishing timer expires or if there are no *Notifications* available, it enters the keep-alive state and sets the keep-alive counter to its maximum value as defined for the *Subscription*.

While in the keep-alive state, it checks for *Notifications* each time the publishing timer expires. If one or more have been generated, a *Publish* request is dequeued and a *NotificationMessage* is returned in the response. However, if the publishing timer reaches the maximum keep-alive count without a *Notification* becoming available, a *Publish* request is dequeued and a keep-alive *Message* is returned in the response. The *Subscription* then returns to the normal state of waiting for the publishing timer to expire again. If, in either of these cases, there are no *Publish* requests queued, the *Subscription* waits for the next *Publish* request to be received, as described in case (b).

The *Subscription* states are defined in Table 103.

**Table 103 – Subscription States**

State	Description
CLOSED	The <i>Subscription</i> has not yet been created or has terminated
CREATING	The <i>Subscription</i> is being created.
NORMAL	The <i>Subscription</i> is cyclically checking for <i>Notifications</i> from its <i>MonitoredItems</i> . The keep-alive counter is not used in this state.
LATE	The publishing timer has expired and there are <i>Notifications</i> available or a keep-alive <i>Message</i> is ready to be sent, but there are no <i>Publish</i> requests queued. When in this state, the next <i>Publish</i> request is processed when it is received. The keep-alive counter is not used in this state.
KEEPALIVE	The <i>Subscription</i> is cyclically checking for <i>Notifications</i> from its <i>MonitoredItems</i> or for the keep-alive counter to count down to 0 from its maximum.

The state table is described in Table 104. The following rules and conventions apply:

- a) Events represent the receipt of Service requests and the occurrence internal Events, such as timer expirations.
- b) Service requests Events may be accompanied by conditions that test Service parameter values. Parameter names begin with a lower case letter.
- c) Internal Events may be accompanied by conditions that test state Variable values. State Variables are defined in Clause 5.13.1.3. They begin with an upper case letter.
- d) Service request and internal Events may be accompanied by conditions represented by functions whose return value is tested. Functions are identified by “()” after their name. They are described in Clause 5.13.1.4.
- e) When an Event is received, the first transition for the current state is located and the transitions are searched sequentially for the first transition that meets the Event or conditions criteria. If none are found, the Event is ignored.
- f) Actions are described by functions and state Variable manipulations.
- g) The LifetimeTimerExpires Event is triggered when its corresponding counter reaches zero.

**Table 104 – Subscription State Table**

#	Current State	Event/Conditions	Action	Next State
1	CLOSED	Receive CreateSubscription Request	CreateSubscription()	CREATING
2	CREATING	CreateSubscription fails	ReturnNegativeResponse()	CLOSED
3	CREATING	CreateSubscription succeeds	InitializeSubscription() MessageSent = FALSE ReturnResponse()	NORMAL
4	NORMAL	Receive <i>Publish</i> Request && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && MoreNotifications == FALSE) )	ResetLifetimeCounter() DeleteAckedNotificationMsgs() EnqueuePublishingReq()	NORMAL
5	NORMAL	Receive <i>Publish</i> Request && PublishingEnabled == TRUE && MoreNotifications == TRUE	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
6	NORMAL	PublishingTimer Expires && PublishingReqQueued == TRUE && PublishingEnabled == TRUE && NotificationsAvailable == TRUE	StartPublishingTimer() DequeuePublishReq() ReturnNotifications() MessageSent == TRUE	NORMAL
7	NORMAL	PublishingTimer Expires && PublishingReqQueued == TRUE && MessageSent == FALSE && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == FALSE) )	StartPublishingTimer() DequeuePublishReq() ReturnKeepAlive() MessageSent == TRUE	NORMAL
8	NORMAL	PublishingTimer Expires && PublishingReqQueued == FALSE && ( MessageSent == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == TRUE) )	StartPublishingTimer()	LATE
9	NORMAL	PublishingTimer Expires && MessageSent == TRUE && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == FALSE) )	StartPublishingTimer() ResetKeepAliveCounter()	KEEPALIVE
10	LATE	Receive <i>Publish</i> Request && PublishingEnabled == TRUE && (NotificationsAvailable == TRUE    MoreNotifications == TRUE)	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
11	LATE	Receive <i>Publish</i> Request && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == FALSE && MoreNotifications == FALSE) )	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnKeepAlive() MessageSent = TRUE	KEEPALIVE
12	LATE	PublishingTimer Expires	StartPublishingTimer()	LATE

#	Current State	Event/Conditions	Action	Next State
13	KEEPALIVE	Receive <i>Publish Request</i>	ResetLifetimeCounter() DeleteAckedNotificationMsgs() EnqueuePublishingReq()	KEEPALIVE
14	KEEPALIVE	PublishingTimer Expires && PublishingEnabled == TRUE && NotificationsAvailable == TRUE && PublishingReqQueued == TRUE	StartPublishingTimer() DequeuePublishReq() ReturnNotifications() MessageSent == TRUE	NORMAL
15	KEEPALIVE	PublishingTimer Expires && PublishingReqQueued == TRUE && KeepAliveCounter == 1 && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == FALSE) )	StartPublishingTimer() DequeuePublishReq() ReturnKeepAlive() ResetKeepAliveCounter()	KEEPALIVE
16	KEEPALIVE	PublishingTimer Expires && KeepAliveCounter > 1 && ( PublishingEnabled == FALSE    (PublishingEnabled == TRUE && NotificationsAvailable == FALSE) )	StartPublishingTimer() KeepAliveCounter--	KEEPALIVE
17	KEEPALIVE	PublishingTimer Expires && PublishingReqQueued == FALSE && ( KeepAliveCounter == 1    (KeepAliveCounter > 1 && PublishingEnabled == TRUE && NotificationsAvailable == TRUE) )	StartPublishingTimer()	LATE

#	Current State    LATE    KEEPALIVE	Event/Conditions	Action	Next State
18	NORMAL    LATE    KEEPALIVE	Receive ModifySubscription Request	ResetLifetimeCounter() UpdateSubscriptionParams() ReturnResponse()	SAME
19	NORMAL    LATE    KEEPALIVE	Receive SetPublishingMode Request	ResetLifetimeCounter() SetPublishingEnabled() MoreNotifications = FALSE ReturnResponse()	SAME
20	NORMAL    LATE    KEEPALIVE	Receive Republish Request && RequestedMessageFound == TRUE	ResetLifetimeCounter() ReturnResponse()	SAME
21	NORMAL    LATE    KEEPALIVE	Receive Republish Request && RequestedMessageFound == FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
22	NORMAL    LATE    KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == FALSE	ResetLifetimeCounter() ReturnNegativeResponse ()	SAME
23	NORMAL    LATE    KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == TRUE && ClientValidated() ==TRUE	SetSession() ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnResponse()	SAME
24	NORMAL    LATE    KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == TRUE && ClientValidated() == FALSE	ReturnNegativeResponse()	SAME
25	NORMAL    LATE    KEEPALIVE	Receive DeleteSubscriptions Request && SubscriptionAssignedToClient ==TRUE	DeleteMonitoredItems() DeleteClientPublReqQueue()	CLOSED
26	NORMAL    LATE    KEEPALIVE	Receive DeleteSubscriptions Request && SubscriptionAssignedToClient ==FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
27	NORMAL    LATE    KEEPALIVE	LifetimeTimer Expires	DeleteMonitoredItems()	CLOSED

### 5.13.1.3 State Variables and parameters

The state *Variables* are defined alphabetically in Table 105.

**Table 105 – State variables and parameters**

State Variable	Description
MoreNotifications	A boolean <i>Variable</i> that is set to TRUE only by the CreateNotificationMsg() when there were too many <i>Notifications</i> for a single <i>NotificationMessage</i> .
LatePublishRequest	A boolean <i>Variable</i> that is set to TRUE to reflect that, the last time the publishing timer expired, there were no <i>Publish</i> requests queued.
LifetimeCounter	A <i>Variable</i> that contains the number of consecutive publishing timer expirations without <i>Client</i> activity before the <i>Subscription</i> is terminated.
MessageSent	A boolean <i>Variable</i> that is set to TRUE to mean that either a <i>NotificationMessage</i> or a keep-alive <i>Message</i> has been sent on the <i>Subscription</i> . It is a flag that is used to ensure that either a <i>NotificationMessage</i> or a keep-alive <i>Message</i> is sent out the first time the publishing timer expires.
NotificationsAvailable	A boolean <i>Variable</i> that is set to TRUE only when there is at least one <i>MonitoredItem</i> that is in the reporting mode and that has a <i>Notification</i> queued or there is at least one item to report whose triggering item has triggered and that has a <i>Notification</i> queued. The transition of this state <i>Variable</i> from FALSE to TRUE creates the “New Notification Queued” <i>Event</i> in the state table.
PublishingEnabled	The parameter that requests publishing to be enabled or disabled.
PublishingReqQueued	A boolean <i>Variable</i> that is set to TRUE only when there is a <i>Publish</i> request <i>Message</i> enqueued to the <i>Subscription</i> .
RequestedMessageFound	A boolean <i>Variable</i> that is set to TRUE only when the <i>Message</i> requested to be retransmitted was found in the retransmission queue.
SeqNum	The <i>Variable</i> that records the value of the sequence number used in <i>NotificationMessages</i>
SubscriptionAssignedToClient	A boolean <i>Variable</i> that is set to TRUE only when the <i>Subscription</i> requested to be deleted is assigned to the <i>Client</i> that issued the request. A <i>Subscription</i> is assigned to the <i>Client</i> that created it. That assignment can only be changed through successful completion of the TransferSubscriptions Service.

### 5.13.1.4 Functions

The action functions are defined alphabetically in Table 106.

**Table 106 – Functions**

State	Description
BindSession()	Bind the <i>Client Session</i> associated with the <i>Subscription</i> to the <i>Client Session</i> used to send the Service being processed. If this was the last <i>Subscription</i> bound to the previous <i>Client</i> , clear the <i>Publish</i> request queue of all <i>Publish</i> requests sent by the previous <i>Client</i> and return negative responses for each.
ClientValidated()	A boolean function that returns TRUE only when the <i>Client</i> that is submitting a TransferSubscriptions request is operating on behalf of the same user and supports the same Profiles as the <i>Client</i> of the previous Session.
CreateNotificationMsg()	Increment the SeqNum and create a <i>NotificationMessage</i> from the <i>MonitoredItems</i> assigned to the <i>Subscription</i> . Save the newly-created <i>NotificationMessage</i> in the retransmission queue. If all available <i>Notifications</i> can be sent in the <i>Publish</i> response, the MoreNotifications state Variable is set to FALSE. Otherwise, it is set to TRUE.
CreateSubscription()	Attempt to create the <i>Subscription</i> .
DeleteAckedNotificationMsgs()	Delete the <i>NotificationMessages</i> from the retransmission queue that were acknowledged by the request.
DeleteClientPubReqQueue()	Clear the <i>Publish</i> request queue for the <i>Client</i> that is sending the DeleteSubscriptions request, if there are no more <i>Subscriptions</i> assigned to that <i>Client</i> .
DeleteMonitoredItems()	Delete all <i>MonitoredItems</i> assigned to the <i>Subscription</i>
DequeuePublishReq()	Dequeue a publishing request in first-in first-out order.
EnqueuePublishingReq()	Enqueue the publishing request
InitializeSubscription()	ResetLifetimeCounter() MoreNotifications = FALSE PublishRateChange = FALSE PublishingEnabled = value of publishingEnabled parameter in the CreateSubscription request PublishingReqQueued = FALSE SeqNum = 0 SetSession() StartPublishingTimer()
ResetKeepAliveCounter()	Reset the keep-alive counter to the maximum keep-alive count of the <i>Subscription</i> . The maximum keep-alive count is set by the <i>Client</i> when the <i>Subscription</i> is created and may be modified using the ModifySubscription Service.
ResetLifetimeCounter()	Reset the LifetimeCounter Variable to the value specified for the lifetime of a <i>Subscription</i> in the CreateSubscription Service (Clause 5.13.2).
ReturnKeepAlive()	CreateKeepAliveMsg() ReturnResponse()
ReturnNegativeResponse ()	Return a Service response indicating the appropriate Service level error. No parameters are returned other than the responseHeader that contains the Service level StatusCode.
ReturnNotifications()	CreateNotificationMsg() ReturnResponse() If (MoreNotifications == TRUE) && (PublishingReqQueued == TRUE) { DequeuePublishReq() Loop through this function again }
ReturnResponse()	Return the appropriate response, setting the appropriate parameter values and StatusCodes defined for the Service.
SessionChanged()	A boolean function that returns TRUE only when the Session used to send a TransferSubscriptions request is different than the Client Session currently associated with the Subscription.
SetPublishingEnabled ()	Set the PublishingEnabled state Variable to the value of the publishingEnabled parameter received in the request.
SetSession	Set the Session information for the Subscription to match the Session on which the TransferSubscriptions request was issued.
StartPublishingTimer()	Start or restart the publishing timer and decrement the LifetimeCounter Variable.
UpdateSubscriptionParams()	Negotiate and update the <i>Subscription</i> parameters. If the new keep-alive interval is less than the current value of the keep-alive counter, perform ResetKeepAliveCounter() and ResetLifetimeCounter().

### 5.13.2 CreateSubscription

#### 5.13.2.1 Description

This Service is used to create a *Subscription*. *Subscriptions* monitor a set of *MonitoredItems* for *Notifications* and return them to the *Client* in response to *Publish* requests.

#### 5.13.2.2 Parameters

Table 107 defines the parameters for the Service.

**Table 107 – CreateSubscription Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	Request Header	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
requestedPublishingInterval	Duration	This interval defines the cyclic rate that the <i>Subscription</i> is being requested to return <i>Notifications</i> to the <i>Client</i> (see Clause 7.6 for <i>Duration</i> definition). This interval is expressed in milliseconds. This interval is represented by the publishing timer in the <i>Subscription</i> state table (see Clause 5.13.1.2). The negotiated value for this parameter returned in the response is used as the default sample interval for <i>MonitoredItems</i> assigned to this <i>Subscription</i> .
requestedLifetimeCount	Counter	Requested lifetime count (see Clause 7.3 for <i>Counter</i> definition). The lifetime count must be a minimum of three times the keep alive count. When the publishing timer has expired this number of times without a <i>NotificationMessage</i> being sent, the <i>Subscription</i> will be deleted by the <i>Server</i> .
requestedMaxKeepAliveCount	Counter	Requested maximum keep alive count (see Clause 7.3 for <i>Counter</i> definition). When the publishing timer has expired this number of times without a <i>NotificationMessage</i> being sent, the <i>Subscription</i> sends a keep alive <i>Message</i> to the <i>Client</i> .
publishingEnabled	Boolean	A Boolean parameter with the following values : TRUE publishing is enabled for the <i>Subscription</i> . FALSE publishing is disabled for the <i>Subscription</i> . The value of this parameter does not affect the value of the monitoring mode <i>Attribute</i> of <i>MonitoredItems</i> .
priority	Byte	Indicates the relative priority of the <i>Subscription</i> . When more than one <i>Subscription</i> needs to send <i>Notifications</i> , the <i>Server</i> should dequeue a Publish request to the <i>Subscription</i> with the highest priority number. For <i>Subscriptions</i> with equal priority the <i>Server</i> should dequeue Publish requests in a round-robin fashion. Any <i>Subscription</i> that needs to send a keep alive <i>Message</i> must take precedence regardless of its priority, in order to prevent the <i>Subscription</i> from expiring. A Client that does not require special priority settings should set this value to zero.
<b>Response</b>		
responseHeader	Response Header	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
subscriptionId	IntegerId	The <i>Server</i> -assigned identifier for the <i>Subscription</i> (see Clause 7.9 for <i>IntegerId</i> definition). This identifier must be unique for the entire <i>Server</i> , not just for the <i>Session</i> , in order to allow the <i>Subscription</i> to be transferred to another <i>Session</i> using the <i>TransferSubscriptions</i> service.
revisedPublishingInterval	Duration	The actual publishing interval that the <i>Server</i> will use, expressed in milliseconds (see Clause 7.6 for <i>Duration</i> definition). The <i>Server</i> should attempt to honor the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.
revisedLifetimeCount	Counter	The lifetime of the <i>Subscription</i> must be a minimum of three times the keep alive interval negotiated by the <i>Server</i> .
revisedMaxKeepAliveCount	Counter	The actual maximum keep alive count (see Clause 7.3 for <i>Counter</i> definition). The <i>Server</i> should attempt to honor the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.

#### 5.13.2.3 Service results

Table 108 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 108 – CreateSubscription Service Result Codes**

Symbolic Id	Description
Bad_TooManySubscriptions	The Server has reached its maximum number of subscriptions.

### 5.13.3 ModifySubscription

#### 5.13.3.1 Description

This Service is used to modify a *Subscription*.

#### 5.13.3.2 Parameters

Table 109 defines the parameters for the Service. Changes to the publishing interval become effective the next time the publishing timer expires.

**Table 109 – ModifySubscription Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> (see Clause 7.9 for <i>IntegerId</i> definition).
requestedPublishingInterval	Duration	<p>This interval defines the cyclic rate that the <i>Subscription</i> is being requested to return <i>Notifications</i> to the <i>Client</i> (see Clause 7.6 for <i>Duration</i> definition). This interval is expressed in milliseconds. This interval is represented by the publishing timer in the <i>Subscription</i> state table (see Clause 5.13.1.2).</p> <p>The negotiated value for this parameter returned in the response is used as the default sample interval for <i>MonitoredItems</i> assigned to this <i>Subscription</i>.</p>
requestedLifetimeCount	Counter	<p>Requested lifetime count (see Clause 7.3 for <i>Counter</i> definition). The lifetime count must be a minimum of three times the keep-alive count.</p> <p>When the publishing timer has expired this number of times without a <i>NotificationMessage</i> being sent, the <i>Subscription</i> will be deleted by the Server.</p>
requestedMaxKeepAliveCount	Counter	Requested maximum keep-alive count (see Clause 7.3 for <i>Counter</i> definition). When the publishing timer has expired this number of times without a <i>NotificationMessage</i> being sent, the <i>Subscription</i> sends a keep-alive Message to the <i>Client</i> .
priority	Byte	<p>Indicates the relative priority of the <i>Subscription</i>. When more than one <i>Subscription</i> needs to send <i>Notifications</i>, the Server should dequeue a Publish request to the <i>Subscription</i> with the highest <i>priority</i> number. For <i>Subscriptions</i> with equal <i>priority</i> the Server should dequeue Publish requests in a round-robin fashion. Any <i>Subscription</i> that needs to send a keep-alive Message must take precedence regardless of its <i>priority</i>, in order to prevent the <i>Subscription</i> from expiring.</p> <p>A Client that does not require special priority settings should set this value to zero.</p>
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
revisedPublishingInterval	Duration	The actual publishing interval that the Server will use, expressed in milliseconds (see Clause 7.6 for <i>Duration</i> definition). The Server should attempt to honor the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.
revisedLifetimeCount	Counter	The lifetime of the <i>Subscription</i> must be a minimum of three times the keep-alive interval negotiated by the Server.
revisedMaxKeepAliveCount	Counter	The actual maximum keep-alive count (see Clause 7.3 for <i>Counter</i> definition). The Server should attempt to honor the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.

### 5.13.3.3 Service results

Table 110 defines the Service results specific to this Service. Common StatusCodes are defined in Table 156.

**Table 110 – ModifySubscription Service Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.13.4 SetPublishingMode

#### 5.13.4.1 Description

This Service is used to enable sending of *Notifications* on one or more *Subscriptions*.

#### 5.13.4.2 Parameters

Table 111 defines the parameters for the Service.

**Table 111 – SetPublishingMode Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
publishingEnabled	Boolean	A Boolean parameter with the following values : TRUE publishing of <i>NotificationMessages</i> is enabled for the <i>Subscription</i> . FALSE publishing of <i>NotificationMessages</i> is disabled for the <i>Subscription</i> . The value of this parameter does not affect the value of the monitoring mode <i>Attribute</i> of <i>MonitoredItems</i> . Setting this value to FALSE does not discontinue the sending of keep-alive <i>Messages</i> .
subscriptionIds []	IntegerId	List of Server-assigned identifiers for the <i>Subscriptions</i> to enable or disable (see Clause 7.9 for <i>IntegerId</i> definition).
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>Subscriptions</i> to enable/disable (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Subscriptions</i> to enable/disable (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.13.4.3 Service results

Table 112 defines the Service results specific to this Service. Common StatusCodes are defined in Table 156.

**Table 112 – SetPublishingMode Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

#### 5.13.4.4 StatusCodes

Table 113 defines values for the *results* parameter that are specific to this Service. Common StatusCodes are defined in Table 157.

**Table 113 – SetPublishingMode Operation Level Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.13.5 Publish

#### 5.13.5.1 Description

This Service is used for two purposes. First, it is used to acknowledge the receipt of *NotificationMessages* for one or more *Subscriptions*. Second, it is used to request the Server to return a *NotificationMessage* or a keep-alive Message. Since *Publish* requests are not directed to a specific *Subscription*, they may be used by any *Subscription*. Clause 5.13.1.2 describes the use of the *Publish Service*.

*Client* strategies for issuing *Publish* requests may vary depending on the networking delays between the *Client* and the *Server*. In many cases, the *Client* may wish to issue a *Publish* request immediately after creating a *Subscription*, and thereafter, immediately after receiving a *Publish* response.

In other cases, especially in high latency networks, the *Client* may wish to pipeline *Publish* requests to ensure cyclic reporting from the *Server*. Pipelining involves sending more than one *Publish* request for each *Subscription* before receiving a response. For example, if the network introduces a delay between the *Client* and the *Server* of 5 seconds and the publishing interval for a *Subscription* is one second, then the *Client* will have to issue *Publish* requests every second instead of waiting for a response to be received before sending the next request.

A server should limit the number of active *Publish* requests to avoid an infinite number since it is expected that the *Publish* requests are queued in the *Server*. But a *Server* must accept more queued *Publish* requests than created *Subscriptions*. It is expected that a *Server* supports several *Publish* requests per *Subscription*. The *Server* must return the *Service* result *Bad\_TooManyPublishRequests* if the number of *Publish* requests exceeds the limit. If a *Client* receives this *Service* result for a *Publish* request it must not issue another *Publish* request before one of its outstanding *Publish* requests is returned from the *Server*.

### 5.13.5.2 Parameters

Table 114 defines the parameters for the Service.

**Table 114 – Publish Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscription Acknowledgements []	Subscription Acknowledgement	The list of acknowledgements for one or more <i>Subscriptions</i> . This list may contain multiple acknowledgements for the same <i>Subscription</i> (multiple entries with the same <i>subscriptionId</i> ).
subscriptionId	IntegerId	The Server assigned identifier for a <i>Subscription</i> (see Clause 7.9 for <i>IntegerId</i> definition).
sequenceNumber	Counter	The sequence number being acknowledged (see Clause 7.3 for <i>Counter</i> definition). The Server may delete the <i>Message</i> with this sequence number from its retransmission queue.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
subscriptionId	IntegerId	The Server-assigned identifier for the <i>Subscription</i> for which <i>Notifications</i> are being returned (see Clause 7.9 for <i>IntegerId</i> definition). The value 0 is used to indicate that there were no <i>Subscriptions</i> defined for which a response could be sent.
availableSequence Numbers []	Counter	A list of sequence number ranges that identify unacknowledged <i>NotificationMessages</i> that are available for retransmission from the <i>Subscription</i> 's retransmission queue. This list is prepared after processing the acknowledgements in the request (see Clause 7.3 for <i>Counter</i> definition).
moreNotifications	Boolean	A Boolean parameter with the following values : TRUE            the number of <i>Notifications</i> that were ready to be sent could not be sent in a single response. FALSE            all <i>Notifications</i> that were ready are included in the response.
notificationMessage	Notification Message	The <i>NotificationMessage</i> that contains the list of <i>Notifications</i> . The <i>NotificationMessage</i> parameter type is specified in Clause 7.13.
results []	StatusCodes	List of results for the acknowledgements (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>subscriptionAcknowledgements</i> request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the acknowledgements (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>subscriptionAcknowledgements</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

### 5.13.5.3 Service results

Table 115 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 115 – Publish Service Result Codes**

Symbolic Id	Description
Bad_TooManyPublishRequests	The server has reached the maximum number of queued publish requests.
Bad_NoSubscription	There is no subscription available for this session.

### 5.13.5.4 StatusCodes

Table 116 defines values for the *acknowledgeResults* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 116 – Publish Operation Level Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.13.6 Republish

#### 5.13.6.1 Description

This Service requests the *Subscription* to republish a *NotificationMessage* from its retransmission queue. If the Server does not have the requested *Message* in its retransmission queue, it returns an error response.

See Clause 5.13.1.2 for the detail description of the behaviour of this Service.

#### 5.13.6.2 Parameters

Table 117 defines the parameters for the Service.

**Table 117 – Republish Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionId	IntegerId	The Server assigned identifier for the <i>Subscription</i> to be republished (see Clause 7.9 for <i>IntegerId</i> definition).
retransmitSequence Number	Counter	The sequence number of a specific <i>NotificationMessage</i> to be republished (see Clause 7.3 for <i>Counter</i> definition).
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
notificationMessage	Notification Message	The requested <i>NotificationMessage</i> . The <i>NotificationMessage</i> parameter type is specified in Clause 7.13.

#### 5.13.6.3 Service results

Table 118 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 118 – Republish Service Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.
Bad_MessageNotAvailable	The requested message is no longer available.

### 5.13.7 TransferSubscriptions

#### 5.13.7.1 Description

This Service is used to transfer a *Subscription* and its *MonitoredItems* from one *Session* to another. For example, a *Client* may need to reopen a *Session* and then transfer its *Subscriptions* to that *Session*. It may also be used by one *Client* to take over a *Subscription* from another *Client* by transferring the *Subscription* to its *Session*.

The *sessionId* contained in the request header identifies the *Session* to which the *Subscription* and *MonitoredItems* will be transferred. The *Server* must validate that the *Client* of that *Session* is operating on behalf of the same user and that the potentially new *Client* supports the *Profiles* that are necessary for the *Subscription*. If the *Server* transfers the *Subscription*, it returns the sequence numbers of the *NotificationMessages* that are available for retransmission. The *Client* should acknowledge all *Messages* in this list for which it will not request retransmission.

If the *Server* transfers the *Subscription* to the new *Session*, the *Server* must return the service result *Good\_SubscriptionTransferred* with the *subscriptionId* and an empty *notificationMessage* for the next *Publish* request in the context of the old *Session*.

#### 5.13.7.2 Parameters

Table 119 defines the parameters for the Service.

**Table 119 – TransferSubscriptions Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionIds []	IntegerId	List of identifiers for the <i>Subscriptions</i> to be transferred to the new <i>Client</i> (see Clause 7.9 for <i>IntegerId</i> definition). These identifiers are transferred from the primary <i>Client</i> to a backup <i>Client</i> via external mechanisms.
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	TransferResult	List of results for the <i>Subscriptions</i> to transfer. The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter.
statusCode	StatusCodes	<i>StatusCodes</i> for each <i>Subscription</i> to be transferred (see Clause 7.28 for <i>StatusCodes</i> definition).
availableSequenceNumbersRanges []	NumericRange	A list of sequence number ranges that identify <i>NotificationMessages</i> that are in the <i>Subscription</i> 's retransmission queue. This parameter is null if the transfer of the <i>Subscription</i> failed. The <i>NumericRange</i> type is defined in Clause 7.14.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Subscriptions</i> to transfer (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.13.7.3 Service results

Table 120 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 120 – TransferSubscriptions Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.
Bad_UserAccessDenied	See Table 156 for the description of this result code. The <i>Client</i> of the current <i>Session</i> is not operating on behalf of the same user as the <i>Session</i> that owns the <i>Subscription</i> .
Bad_InsufficientClientProfile	The <i>Client</i> of the current <i>Session</i> does not support one or more <i>Profiles</i> that are necessary for the <i>Subscription</i> .

### 5.13.7.4 StatusCodes

Table 121 defines values for the operation level *statusCode* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 121 – TransferSubscriptions Operation Level Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

### 5.13.8 DeleteSubscriptions

#### 5.13.8.1 Description

This Service is invoked by the *Client* to delete one or more *Subscriptions* that it has created and that have not been transferred to another *Client* or that have been transferred to it.

Successful completion of this Service causes all *MonitoredItems* that use the *Subscription* to be deleted. If this is the last *Subscription* assigned to the *Client* issuing the request, then all *Publish* requests queued by that *Client* are dequeued and a negative response is returned for each.

#### 5.13.8.2 Parameters

Table 122 defines the parameters for the Service.

**Table 122 – DeleteSubscriptions Service Parameters**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	Common request parameters (see Clause 7.19 for <i>RequestHeader</i> definition).
subscriptionIds []	IntegerId	The Server-assigned identifier for the <i>Subscription</i> (see Clause 7.9 for <i>IntegerId</i> definition).
<b>Response</b>		
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).
results []	StatusCodes	List of <i>StatusCodes</i> for the <i>Subscriptions</i> to delete (see Clause 7.28 for <i>StatusCodes</i> definition). The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Subscriptions</i> to delete (see Clause 7.5 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>subscriptionIds</i> request parameter. This list is empty if diagnostics information was not requested in the request header.

#### 5.13.8.3 Service results

Table 123 defines the Service results specific to this Service. Common *StatusCodes* are defined in Table 156.

**Table 123 – DeleteSubscriptions Service Result Codes**

Symbolic Id	Description
Bad_NothingToDo	See Table 156 for the description of this result code.

#### 5.13.8.4 StatusCodes

Table 124 defines values for the *results* parameter that are specific to this Service. Common *StatusCodes* are defined in Table 157.

**Table 124 – DeleteSubscriptions Operation Level Result Codes**

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 156 for the description of this result code.

## 6 Service behaviours

### 6.1 Security

#### 6.1.1 Overview

The UA services define a number of mechanisms to meet the security requirements outlined in [UA Part 2]. This section describes a number of important security-related procedures that *UA Applications* must follow.

#### 6.1.2 Obtaining and Installing an Application Instance Certificate

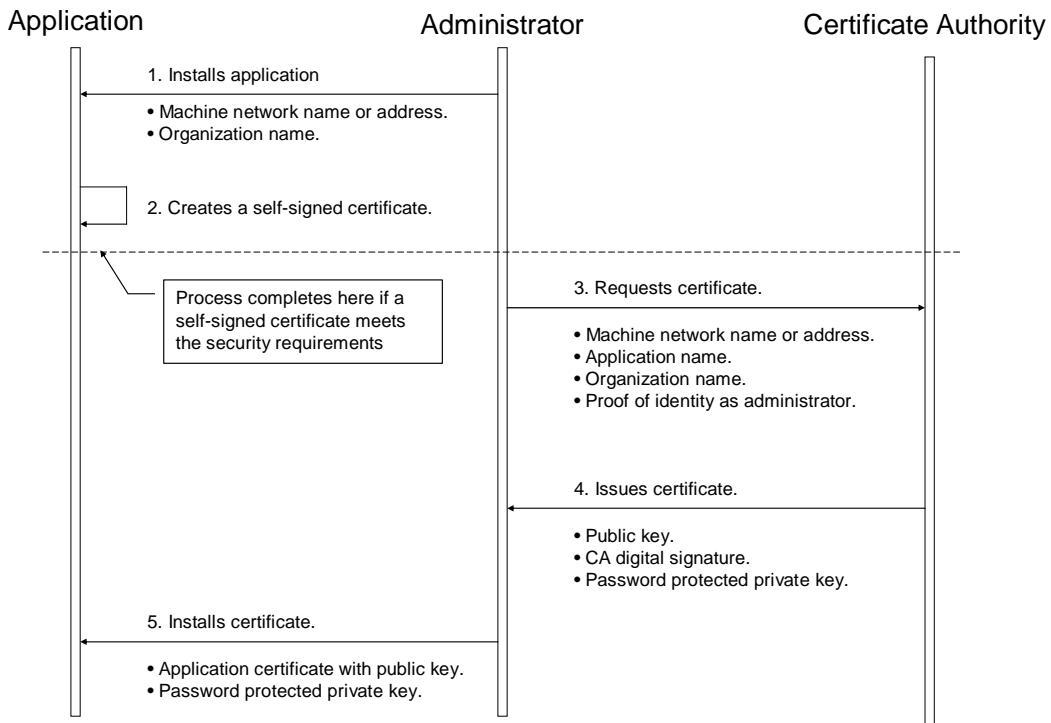
All *UA Applications* require an application instance certificate which should contain the following information:

- The network name or address of the computer where the application runs;
- The name of the organisation that administers or owns the application;
- The name of the application;
- The name of the certificate authority that issued the certificate;
- The issue and expiry date for the certificate;
- The public key issued to the application by the certificate authority (CA);
- A digital signature created by the certificate authority (CA).

In addition, each application instance certificate has a private key which should be stored in a location that can only be accessed by the application. If this private key is compromised, the administrator must assign a new application instance certificate and private key to the application.

This certificate may be generated automatically when the application is installed. In this situation the private key assigned to the certificate must be used to create the certificate signature. Certificates created in this way are called self-signed certificates.

If the administrator responsible for the application decides that a self-signed certificate does not meet the security requirements of the organisation, then the administrator should install a certificate issued by a certification authority. The steps involved in requesting an application instance certificate from a certificate authority are shown in Figure 28.



**Figure 28 – Obtaining and Installing an Application Instance Certificate**

The figure above illustrates the interactions between the *Application*, the *Administrator* and the *CertificateAuthority*. The *Application* is a *UA Application* installed on a single machine. The *Administrator* is the person responsible for managing the machine and the *UA Application*. The *CertificateAuthority* is an entity that can issue digital certificates that meet the requirements of the organisation deploying the *UA Application*.

If the *Administrator* decides that a self-signed certificate meets the security requirements for the organisation, then the *Administrator* may skip Steps 3 through 5. Application vendors must always create a default self-signed certificate during the installation process. Every *UA Application* must allow the *Administrators* to replace application instance certificates with certificates that meet their requirements.

When the *Administrator* requests a new certificate from a certificate authority, the certificate authority may require that the *Administrator* provide proof of authorization to request certificates for the organisation that will own the certificate. The exact mechanism used to provide this proof depends on the certificate authority.

Vendors may choose to automate the process of acquiring certificates from an authority. If this is the case, the *Administrator* would still go through the steps illustrated in the figure, however, the application install program would do them automatically and only prompt the *Administrator* to provide information about the application instance being installed.

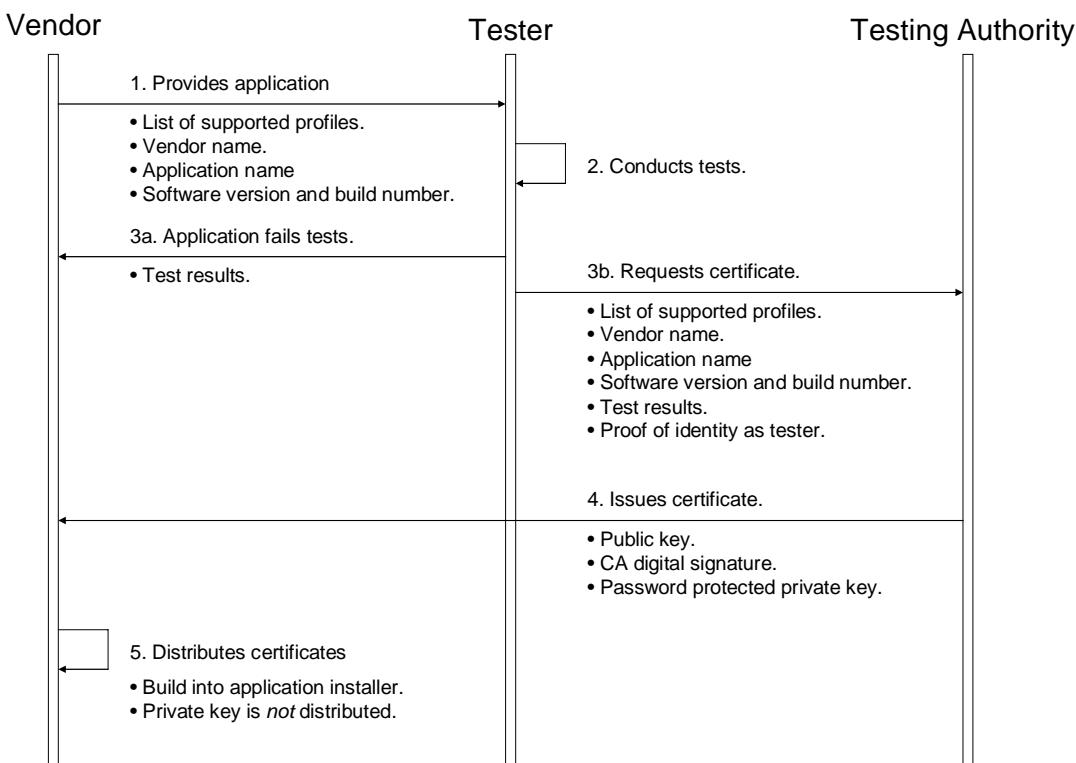
### 6.1.3 Obtaining and Installing an Software Certificate

All *UA Applications* may have one or more software certificates that are issued by testing authorities and that describe the profiles the application supports and the certification tests the application has passed. These software certificates contain the following information:

- The name of the organisation that developed the application;
- The name of the application;
- The software version and build number;
- A list of profiles supported by the application;
- The certification testing status for each supported profile;
- The name of the testing authority that issued the certificate;
- The issue and expiry date for the certificate;
- The public key issued to the application by the testing authority;
- A digital signature created by the testing authority.

The application vendor is responsible for completing the testing process and requesting software certificates from the testing authorities. The vendor should install the software certificates when an application is installed on machine. When distributing these certificates, the application vendors should take precautions to prevent unauthorized users from acquiring their software certificates and using them for applications that the vendor did not develop. Misused software certificates are not a security risk, but vendors could find that they are blamed for interoperability problems caused by use by unauthorized applications.

The steps involved in acquiring and installing a software certificate from a certificate authority are shown in Figure 29.



**Figure 29 – Obtaining and Installing a Software Certificate**

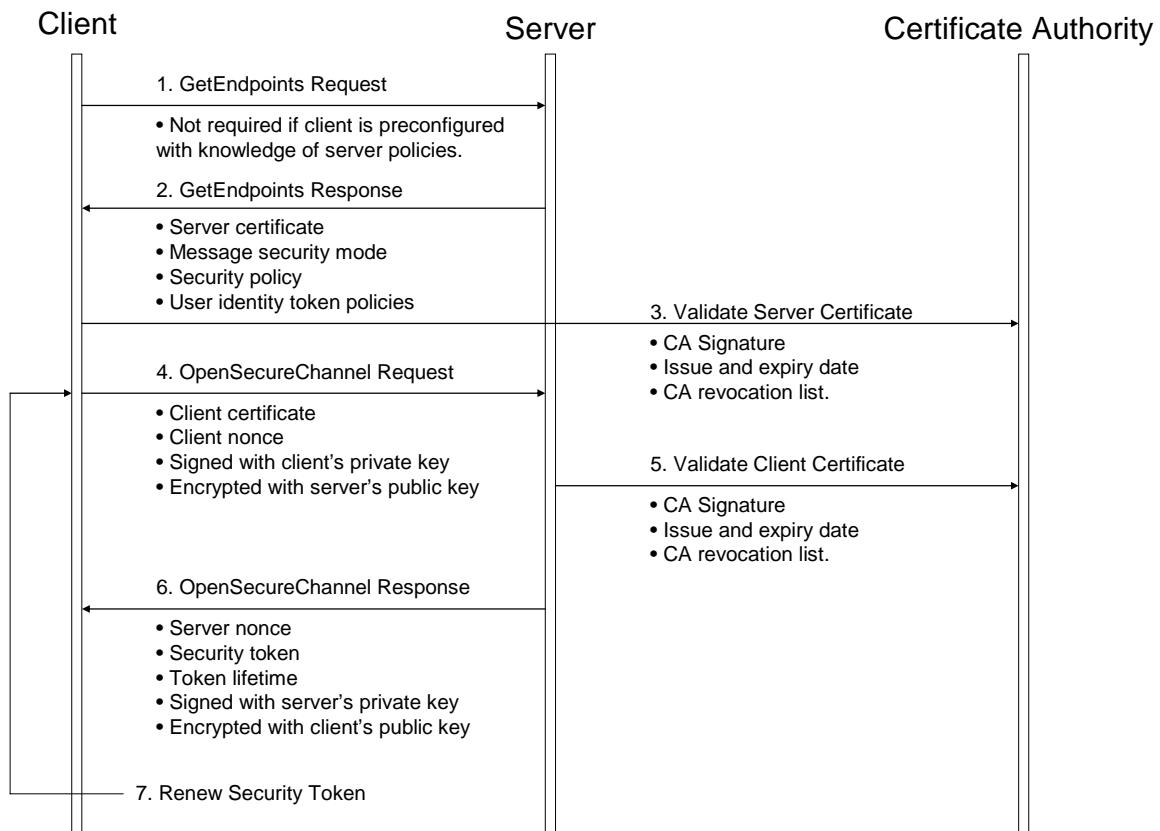
The figure above illustrates the interactions between the *Vendor*, the *Tester* and the *TestingAuthority*. The *Vendor* is the organisation that developed the *UA Application*. The *Tester* may be the vendor (for self-certification) or it may be a third-party testing facility. The *TestingAuthority* is a certificate authority managed by the organisation that created the *UA Application* profiles and certification programmes.

The *TestingAuthority* will issue certificates only to people it trusts. For that reason, the *Tester* must provide proof of identity before the *TestingAuthority* will issue a certificate.

#### 6.1.4 Creating a SecureChannel

All *UA Applications* must establish a *SecureChannel* before creating a *Session*. This *SecureChannel* requires that both applications have access to certificates that can be used to encrypt and sign *Messages* exchange. The application instance certificates installed by following the process described in Clause 6.1.2 may be used for this purpose.

The steps involved in establishing a *SecureChannel* are shown in Figure 30.



**Figure 30 – Establishing a SecureChannel**

The figure above assumes *Client* and *Server* have online access to a certificate authority (CA). If online access is not available and if the administrator has installed the CA public key on the local machine, then the *Client* and *Server* must still validate the application certificates using that key. The figure shows only one CA, however, there is no requirement that the *Client* and *Server* certificates be issued by the same authority. A self-signed application instance certificate does not need to be verified with a CA.

Both the *Client* and *Server* will have a list of certificates that they have been configured to trust (sometimes called the Certificate Trust List or CTL). These trusted certificates may be certificates for certificate authorities or they may be *UA Application* instance certificates. *UA Applications* may be configured to reject connections with applications that do not have a trusted certificate.

Certificates can be compromised, which means they should no longer be trusted. Administrators can revoke a certificate by removing it from the trust list for all applications or they can add the certificate to the Certificate Revocation List (CRL) for the certificate authority that issued the certificate. Administrators may save a local copy of the CRL for each certificate authority when online access is not available.

A *Client* does not need to call *GetEndpoints* each time it connects to the *Server*. This information should change rarely and the *Client* can cache it locally. If the *Server* rejects the *OpenSecureChannel* request the *Client* should call *GetEndpoints* and make sure the *Server* configuration has not changed.

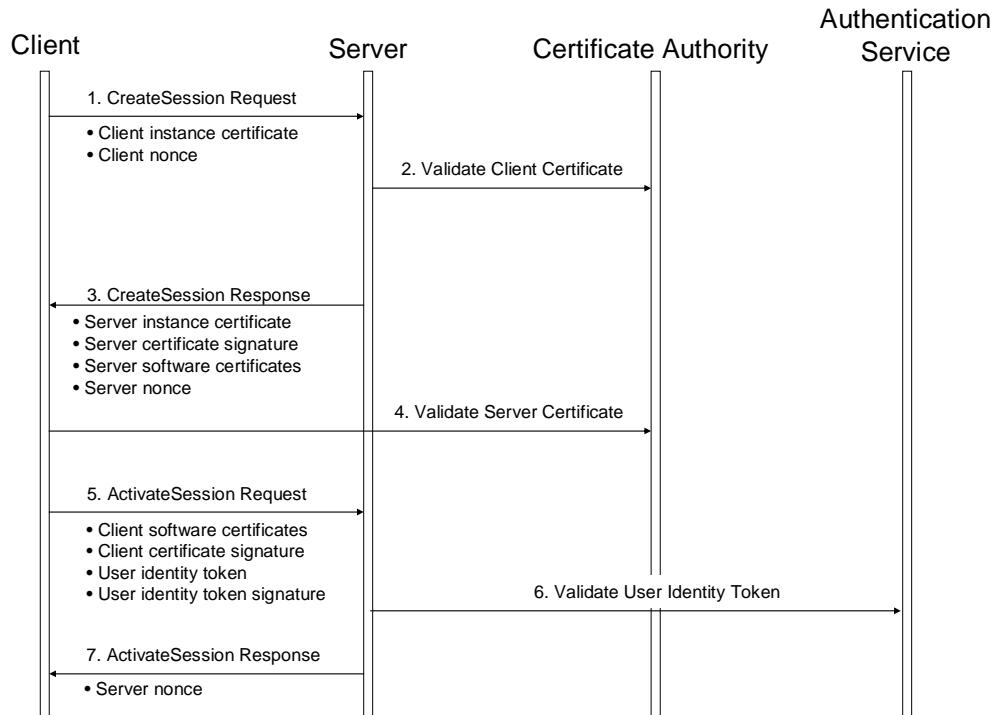
The exact mechanisms for using the security token to sign and encrypt *Messages* exchanged over the *SecureChannel* are described in [UA Part 6]. The process for renewing tokens is also described in detail in [UA Part 6].

In many cases, the certificates used to establish the *SecureChannel* will be the application instance certificates. However, some communication stacks might not support certificates that are specific to a single application. Instead, they expect all communication to be secured with a certificate specific to a user or the entire machine. For this reason, *UA Applications* will need to exchange their application instance certificates when creating a *Session*.

### 6.1.5 Creating a Session

Once a UA *Client* has established a *SecureChannel* with a *Server* it can create a UA *Session*.

The steps involved in establishing a *Session* are shown in Figure 31.



**Figure 31 – Establishing a Session**

The figure above illustrates the interactions between a *Client*, a *Server*, a certificate authority (CA) and an authentication service. The CA is responsible for issuing the application certificates. If the *Client* or *Server* do not have online access to the CA, then they must validate the application certificates using the CA public key that the administrator must install on the local machine.

The authentication service is a central database that can verify that user token provided by the *Client*. This authentication service may also tell the *Server* what access rights the user has. The authentication service depends on the user identity token. It could be a certificate authority, a Kerberos ticket granting service, a WS-Trust Server or a proprietary database of some sort.

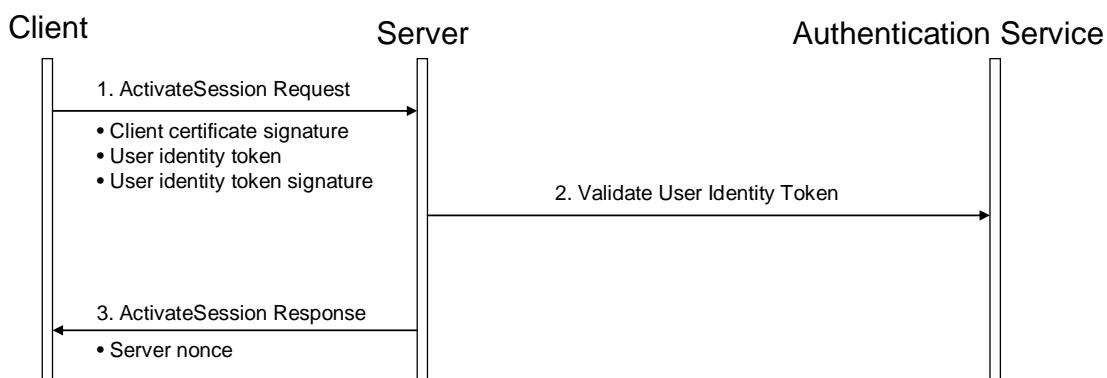
The *Client* and *Server* must prove possession of their application certificates by signing the certificates with a nonce appended. The exact mechanism used to create the proof of possession signatures is described in [UA Part 6]. Similarly, the *Client* must prove possession of some types of user identity tokens by creating signatures with the secret associated with the token.

The application instance certificates exchanged while creating a *Session* may be the same certificates that are exchanged when the *SecureChannel* was opened. If this is the case, the applications do not need to re-validate the certificate. UA requires that the application instance certificates be provided twice because the UA application might not be able to discover what certificates were used to establish the *SecureChannel* and the UA application may need these certificates to apply access restrictions that are based on an application instance rather than a specific user.

### 6.1.6 Impersonating a User

Once a UA *Client* has established a *Session* with a *Server* it can change the user identity associated with the *Session* by calling the *ActivateSession* service.

The steps involved in impersonating a user are shown in Figure 32.



**Figure 32 – Impersonating a User**

## 6.2 UA Auditing

### 6.2.1 Overview

Auditing is a requirement in many systems. It provides a means of tracking activities that occur as part of normal operation of the system. It also provides a means of tracking abnormal behaviour. It is also a requirement from a security standpoint. For more information on the security aspects of auditing see [UA Part 2]. This section describes what is expected of a UA *Server* and *Client* with respect to auditing and it details the audit recommendation for each service set.

### 6.2.2 General audit logs

Each UA service request contains a string parameter that is used to carry an audit record id. A *Client* or any *Server* operating as a *Client*, such as an aggregating *Server*, can create a local audit log entry for a request that it submits. This parameter allows this *Client* to pass the identifier for this entry with the request. If this *Server* also maintains an audit log, it should include this id in its audit log entry that it writes. When this log is examined and that entry is found, the examiner will be able to relate it directly to the audit log entry created by the *Client*. This capability allows for traceability across audit logs within a system.

### 6.2.3 General audit Events

A Server that maintains an audit log must provide the audit log entries via standard *Event Messages*. The *AuditEventType* and its sub-types are defined in [UA Part 3]. An audit *Event Message* also includes the audit record Id. The details of the *AuditEventType* and its sub types are defined in [UA Part 5]. A Server that is an aggregating Server that supports auditing must also subscribe for audit events for all of the Servers that it is aggregating (assuming they provide auditing). The combined stream should be available from the aggregating Server.

### 6.2.4 Auditing for SecureChannel Service Set

All Services in this Service Set for Servers that support auditing must generate audit entries and audit *Events* for failed service invocations and for successful invocation of the *OpenSecureChannel* and *CloseSecureChannel* Services. The audit entries should be setup prior to the actual call, allowing the correct audit record Id to be provided. The *OpenSecureChannel* Service must generate an audit *Event* of type *AuditOpenSecureChannelEventType* or a subtype of it. The *CloseSecureChannel* service must generate an audit *Event* of type *AuditCloseSecureChannelEventType* or a subtype of it. Both of these *Event* types are sub-types of the *AuditChannelEventType*. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the service failed. The additional parameters should include the details of the request. It is understood that these events may be generated by the underlining stack in many cases, but they must be made available to the Server and the Server must report them.

### 6.2.5 Auditing for Session Service Set

All Services in this Service Set for Servers that support auditing must generate audit entries and audit *Events* for both successful and failed Service invocations. These Services must generate an audit *Event* of type *AuditSessionEventType* or a sub-type of it. In particular, they must generate the base *EventType* or the appropriate sub-type, depending on the service that was invoked. The *CreateSession* service must generate *AuditCreateSessionEventType* events or sub-types of it. The *ActivateSession* service must generate *AuditActivateSessionType* events or sub-types of it. When the *ActivateSession* Service is called to change the user identity then the server must generate *AuditImpersonateUserEventType* events or sub-types of it. The *CloseSession* service must generate the base *EventType* of *AuditSessionEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the Service failed. The additional parameters should include the details of the request.

For *Clients*, that support auditing, accessing the services in the *Session Service Set* must generate audit entries for both successful and failed invocations of the Service. These audit entries should be setup prior to the actual Service invocation, allowing the invocation to contain the correct audit record id.

### 6.2.6 Auditing for NodeManagement Service Set

All Services in this Service Set for Servers that support auditing must generate audit entries and audit *Events* for both successful and failed Service invocations. These Services must generate an audit *Event* of type *AuditNodeManagementEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for *Events* of this type should include a description of why the service failed. The additional parameters should include the details of the request.

For *Clients* that support auditing, accessing the Services in the *NodeManagement Service Set* must generate audit entries for both successful and failed invocations of the Service. All audit entries should be setup prior to the actual Service invocation, allowing the invocation to contain the correct audit record id.

### 6.2.7 Auditing for Attribute Service Set

The *Write* or *HistoryUpdate* Services in this Service Set for Servers that support auditing must generate audit entries and audit *Events* for both successful and failed Service invocations. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the Service failed. The additional parameters should include the details of the request.

The *Read* and *HistoryRead* Services may generate audit entries and audit *Events* for failed Service invocations. These Services should generate an audit *Event* of type *AuditEventType*. The *SourceNode* for *Events* of this type should be assigned to the *Nodeid* of the *Node* that reports the error. The *SourceName* for *Events* of this type should be “Attribute/” and the service that generates the event (*Read*, *HistoryRead*). The *Message* for *Events* of this type should include a description of why the Service failed.

For *Clients* that support auditing, accessing the *Write* or *HistoryUpdate* services in the *Attribute Service Set* must generate audit entries for both successful and failed invocations of the Service. Invocations of the other Services in this Service Set may generate audit entries. All audit entries should be setup prior to the actual Service invocation, allowing the invocation to contain the correct audit record id.

### 6.2.8 Auditing for Method Service Set

All Services in this Service Set for Servers that support auditing must generate audit entries and audit *Events* for both successful and failed service invocations if the invocation modifies the address space, writes a value or modifies the state of the system (alarm acknowledge, batch sequencing or other system changes). Methods that do not modify the address space, write values or modify the state of the system may generate events. These method calls should generate the most appropriate *EventType* for the call. The *SourceNode* for *Events* of this type should be the *Session* that the call was issued on. The *SourceName* for *Events* of this type should be “Call/” the *Method* name that was invoked.

For *Clients* that support auditing, accessing the *Method Service Set* must generate audit entries for both successful and failed invocations of the Service, if the invocation modifies the address space, writes a value or modifies the state of the system (alarm acknowledge, batch sequencing or other system changes). Invocations of the other *Methods* may generate audit entries. All audit entries should be setup prior to the actual Service invocation, allowing the invocation to contain the correct audit record id.

### 6.2.9 Auditing for View, Query, MonitoredItem and Subscription Service Set

All of the Services in these four Service Sets only provide the *Client* with information, with the exception of the *TransferSubscriptions* Service in the *Subscription Service Set*. In general, these services will not generate audit entries or audit Event Messages. The *TransferSubscriptions* Service must generate an audit *Event* of type *AuditSessionEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for *Events* of this type should include a description of why the service failed.

For *Clients* that support auditing, accessing the *TransferSubscriptions* Service in the *Subscription Service Set* must generate audit entries for both successful and failed invocations of the Service. Invocations of the other Services in this Service Set do not require audit entries. All audit entries should be setup prior to the actual Service invocation, allowing the invocation to contain the correct audit record id.

## 6.3 Redundancy

### 6.3.1 Redundancy overview

Redundancy in OPC UA ensures that both Clients and Server can be redundant. OPC UA does not provide redundancy, it provides the data structures and services by which redundancy may be achieved in a standard manner.

### 6.3.2 Server redundancy overview

Server redundancy comes in two modes, transparent and non-transparent. By definition, in transparent redundancy the failover of Server responsibilities from one Server to another is transparent to the *Client*: the *Client* does care or even know that failover has occurred; the *Client* does not need to do anything at all to keep data flowing. In contrast, non-transparent failover requires some activity on the part of the *Client*.

The two areas where redundancy creates specific needs are in keeping the Server and *Client* information synchronised across Servers, and in controlling the failover of data flow from one Server to another.

#### 6.3.2.1 Transparent redundancy

For transparent redundancy, all OPC UA provides is the data structures to allow the *Client* to identify what Servers are available in the redundant set, what the service level of each Server is and which Server is currently supporting a specified *Session*. All OPC UA interactions within a given session will be supported by one Server and the *Client* is able to identify which Server that is, allowing a complete audit trail for the data. It is the responsibility of the Servers to ensure that information is synchronised between the Servers and to effect the switching of the address from one Server to another upon failover.

Figure 33 shows a typical transparent redundancy setup.

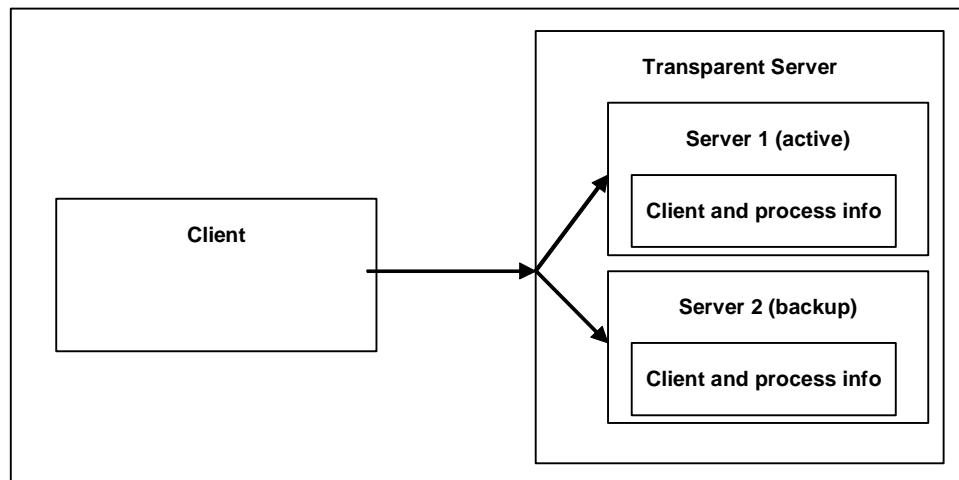
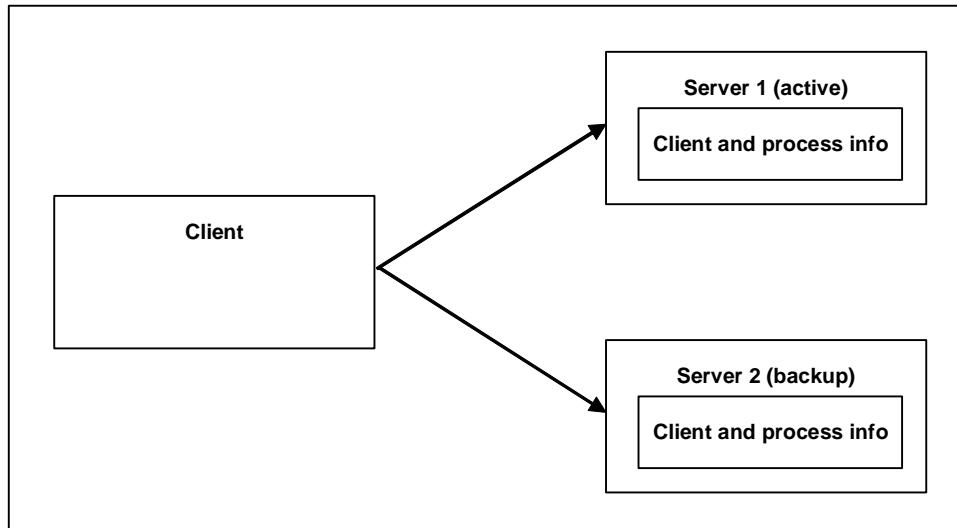


Figure 33 – Transparent Redundancy setup

#### 6.3.2.2 Non-transparent redundancy

For non-transparent redundancy, OPC UA provides the same data structures and also Server information which tells the *Client* what modes of failover the Server supports. This information allows the *Client* to determine what actions it may need to take in order to accomplish failover.

Figure 34 shows a typical non-transparent redundancy setup.

**Figure 34 – Non-Transparent Redundancy setup**

For non-transparent redundancy the *Server* has additional concepts of cold, warm and hot failover. Cold failovers are for *Servers* where only one *Server* can be active at a time. Warm failovers are for *Servers* where the backup *Servers* can be active, but cannot connect to actual data points (typically a system where the underlying devices are limited to a single connection). Hot failovers are for *Servers* where more than one *Server* can be active and fully operational

Table 125 defines the list of failover actions.

**Table 125 – Redundancy failover actions**

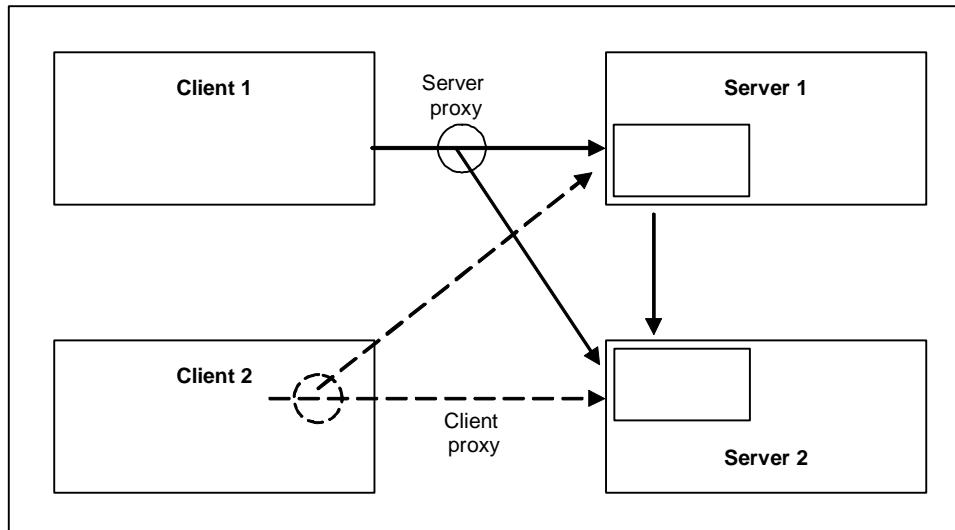
<b>Failover mode</b>	<b>COLD</b>	<b>WARM</b>	<b>HOT</b>
On initial connection:			
Connect to more than one OPC UA Server.		X	X
Creating <i>Subscriptions</i> and adding monitored items to them.		X	X
Activating sampling on the <i>Subscriptions</i>			X
At Failover:			
Connect to backup OPC UA Server	X		
Creating <i>Subscriptions</i> and adding monitored items.	X		
Activating sampling on the <i>Subscriptions</i> .	X	X	
Activate publishing.	X	X	X

Some or all of that activity may be pushed into a *Server proxy* on the *Client* machine, to reduce the amount of functionality that must be designed into the *Client* and to enable simpler *Clients* to take advantage of non-transparent redundancy. By using the *TransferSubscriptions* Service, which allows a *Client* to request that a set of *Subscriptions* be moved from one *Session* to another, a *Server vendor* can effectively make transparent failover a part of a proxy stub that lives on the *Client*. There are two ways to do this, one requiring code in the *Server* to support this and the other doing it all from the *Client proxy* process.

When the *Client proxy* is used, the proxy simply duplicates *Subscriptions* and modifications to *Subscriptions*, by passing the calls on to both *Servers*, but only enabling publishing or sampling on one *Server*. When the proxy detects a failure, it enables publishing and/or sampling on the backup *Server*, just as the *Client* would if it were a redundancy-aware *Client*.

The other method also requires a *Client stub*, but in this case the stub is a much lighter-weight process. In this mode, it is the *Server* which mirrors all *Subscriptions* in the other *Server*, but the *Client endpoint* for these *Subscriptions* is the active *Server*. When the stub detects that the active *Server* has failed, it issues a *TransferSubscriptions* call to the backup *Server*, moving the *Subscriptions* from the *Session* owned by the failed *Server* to its own *Session*, and activating publishing.

Figure 35 shows the difference between *Client proxy* and *Server proxy* redundancy.



**Figure 35 – Redundancy mode**

### 6.3.3 Client redundancy

*Client redundancy* is supported in OPC UA by the *TransferSubscriptions* call and by exposing *Client* information in the *Server* information structures. Since *Subscription* lifetime is not tied to the *Session* in which it was created, backup *Clients* can monitor the active *Client's Session* with the *Server*, just as they would monitor any other data variable. If the active *Client* ceases to be active, the *Server* will send a data update to any *Client* which has that variable monitored. Upon receiving such notification, a backup *Client* would then instruct the *Server* to transfer the *Subscriptions* to its own session. If the *Subscription* is crafted carefully, with sufficient resources to buffer data during the change-over, there need be no data loss from a *Client failover*.

OPC UA does not provide a standardized mechanism for conveying the *SessionId* and *SubscriptionIds* from the active *Client* to the backup *Clients*, but as long as the backup *Clients* know the *Client* name of the active *Client*, this information is readily available using the *SessionDiagnostics* and *SubscriptionDiagnostics* portions of the *ServerDiagnostics* data.

## 7 Common parameter type definitions

### 7.1 BuildInfo

The components of this data type are defined in Table 126.

**Table 126 – BuildInfo Structure**

Name	Type	Description
BuildInfo	structure	Information that describes the build of the software.
applicationUri	String	URI that identifies the software
manufacturerName	String	Name of the software manufacturer.
applicationName	String	Name of the software.
softwareVersion	String	Software version
buildNumber	String	Build number
buildDate	UtcTime	Data and time of the build.

### 7.2 ContentFilter

The *ContentFilter* structure defines a collection of elements that make up a filtering criteria. Each element in the collection describes an operator and an array of operands to be used by the operator. The operators that can be used in a ContentFilter are described in Table 127. The filter is evaluated by evaluating the first entry in the element array starting with the first operand in the operand array. The operands of an element may contain *References* to sub-elements resulting in the evaluation continuing to the referenced elements in the element array.

Table 127 defines the ContentFilter structure.

**Table 127 – ContentFilter Structure**

Name	Type	Description
ContentFilter	structure	
elements []	ContentFilterElement	List of operators and their operands that compose the filter criteria. The filter is evaluated by starting with the first entry in this array.
filterOperator	enum FilterOperator	Filter operator to be evaluated. The <i>FilterOperator</i> enumeration is defined in Table 128.
filterOperands []	Extensible Parameter FilterOperand	Operands used by the selected operator. The number and use depend on the operands defined in Table 128. This array needs at least one entry. This extensible parameter type is the <i>FilterOperand</i> parameter type specified in Clause 8.4. It specifies the list of valid <i>FilterOperand</i> values.

Table 128 defines the valid operators that can be used in a ContentFilter.

**Table 128 –FilterOperator Definition**

Operator	Number of Operands	Description
Equals	2	TRUE if operand[0] is equal to operand[1].
IsNull	1	TRUE if operand[0] is a null value.
GreaterThan	2	TRUE if operand[0] is greater than operand[1].
LessThan	2	TRUE if operand[0] is less than operand[1].
GreaterThanOrEqual	2	TRUE if operand[0] is greater than or equal to operand[1].
LessThanOrEqual	2	TRUE if operand[0] is less than or equal to operand[1].
Like	2	TRUE if operand[0] matches a pattern defined by operand[1]. See Table 129 for the definition of the pattern syntax.
Not	1	TRUE if operand[0] is FALSE.
Between	3	TRUE if operand[0] is greater or equal to operand[1] and less than or equal to operand[2].
InList	2..n	TRUE if operand[0] is equal to one or more of the remaining operands.
And	2	TRUE if operand[0] and operand[1] are TRUE.
Or	2	TRUE if operand[0] or operand[1] are TRUE.
InView	1	TRUE if contained in the View defined by operand[0]. Operand[0] should be of an AttributeOperand type where the attribute is the Nodeld
OfType	1	TRUE if of type operand[0] or of a subtype of operand[0]. Operand[0] should be of an AttributeOperand type where the attribute is the Nodeld
RelatedTo	4	TRUE if the Object is of type Operand[0] and is related to a Nodeld of the type defined in Operand[1] by the Reference type defined in Operand[2]. Operand[0] or Operand[1] can also point to an element Reference where the referred to element is another RelatedTo operator. This allows joining and chaining of relationships. In this case, the referred to element returns a list of Nodelds instead of TRUE or FALSE. Operand[3] defines the number of hops the relationship should be followed. If Operand[3] is 1, then objects must be directly related. If a hop is greater than 1, then a Nodeld of the type described in Operand[1] is checked for at the depth specified by the hop. In this case, the type of the intermediate Node is undefined, and only the Reference type used to reach the end Node is defined. If the requested number of hops cannot be followed, then the result is FALSE, i.e., an empty Node list. If Operand[3] is 0, the relationship is followed to its logical end in a forward direction and each Node is checked to be of the type specified in Operand[1]. If any Node satisfies this criteria, then the result is TRUE, i.e., the Nodeld is included in the sub-list. Operand[0], [1],[2] should be of an AttributeOperand type where the attribute is the Nodeld.

The *Like* operator can be used to perform wildcard comparisons. Several special characters can be included in the second operand of the *Like* operator. The valid characters are defined in Table 129.

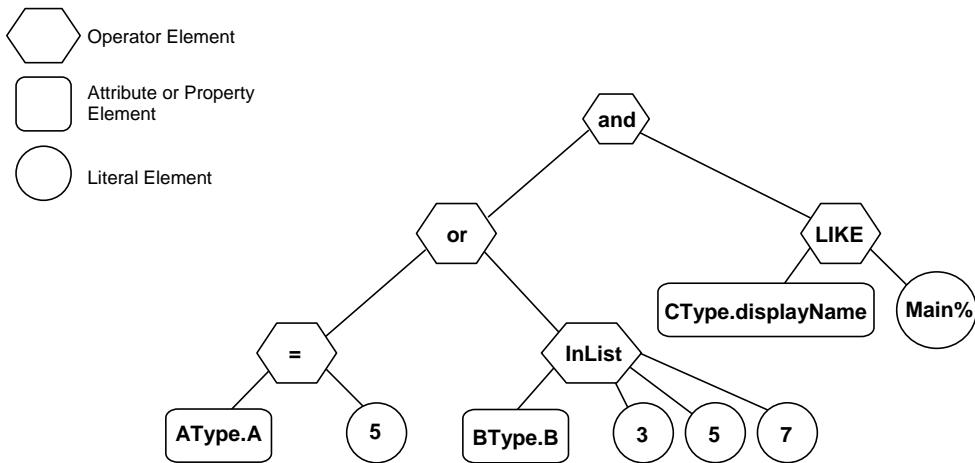
The RelatedTo operator can be used to identify if a given type, set as operand[1], is a subtype of another type set as operand[0] by setting operand[2] to the HasSubtype ReferenceType and operand[3] to 0.

**Table 129 – Wildcard characters**

Special Character	Description
%	Match any character or group of characters
_	Match any single character
\	Escape character allows literal interpretation (i.e. \\ is \, \% is %, \_ is _)
[]	Match any single character in a list (i.e. [1,3-6,8] would match 1,3,4,5,6, and 8)
[!]	Not Matching any single character in a list (i.e. [!1,3-5] would not match 1,3,4, and 5)

For example the logic describe by ‘((AType.A = 5) or InList(BType.B, 3,5,7)) and CType.displayName LIKE “Main%”’ would result in a logic tree as shown in Figure 36 and a ContentFilter as shown in Table 130.

**Ex: (((AType.A = 5) or Inlist(BType.B,3,5,7)) and CType.displayName LIKE “Main%”)**



**Figure 36 – Filter Logic Tree Example**

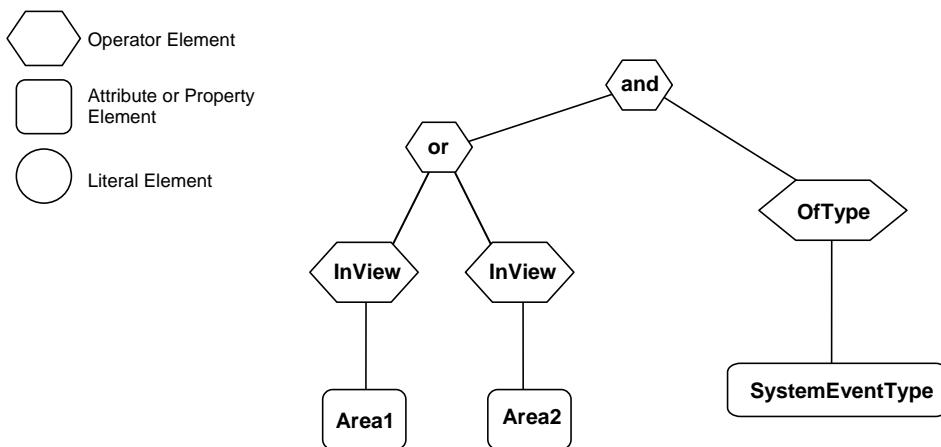
Table 130 describes the elements, operators and operands used in the example.

**Table 130 – ContentFilter Example**

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	And	ElementOperand = 1	Element Operand = 4		
1	Or	ElementOperand = 2	Element Operand = 3		
2	Equals	PropertyOperand = Nodeld: AType, Property: A	LiteralOperand = '5'		
3	InList	PropertyOperand = Nodeld: BType, Property: B	LiteralOperand = '3'	LiteralOperand = '5'	LiteralOperand = '7'
4	Like	AttributeOperand = Nodeld: CType, Attributeld: displayName	LiteralOperand = "Main%"		

As another example a filter to select all *SystemEvents* (including derived types) that are contained in the *Area1 View* or the *Area2 View* would result in a logic tree as shown in Figure 37 and a ContentFilter as shown in Table 131.

**Ex: (InView(Area1) or InView(Area2)) and OfType(SytemEventType)**



**Figure 37 – Filter Logic Tree Example**

Table 131 describes the elements, operators and operands used in the example.

**Table 131 – ContentFilter Example**

Element[0]	Operator	Operand[0]	Operand[1]
0	And	ElementOperand = 1	ElementOperand = 4
1	Or	ElementOperand = 2	ElementOperand = 3
2	InView	AttributeOperand = Nodeld: Area1, Attributeld: Nodeld	
3	InView	AttributeOperand = Nodeld: Area2, Attributeld: Nodeld	
4	OfType	AttributeOperand = Nodeld: SystemEventType, Attributeld: Nodeld"	

Table 132 defines the casting rules for the operand values. The source types in the table are automatically converted to the target types listed in the table. The types used in the table are defined in [UA Part 3]. An overflow during conversion should result in a bad *StatusCodes*.

**Table 132 – Casting rules**

Source Type	Target Type
String	SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Guid, XmlElement
Double	Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean
Float	Double, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean
SByte	String, Double, Float, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean
Int16	String, Double, Float, SByte, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean
Int32	String, Double, Float, SByte, Int16, Int64, Byte, UInt16, UInt32, UInt64, Boolean
Int64	String, Double, Float, SByte, Int16, Int32, Byte, UInt16, UInt32, UInt64, Boolean
Byte	String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean
UInt16	String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt32, UInt64, Boolean
UInt32	String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt64, Boolean
UInt64	String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, Boolean
Boolean	Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64
Guid	String

### 7.3 Counter

This primitive data type is a UInt32 that represents the value of a counter. The initial value of a counter is specified by its use. Modulus arithmetic is used for all calculations, where the modulus is max value + 1. Therefore,

$$x + y = (x + y) \bmod (\text{max value} + 1)$$

For example:

$$\text{max value} + 1 = 0$$

$$\text{max value} + 2 = 1$$

### 7.4 DataValue

The components of this parameter are defined in Table 133.

**Table 133 – DataValue**

Name	Type	Description
DataValue	structure	The value and associated information.
value	BaseDataType	The data value.
statusCode	StatusCodes	The <i>StatusCodes</i> that defines with the Server's ability to access/provide the value. The <i>StatusCodes</i> type is defined in Clause 7.28
sourceTimestamp	UtcTime	The source timestamp for the value.
serverTimestamp	UtcTime	The Server timestamp for the value.

### SourceTimestamp

The *sourceTimestamp* is used to reflect the timestamp that was applied to a *Variable* value by the data source. Once a value has been assigned a source timestamp, the source timestamp for that value instance never changes. In this context, “value instance” refers to the value received, independent of its actual value.

The *sourceTimestamp* must be UTC time and should indicate the time of the last change of the *value* or *statusCode*.

The *sourceTimestamp* should be generated as close as possible to the source of the value but the timestamp needs to be set always by the same physical clock. In the case of redundant sources, the clocks of the sources should be synchronised.

If the OPC UA Server receives the *Variable* value from another OPC UA Server, then the OPC UA Server must always pass the source timestamp without changes. If the source that applies the timestamp is not available, the source time stamp is set to null. For example if a value could not be read because of some error during processing like invalid arguments passed in the request then the *sourceTimestamp* must be null.

In the case of a bad or uncertain status *sourceTimestamp* is used to reflect the time that the source recognized the non-good status or the time the Server last tried to recover from the bad or uncertain status.

The *sourceTimestamp* is only returned with a *Value Attribute*. For all other *Attributes* the returned *sourceTimestamp* is set to null.

### ServerTimestamp

The *serverTimestamp* is used to reflect the time that the Server received a *Variable* value or knew it to be accurate.

In the case of a bad or uncertain status, *serverTimestamp* is used to reflect the time that the Server received the status or that the Server last tried to recover from the bad or uncertain status.

In the case where the OPC UA Server subscribes to a value from another OPC UA Server, each Server applies its own *serverTimestamp*. This is in contrast to the *sourceTimestamp* in which only the originator of the data is allowed to apply the *sourceTimestamp*.

If the Server is subscribing to the value from another Server every ten seconds and the value changes, then the *serverTimestamp* is updated each time a new value is received. If the value does not change, then new values will not be received on the *Subscription*. However, in the absence of errors, the receiving Server applies a new *serverTimestamp* every ten seconds because not receiving a value means that the value has not changed. Thus, the *serverTimestamp* reflects the time at which the Server knew the value to be accurate.

This concept also applies to OPC UA Servers that receive values from exception-based data sources. For example, suppose that a Server is receiving values from an exception-based device, and that

- a) the device is checking values every 0.5 second,
- b) the connection to the device is good,
- c) the device sent an update three minutes ago with a value of 1.234.

In this case, the *Server* value would be 1.234 and the *serverTimestamp* would be updated every 0.5 seconds after the receipt of the value.

#### *StatusCode* assigned to a value

The *StatusCode* is used to indicate the conditions under which a *Variable* value was generated, and thereby can be used as an indicator of the usability of the value. The *StatusCode* is defined in Cause 7.28.

Overall condition (severity):

- A *StatusCode* with severity Good means that the value is of good quality.
- A *StatusCode* with severity Uncertain means that the quality of the value is uncertain for reasons indicated by the Substatus.
- A *StatusCode* with severity Bad means that the value is not usable for reasons indicated by the Substatus.

Rules:

- The *StatusCode* indicates the usability of the value. Therefore, It is required that *Clients* minimally check the *StatusCode Severity* of all results - even if they do not check the other fields - before accessing and using the value.
- A *Server*, which does not support status information, must return a severity code of Good. It is also acceptable for a *Server* to simply return a severity and a non-specific (0) Substatus.
- If the *Server* has no known value - in particular when *Severity* is BAD - it must return a NULL value.

## 7.5 DiagnosticInfo

The components of this parameter are defined in Table 134.

**Table 134 – DiagnosticInfo**

Name	Type	Description
DiagnosticInfo	structure	Vendor-specific diagnostic information.
identifier	structure	The vendor-specific identifier of an error or condition.
namespacelIndex	Index	The symbolic id defined by the <i>symbolicIdIndex</i> parameter is defined within the context of a namespace. This namespace is represented as a string and is conveyed to the <i>Client</i> in the <i>stringTable</i> parameter of the <i>ResponseHeader</i> parameter defined in Clause 7.20. The <i>namespacelIndex</i> parameter contains the index into the <i>stringTable</i> for this string. The index type is defined in Clause 7.8.
symbolicIdIndex	Index	A vendor-specific symbolic identifier string identifies an error or condition. The maximum length of this string is 32 characters. Servers wishing to return a numeric return code should convert the return code into a string and return the string in this identifier. This symbolic identifier string is conveyed to the <i>Client</i> in the <i>stringTable</i> parameter of the <i>ResponseHeader</i> parameter defined in Clause 7.20. The <i>symbolicIdIndex</i> parameter contains the index into the <i>stringTable</i> for this string. The index type is defined in Clause 7.8.
localizedTextIndex	Index	A vendor-specific localized text string describes the symbolic id. The maximum length of this text string is 256 characters. This localized text string is conveyed to the <i>Client</i> in the <i>stringTable</i> parameter of the <i>ResponseHeader</i> parameter defined in Clause 7.20. The <i>localizedTextIndex</i> parameter contains the index into the <i>stringTable</i> for this string. The index type is defined in Clause 7.8.
additionalInfo	String	Vendor-specific diagnostic information.
innerStatusCode	StatusCode	The <i>StatusCode</i> from the inner operation. Many applications will make calls into underlying systems during UA request processing. A UA Server has the option of reporting the status from the underlying system in the diagnostic info.
innerDiagnosticInfo	DiagnosticInfo	The diagnostic info associated with the inner <i>StatusCode</i> .

## 7.6 Duration

This primitive data type is an Int32 that defines an interval of time in milliseconds. Negative values are generally invalid but may have special meanings for the parameter where the Duration is used.

## 7.7 ExpandedNodeId

The components of this parameter are defined in Table 135. *ExpandedNodeIds* allow the namespace to be specified explicitly as a string or with an index in the Server's namespace table.

**Table 135 – ExpandedNodeId**

Name	Type	Description
ExpandedNodeId	structure	The <i>NodeId</i> with the namespace expanded to its string representation.
serverIndex	Index	<p>Index that identifies the Server that contains the <i>TargetNode</i>. This Server may be the local Server or a remote Server.</p> <p>This index is the index of that Server in the local Server's Server table. The index of the local Server in the Server table is always 0. All remote Servers have indexes greater than 0. The Server table is contained in the Server Object in the AddressSpace (see [UA Part 3] and [UA Part 5]).</p> <p>The Client may read the Server table Variable to access the description of the target Server</p>
namespaceUri	String	<p>The URI of the namespace.</p> <p>If this parameter is specified then the namespace index is ignored.</p> <p>[UA Part 6] describes discovery mechanism that can be used to resolve URLs into URLs.</p>
namespaceIndex	Index	<p>The index in the Server's namespace table.</p> <p>This parameter must be 0 and is ignored in the Server if the namespace URI is specified.</p>
identifierType	IdType	Type of the identifier element of the <i>NodeId</i> .
identifier	*	The identifier for a <i>Node</i> in the address space of a UA Server. (see <i>NodeId</i> definition in [UA Part 3]).

## 7.8 Index

This primitive data type is an UInt32 that identifies an element of an array.

## 7.9 IntegerId

This primitive data type is an UInt32 that is used as an identifier, such as a handle. All values, except for 0, are valid.

## 7.10 MessageSecurityMode

The *MessageSecurityMode* is an enumeration that specifies what security should be applied to messages exchanges during a Session. The possible values are described in Table 136.

**Table 136 – MessageSecurityMode Values**

Value	Description
NONE_0	No security is applied.
SIGN_1	All messages are signed but not encrypted.
SIGNANDENCYPT_2	All messages are signed and encrypted.

## 7.11 MonitoringAttributes

The components of this parameter are defined in Table 137.

**Table 137 – MonitoringAttributes**

Name	Type	Description								
MonitoringAttributes	structure	<i>Attributes that define the monitoring characteristics of a MonitoredItem.</i>								
clientHandle	IntegerId	<i>Client-supplied id of the MonitoredItem. This id is used in Notifications generated for the list Node. The IntegerId type is defined in Clause 7.9.</i>								
samplingInterval	Duration	The interval that defines the fastest rate at which the MonitoredItem(s) should be accessed and evaluated. This interval is defined in milliseconds. The value 0 indicates that the Server should use the fastest practical rate. The value -1 indicates that the default sampling interval defined by the publishing rate of the Subscription is used. The Server uses this parameter to assign the MonitoredItems to a sampling interval that it supports. The Duration type is defined in Clause 7.6.								
filter	Extensible Parameter MonitoringFilter	A filter used by the Server to determine if the MonitoredItem should generate a Notification. If not used, this parameter is null. The MonitoringFilter parameter type is an extensible parameter type specified in Clause 8.3. It specifies the types of filters that can be used.								
queueSize	Counter	The requested size of the MonitoredItem queue. If Events are lost an Event of the type EventQueueOverflow is generated. The following values have special meaning: <table> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>the queue has a single entry, effectively disabling queuing.</td> </tr> <tr> <td>&gt;1</td> <td>a first-in-first-out queue is to be used.</td> </tr> <tr> <td>Max Value</td> <td>the max size that the Server can support. This is used for Event Notifications. In this case the Server is responsible for the Event buffer.</td> </tr> </tbody> </table>	Value	Meaning	1	the queue has a single entry, effectively disabling queuing.	>1	a first-in-first-out queue is to be used.	Max Value	the max size that the Server can support. This is used for Event Notifications. In this case the Server is responsible for the Event buffer.
Value	Meaning									
1	the queue has a single entry, effectively disabling queuing.									
>1	a first-in-first-out queue is to be used.									
Max Value	the max size that the Server can support. This is used for Event Notifications. In this case the Server is responsible for the Event buffer.									
discardOldest	Boolean	A boolean parameter that specifies the discard policy when the queue is full and a new Notification is to be enqueued. It has the following values: <table> <tbody> <tr> <td>TRUE</td> <td>the oldest (first) Notification in the queue is discarded. The new Notification is added to the end of the queue.</td> </tr> <tr> <td>FALSE</td> <td>the new Notification is discarded. The queue is unchanged.</td> </tr> </tbody> </table>	TRUE	the oldest (first) Notification in the queue is discarded. The new Notification is added to the end of the queue.	FALSE	the new Notification is discarded. The queue is unchanged.				
TRUE	the oldest (first) Notification in the queue is discarded. The new Notification is added to the end of the queue.									
FALSE	the new Notification is discarded. The queue is unchanged.									

## 7.12 MonitoringMode

The MonitoringMode is an enumeration that specifies whether sampling and reporting are enabled or disabled for a MonitoredItem. The value of the publishing enabled parameter for a Subscription does not affect the value of the monitoring mode for a MonitoredItem of the Subscription. The values of this parameter are defined in Table 138.

**Table 138 – MonitoringMode Values**

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

### 7.13 NotificationMessage

The components of this parameter are defined in Table 139.

**Table 139 – NotificationMessage**

Name	Type	Description
NotificationMessage	structure	The <i>Message</i> that contains one or more <i>Notifications</i> .
sequenceNumber	Counter	The sequence number of the <i>NotificationMessage</i> .
publishTime	UtcTime	The time that this <i>Message</i> was sent to the <i>Client</i> . If this <i>Message</i> is retransmitted to the <i>Client</i> , this parameter contains the time it was first transmitted to the <i>Client</i> .
notificationData []	Extensible Parameter NotificationData	<p>The list of <i>NotificationData structures</i>.</p> <p>The <i>NotificationData</i> parameter type is an extensible parameter type specified in Clause 8.5. It specifies the types of <i>Notifications</i> that can be sent. The <i>ExtensibleParameter</i> type is specified in Clause 8.5.</p> <p>Notifications of the same type should be grouped into one <i>NotificationData</i> element. If a <i>Subscription</i> contains <i>MonitoredItems</i> for events and data, this array should have not more than 2 elements. If the <i>Subscription</i> contains <i>MonitoredItems</i> only for data or only for events, the array size should always be one for this <i>Subscription</i>.</p>

### 7.14 NumericRange

This parameter is defined in Table 140. A formal BNF definition of the numeric range can be found in Appendix A3.

**Table 140 – NumericRange**

Name	Type	Description
NumericRange	String	<p>A number or a numeric range. The syntax for the string contains one of the following two constructs.</p> <p>The first construct is the string representation of an individual integer. For example, “6” is valid, but “6.0” and “3.2” are not. The minimum and maximum values that can be expressed are defined by the use of this parameter and not by this parameter type definition.</p> <p>The second construct is a range represented by two integers separated by the colon (“:”) character. The first integer must always have a lower value than the second. For example, “5:7” is valid, while “7:5” and “5:5” are not. The minimum and maximum values that can be expressed by these integers are defined by the use of this parameter, and not by this parameter type definition.</p> <p>No other characters, including white-space characters, are permitted.</p> <p>A null string indicates that this parameter is not used.</p>

### 7.15 QueryDataSet

The components of this parameter are defined in Table 141.

**Table 141 – QueryDataSet**

Name	Type	Description
QueryDataSet	structure	Data related to a <i>Node</i> returned in a <i>Query response</i> .
nodeId	ExpandedNodeId	The <i>NodeId</i> for this <i>Node</i> description.
typeDefinitionNode	ExpandedNodeId	The <i>NodeId</i> for the type definition for this <i>Node</i> description.
values[]	BaseDataType	Values for the selected <i>Attributes</i> . The order of returned items matches the order of the requested items. There is an entry for each requested item for the given <i>TypeDefinitionNode</i> that matches the selected instance, this includes any related nodes that were specified using a relative path from the selected instance’s <i>TypeDefinitionNode</i> . If no values were found for a given requested item a null value is return for that item. If multiple values exist for a requested item then an array of values is returned. If the requested item is a reference then a <i>ReferenceDescription</i> or array of <i>ReferenceDescriptions</i> are returned for that item.

## 7.16 ReadValueId

The components of this parameter are defined in Table 142.

**Table 142 – ReadValueId**

Name	Type	Description
ReadValueId	structure	Identifier for an item to read or to monitor.
nodeId	NodeId	<i>NodeId</i> of a <i>Node</i> .
attributeId	IntegerId	Id of the <i>Attribute</i> . This must be a valid <i>Attribute</i> id. The <i>IntegerId</i> is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA Part 6].
indexRange	NumericRange	This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. If a range of elements is specified, the values are returned as a composite. The first element is identified by index 0 (zero). The <i>NumericRange</i> type is defined in Clause 7.14. This parameter is null if the specified <i>Attribute</i> is not an array or a structure. However, if the specified <i>Attribute</i> is an array or a structure, and this parameter is null, then all elements are to be included in the range.
dataEncoding	QualifiedName	This parameter specifies the <i>BrowseName</i> of the <i>DataTypeEncoding</i> that the <i>Server</i> should use when returning the <i>Value Attribute</i> of a <i>Variable</i> . It is an error to specify this parameter for other <i>Attributes</i> . A <i>Client</i> can discover what <i>DataTypeEncodings</i> are available by following the <i>HasEncoding Reference</i> from the <i>DataType Node</i> for a <i>Variable</i> . UA defines standard <i>BrowseNames</i> which <i>Servers</i> must recognize even if the <i>DataType Nodes</i> are not visible in the <i>Server address space</i> . These <i>BrowseNames</i> are:  Default Binary    The default or native binary (or non-XML) encoding. Default XML    The default XML encoding.  Each <i>DataType</i> must support at least one of these standard encodings. <i>DataTypes</i> that do not have a true binary encoding (e.g. they only have a non-XML text encoding) should use the <i>Default Binary</i> name to identify the encoding that is considered to be the default non-XML encoding. <i>DataTypes</i> that support at least one XML-based encoding must identify one of the encodings as the <i>Default XML</i> encoding. Other standards bodies may define other well-known data encodings that could be supported. If this parameter is not specified then the <i>Server</i> must choose either the <i>Default Binary</i> or <i>Default XML</i> encoding according to what <i>Message encoding</i> (see [UA Part 6]) is used for the <i>Session</i> . If the <i>Server</i> does not support the encoding that matches the <i>Message encoding</i> then the <i>Server</i> must choose the default encoding that it does support. If this parameter is specified for a <i>MonitoredItem</i> , the <i>Server</i> must set the <i>StructureChanged</i> bit in the <i>StatusCodes</i> (see Clause 7.28) if the <i>DataTypeEncoding</i> changes. The <i>DataTypeEncoding</i> changes if the <i>DataTypeVersion</i> of the <i>DataTypeDescription</i> or the <i>DataTypeDictionary</i> associated with the <i>DataTypeEncoding</i> changes.

## 7.17 ReferenceDescription

The components of this parameter are defined in Table 143.

**Table 143 – ReferenceDescription**

Name	Type	Description
ReferenceDescription	structure	Reference parameters returned for the browse Service and exported by the <i>ExportNodeService</i> .
referenceTypeId	Nodeld	<i>Nodeld</i> of the <i>ReferenceType</i> that defines the <i>Reference</i> .
isForward	Boolean	If the value is TRUE, the <i>Server</i> followed a forward <i>Reference</i> . If the value is FALSE, the <i>Server</i> followed a inverse <i>Reference</i> .
targetNodeInfo	structure	Information that describes the <i>TargetNode</i> of the <i>Reference</i>
nodeId	Expanded Nodeld	<i>Nodeld</i> of the <i>TargetNode</i> as assigned by the <i>Server</i> identified by the <i>Server</i> index. The <i>ExpandedNodeId</i> type is defined in Clause 7.7. If the <i>Server</i> index indicates that the <i>TargetNode</i> is a remote <i>Node</i> , then the <i>nodeId</i> must contain the absolute namespace URI. If the <i>TargetNode</i> is a local <i>Node</i> the <i>nodeId</i> must contain the namespace index.
browseName <sup>1</sup>	QualifiedName	The <i>BrowseName</i> of the <i>TargetNode</i> .
displayName	LocalizedText	The <i>DisplayName</i> of the <i>TargetNode</i> .
nodeClass <sup>1</sup>	NodeClass	<i>NodeClass</i> of the <i>TargetNode</i> .
typeDefinition <sup>1</sup>	Expanded Nodeld	Type definition <i>Nodeld</i> of the <i>TargetNode</i> .

Notes:

1 If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *TargetNode* *browseName*, *nodeClass* and *typeDefinition* may be null or empty. If they are not, they might not be up to date because the local *Server* might not continuously monitor the remote *Server* for changes.

## 7.18 RelativePath

A *RelativePath* is a string that describes a sequence of *References* and *Nodes* to follow. The components of a *RelativePath* are specified in Table 144. A formal BNF definition of the *RelativePath* can be found in Appendix A2.

**Table 144 – RelativePath**

Symbol	Meaning							
/	The forward slash character indicates that the Server is to follow any subtype of <i>HierarchicalReferences</i> .							
.	The period (dot) character indicates that the Server is to follow any subtype of a <i>Aggregates ReferenceType</i> .							
<ns:ReferenceType>	<p>A string delimited by the '&lt;' and '&gt;' symbols specifies the <i>BrowseName</i> of a <i>ReferenceType</i> to follow. A '!' in front of the <i>BrowseName</i> is used to indicate that the inverse <i>Reference</i> should be followed. If the <i>BrowseName</i> of a <i>ReferenceType</i> with subtypes is specified then any <i>References</i> of the subtypes are followed as well.</p> <p>The <i>BrowseName</i> may be qualified with a namespace index (indicated by a numeric prefix followed by a colon). This namespace index is used to specify the namespace component of the <i>BrowseName</i> for the <i>ReferenceType</i>. If the namespace prefix is omitted then namespace index 0 is used.</p>							
ns:BrowseName	<p>A string that follows a '/', '.' or '&gt;' symbol specifies the <i>BrowseName</i> of a target <i>Node</i> to return or follow. This <i>BrowseName</i> may be prefixed by its namespace index. If the namespace prefix is omitted then namespace index 0 is used.</p> <p>Omitting the final <i>BrowseName</i> from a path is equivalent to a wildcard operation that matches all <i>Nodes</i> which are the target of the <i>Reference</i> specified by the path.</p>							
&	<p>The &amp; sign character is the escape character. It is used to specify reserved characters that appear within a <i>BrowseName</i>. A reserved character is escaped by inserting the '&amp;' in front of it. Examples of <i>BrowseNames</i> with escaped characters are:</p> <table> <thead> <tr> <th>Received browse path name</th> <th>Resolves to</th> </tr> </thead> <tbody> <tr> <td>"&amp;/Name_1"</td> <td>"/Name_1"</td> </tr> <tr> <td>"&amp;.Name_2"</td> <td>".Name_2"</td> </tr> <tr> <td>"&amp;:Name_3"</td> <td">":Name_3"</td"></tr></tbody></table>	Received browse path name	Resolves to	"&/Name_1"	"/Name_1"	"&.Name_2"	".Name_2"	"&:Name_3"
Received browse path name	Resolves to							
"&/Name_1"	"/Name_1"							
"&.Name_2"	".Name_2"							
"&:Name_3"								
"&&Name_4"	"&Name_4"							

Table 145 provides examples of *RelativePaths*.

**Table 145 – RelativePath Examples**

Browse Path	Description
"/Block&.Output"	Follows any hierarchical <i>Reference</i> with target <i>BrowseName</i> = "Block.Output".
"/Truck.NodeVersion"	Follows any hierarchical <i>Reference</i> with target <i>BrowseName</i> = "Truck" and from there a <i>HasProperty</i> or <i>HasComponent</i> <i>Reference</i> to a target with <i>BrowseName</i> "NodeVersion".
"<ConnectedTo>Boiler/HeatSensor"	Follows any <i>Reference</i> with a <i>BrowseName</i> or <i>InverseName</i> = 'ConnectedTo' and finds targets with <i>BrowseName</i> = 'Boiler'. From there follows any hierarchical <i>Reference</i> and find targets with <i>BrowseName</i> = 'HeatSensor'.
"<ConnectedTo>Boiler/"	Follows any <i>Reference</i> with a <i>BrowseName</i> or <i>InverseName</i> = 'ConnectedTo' and finds targets with <i>BrowseName</i> = 'Boiler'. From there it finds all targets of hierarchical <i>References</i> .
"<0:HasChild>2:Wheel"	Follows any <i>Reference</i> with a <i>BrowseName</i> = 'HasChild' and qualified with the default UA namespace. Then find targets with <i>BrowseName</i> = 'Wheel' qualified with namespace index '2'.
"<!HasChild>Truck"	Follows any inverse <i>Reference</i> with a <i>BrowseName</i> = 'HasChild' (i.e. follows the <i>HasParent Reference</i> ). Then find targets with <i>BrowseName</i> = 'Truck'. In both cases, the namespace component of the <i>BrowseName</i> is ignored.
"<0:HasChild>"	Finds all targets of <i>References</i> with a <i>BrowseName</i> = 'HasChild' and qualified with the default UA namespace.

## 7.19 RequestHeader

The components of this parameter are defined in Table 146.

**Table 146 – RequestHeader**

Name	Type	Description																																				
RequestHeader	structure	Common parameters for all requests submitted on a <i>Session</i> .																																				
securityHeader	Security Header	Common security parameter. This parameter assigns the SecureChannel related security settings to the <i>Message</i> . The <i>SecurityHeader</i> type is defined in Clause 7.21.																																				
sessionId	IntegerId	Server-unique identifier for the <i>Session</i> . The <i>IntegerId</i> type is defined in Clause 7.9.																																				
timestamp	UtcTime	The time the <i>Client</i> sent the request.																																				
sequenceNumber	Counter	The sequence number of the request. The initial sequence number to use is one, and when the sequence number rolls over, it rolls over to one. Zero is never used. The <i>Counter</i> type is defined in Clause 7.3																																				
returnDiagnostics	UInt32	<p>A bit mask that identifies the types of vendor-specific diagnostics to be returned in <i>diagnosticInfo</i> response parameters. The value of this parameter may consist of zero, one or more of the following values. No value indicates that diagnostics are not to be returned.</p> <table> <thead> <tr> <th>Bit Value</th> <th>Diagnostics to return</th> </tr> </thead> <tbody> <tr> <td>0x0000 0001</td> <td>ServiceLevel / SymbolicId</td> </tr> <tr> <td>0x0000 0002</td> <td>ServiceLevel / LocalizedText</td> </tr> <tr> <td>0x0000 0004</td> <td>ServiceLevel / AdditionalInfo</td> </tr> <tr> <td>0x0000 0008</td> <td>ServiceLevel / Inner <i>StatusCodes</i></td> </tr> <tr> <td>0x0000 0010</td> <td>ServiceLevel / Inner Diagnostics</td> </tr> <tr> <td>0x0000 0020</td> <td>OperationLevel / SymbolicId</td> </tr> <tr> <td>0x0000 0040</td> <td>OperationLevel / LocalizedText</td> </tr> <tr> <td>0x0000 0080</td> <td>OperationLevel / AdditionalInfo</td> </tr> <tr> <td>0x0000 0100</td> <td>OperationLevel / Inner <i>StatusCodes</i></td> </tr> <tr> <td>0x0000 0200</td> <td>OperationLevel / Inner Diagnostics</td> </tr> </tbody> </table> <p>Each of these values is composed of two components, <i>level</i> and <i>type</i>, as described below. If none are requested, as indicated by a 0 value, then diagnostics information is not returned.</p> <p><i>Level:</i></p> <table> <tbody> <tr> <td>ServiceLevel</td> <td>return diagnostics in the <i>diagnosticInfo</i> of the <i>Service</i>.</td> </tr> <tr> <td>OperationLevel</td> <td>return diagnostics in the <i>diagnosticInfo</i> defined for individual operations requested in the <i>Service</i>.</td> </tr> </tbody> </table> <p><i>Type::</i></p> <table> <tbody> <tr> <td>SymbolicId</td> <td>return a namespace-qualified, symbolic identifier for an error or condition. The maximum length of this identifier is 32 characters.</td> </tr> <tr> <td>LocalizedText</td> <td>return up to 256 bytes of localized text that describes the symbolic id</td> </tr> <tr> <td>AdditionalInfo</td> <td>return a byte string that contains additional diagnostic information, such as a memory image. The format of this byte string is vendor-specific, and may depend on the type of error or condition encountered.</td> </tr> <tr> <td>InnerStatusCode</td> <td>return the inner <i>StatusCodes</i> associated with the operation or <i>Service</i>.</td> </tr> <tr> <td>InnerDiagnostics</td> <td>return the inner diagnostic info associated with the operation or <i>Service</i>. The contents of the inner diagnostic info structure are determined by other bits in the mask. Note that setting this bit could cause multiple levels of nested diagnostic info structures to be returned.</td> </tr> </tbody> </table>	Bit Value	Diagnostics to return	0x0000 0001	ServiceLevel / SymbolicId	0x0000 0002	ServiceLevel / LocalizedText	0x0000 0004	ServiceLevel / AdditionalInfo	0x0000 0008	ServiceLevel / Inner <i>StatusCodes</i>	0x0000 0010	ServiceLevel / Inner Diagnostics	0x0000 0020	OperationLevel / SymbolicId	0x0000 0040	OperationLevel / LocalizedText	0x0000 0080	OperationLevel / AdditionalInfo	0x0000 0100	OperationLevel / Inner <i>StatusCodes</i>	0x0000 0200	OperationLevel / Inner Diagnostics	ServiceLevel	return diagnostics in the <i>diagnosticInfo</i> of the <i>Service</i> .	OperationLevel	return diagnostics in the <i>diagnosticInfo</i> defined for individual operations requested in the <i>Service</i> .	SymbolicId	return a namespace-qualified, symbolic identifier for an error or condition. The maximum length of this identifier is 32 characters.	LocalizedText	return up to 256 bytes of localized text that describes the symbolic id	AdditionalInfo	return a byte string that contains additional diagnostic information, such as a memory image. The format of this byte string is vendor-specific, and may depend on the type of error or condition encountered.	InnerStatusCode	return the inner <i>StatusCodes</i> associated with the operation or <i>Service</i> .	InnerDiagnostics	return the inner diagnostic info associated with the operation or <i>Service</i> . The contents of the inner diagnostic info structure are determined by other bits in the mask. Note that setting this bit could cause multiple levels of nested diagnostic info structures to be returned.
Bit Value	Diagnostics to return																																					
0x0000 0001	ServiceLevel / SymbolicId																																					
0x0000 0002	ServiceLevel / LocalizedText																																					
0x0000 0004	ServiceLevel / AdditionalInfo																																					
0x0000 0008	ServiceLevel / Inner <i>StatusCodes</i>																																					
0x0000 0010	ServiceLevel / Inner Diagnostics																																					
0x0000 0020	OperationLevel / SymbolicId																																					
0x0000 0040	OperationLevel / LocalizedText																																					
0x0000 0080	OperationLevel / AdditionalInfo																																					
0x0000 0100	OperationLevel / Inner <i>StatusCodes</i>																																					
0x0000 0200	OperationLevel / Inner Diagnostics																																					
ServiceLevel	return diagnostics in the <i>diagnosticInfo</i> of the <i>Service</i> .																																					
OperationLevel	return diagnostics in the <i>diagnosticInfo</i> defined for individual operations requested in the <i>Service</i> .																																					
SymbolicId	return a namespace-qualified, symbolic identifier for an error or condition. The maximum length of this identifier is 32 characters.																																					
LocalizedText	return up to 256 bytes of localized text that describes the symbolic id																																					
AdditionalInfo	return a byte string that contains additional diagnostic information, such as a memory image. The format of this byte string is vendor-specific, and may depend on the type of error or condition encountered.																																					
InnerStatusCode	return the inner <i>StatusCodes</i> associated with the operation or <i>Service</i> .																																					
InnerDiagnostics	return the inner diagnostic info associated with the operation or <i>Service</i> . The contents of the inner diagnostic info structure are determined by other bits in the mask. Note that setting this bit could cause multiple levels of nested diagnostic info structures to be returned.																																					
auditLogEntryId	String	An identifier that identifies the <i>Client's</i> security audit log entry associated with this request. An empty string value means that this parameter is not used.																																				
timeoutHint	Duration	This timeout is used in the <i>Client</i> side <i>Communication Stack</i> to set the timeout on a per-call base. For a <i>Server</i> this timeout is only a hint and can be used to cancel long running operations to free resources. If the <i>Server</i> detects a timeout, he can cancel the operation by sending the <i>Service</i> result <i>Bad_Timeout</i> . The <i>Server</i> should wait at minimum the timeout after he received the request before cancelling the operation. The value of 0 indicates no timeout.																																				
additionalHeader	Extensible Parameter AdditionalHeader	Reserved for future use. Applications that do not understand the header should ignore it.																																				

## 7.20 ResponseHeader

The components of this parameter are defined in Table 147.

**Table 147 – ResponseHeader**

Name	Type	Description
ResponseHeader	structure	Common parameters for all responses.
securityHeader	Security Header	Common security parameter. This parameter assigns the SecureChannel-related security settings to the <i>Message</i> . The <i>SecurityHeader</i> type is defined in Clause 7.21.
sessionId	IntegerId	Server-unique identifier for the <i>Session</i> . The <i>IntegerId</i> type is defined in Clause 7.9.
timestamp	UtcTime	The time the <i>Server</i> sent the response.
sequenceNumber	Counter	The sequence number given by the <i>Client</i> to the request.
serviceResult	StatusCodes	Standard, OPC UA-defined result of the <i>Service</i> invocation. The <i>StatusCodes</i> type is defined in Clause 7.28.
diagnosticInfo	DiagnosticInfo	Diagnostic information for the <i>Service</i> invocation. This parameter is empty if diagnostics information was not requested in the request header. The <i>DiagnosticInfo</i> type is defined in Clause 7.5.
stringTable []	String	There is one string in this list for each unique namespace, symbolic identifier, and localized text string contained in all of the diagnostics information parameters contained in the response (see Clause 7.5). Each is identified within this table by its zero-based index.
additionalHeader	Extensible Parameter AdditionalHeader	Reserved for future use. Applications that do not understand the header should ignore it.

## 7.21 SecurityHeader

The signing and encryption parameters for all *Messages* in the context of a *SecureChannel* are passed in the *securityHeader* parameter of the *RequestHeader* or *ResponseHeader* of the *Messages*. This parameter may be passed automatically by the *Communication Stack* in some of the mappings defined in [UA Part 6]. If this is the case, then this parameter is not part of the *RequestHeader* or *ResponseHeader* parameters.

[UA Part 6] defines the structure of the *SecurityHeader* for the different mappings.

## 7.22 SecurityPolicy

Security policies specify which encryption and hashing functions are used for which functions and which protocols will perform the various functions. The security policies are specified in more detail in [UA Part 2].

The components of this parameter are defined in Table 148.

**Table 148 – SecurityPolicy**

Name	Type	Description
SecurityPolicy	structure	Specifies what security policies the Server requires.
uri	String	A URI that identifies the <i>SecurityPolicy</i> . UA defines the standard <i>SecurityPolicies</i> in [UA Part 7].
digest	String	The URI of the digest algorithm to use.
symmetricSignature	String	The URI of the symmetric signature algorithm to use.
symmetricKeyWrap	String	The URI of the symmetric key wrap algorithm to use.
symmetricKeyEncryption	String	The URI of the symmetric key encryption algorithm to use.
symmetricKeyLength	Int32	The length in bits for the symmetric key.
asymmetricSignature	String	The URI of the asymmetric signature algorithm to use.
asymmetricKeyWrap	String	The URI of the asymmetric key wrap algorithm to use.
asymmetricKeyEncryption	String	The URI of the asymmetric key encryption algorithm to use.
asymmetricKeyMinLength	Int32	The minimum length in bits for the asymmetric key.
asymmetricKeyMaxLength	Int32	The maximum length in bits for the asymmetric key.
derivedKey	String	The key derivation algorithm to use.
derivedEncryptionKeyLength	Int32	The length of the derived key used for encryption.
derivedSignatureKeyLength	Int32	The length of the derived key used for signatures.

## 7.23 ServerDescription

The components of this parameter are defined in Table 149.

**Table 149 – ServerDescription**

Name	Type	Description
ServerDescription	structure	Specifies a Server that is available.
serverUri	String	The globally unique identifier for the Server.
serverName	LocalizedText	A localized descriptive name for the Server.
serverType	Enum ServerType	The type of server. This value is an enumeration with one of the following values: SERVER_0      The server is a regular Server. DISCOVERY_1    The server is a <i>DiscoveryServer</i> .

## 7.24 ServiceFault

The components of this parameter are defined in Table 150.

The *ServiceFault* parameter is returned instead of the Service response message when a service level error occurs. The *sessionId* and *sequenceNumber* in the *ResponseHeader* should be set to what was provided in the *RequestHeader* even if these values were not valid. The level of diagnostics returned in the *ResponseHeader* is specified by the *returnDiagnostics* parameter in the *RequestHeader*.

The exact use of this parameter depends on the mappings defined in [UA Part 6].

**Table 150 – ServiceFault**

Name	Type	Description
ServiceFault	structure	An error response sent when a service level error occurs.
responseHeader	ResponseHeader	Common response parameters (see Clause 7.20 for <i>ResponseHeader</i> definition).

## 7.25 SignatureData

The components of this parameter are defined in Table 151.

**Table 151 – SignatureData**

Name	Type	Description
SignatureData	structure	Contains a digital signature created with a <i>Certificate</i> .
signature	ByteString	This is a signature generated with the private key associated with a <i>Certificate</i> .
signatureAlgorithm	String	A string containing the URI of the <i>signatureAlgorithm</i> . The list of UA-defined names that may be used is specified in [UA Part 7].

## 7.26 SignedSoftwareCertificate

The components of this parameter are defined in Table 152. See [UA Part 6] for the details on the signing of *Certificates*.

**Table 152 – SignedSoftwareCertificate**

Name	Type	Description
SignedSoftwareCertificate	structure	A <i>Certificate</i> that identifies the capabilities of a <i>Client</i> or a <i>Server</i> .
certificateData	ByteString	Contents of the <i>Certificate</i> . The fields of this parameter are defined in Table 153. These fields are serialized using the OPC Binary Encoding Rules and transferred in a byte string. The structure of the <i>softwareCertificate</i> type is specified in Clause 7.27.
issuerSignature	ByteString	Signature of the <i>SoftwareCertificate</i> body byte string. This signature is generated by the <i>Certificate</i> issuer using its <i>PrivateKey</i> . To validate the <i>Certificate</i> , hashes the <i>Certificate</i> data using the issuer algorithm specified in the <i>Certificate</i> and then verifies the signature using the issuer's <i>PublicKey</i> . The <i>PublicKey</i> of the <i>Certificate</i> issuer is obtained by the receiver through external means.

## 7.27 SoftwareCertificate

The components of this parameter are defined in Table 153.

**Table 153 – SoftwareCertificate**

Name	Type	Description
SoftwareCertificate	structure	<i>Certificate</i> signed by the issuer.
serverInfo	BuildInfo	Information that identifies the software build that this <i>Certificate</i> certifies.
issuedBy	String	URI of the certifying authority.
issueDate	UtcTime	Date and time that the <i>Certificate</i> was issued.
expirationDate	UtcTime	Date and time that the <i>Certificate</i> expires.
applicationCertificate	ByteString	The DER encoded form of the X.509 <i>Certificate</i> which is assigned to the application.
issuerCertificateThumbprint	ByteString	The thumbprint of the issuer's X.509 <i>Certificate</i> used to sign the <i>SoftwareCertificate</i> .
issuerSignatureAlgorithm	String	The algorithm used to create the issuer's signature.
supportedProfiles []	structure	List of supported <i>Profiles</i>
profileUri	String	URI that identifies the <i>Profile</i>
profileName	String	Human-readable name for the <i>Profile</i> . This name does not have to be globally-unique.
complianceLevel	enum Compliance Level	An enumeration that specifies the compliance level of the <i>Profile</i> . It has the following values :  UNTESTED_0 the profiled capability has not been tested successfully PARTIAL_1 the profiled capability has been tested and has passed critical tests, as defined by the certifying authority. SELFTESTED_2 the profiled capability has been successfully tested using a self-test system authorized by the certifying authority. CERTIFIED_3 the profiled capability has been successfully tested by a testing organisation authorized the certifying authority.

## 7.28 StatusCode

A *StatusCode* in UA is numerical value that is used to report the outcome of an operation performed by a UA Server. This code may have associated diagnostic information that describes the status in more detail; however, the code by itself is intended to provide *Client* applications with enough information to make decisions on how to process the results of a UA Service.

The *StatusCode* is a 32-bit unsigned integer. The top 16 bits represent the numeric value of the code that must be used for detecting specific errors or conditions. The bottom 16 bits are bit flags that contain additional information but do not affect the meaning of the *StatusCode*.

All UA *Clients* must always check the *StatusCode* associated with a result before using it. Results that have an uncertain/warning status associated with them must be used with care since these results might not be valid in all situations. Results with a bad/failed status must never be used.

UA Servers should return good/success *StatusCodes* if the operation completed normally and the result is always useful. Different *StatusCode* values can provide additional information to the *Client*.

UA Servers should use uncertain/warning *StatusCodes* if they could not complete the operation in the manner requested by the *Client*, however, the operation did not fail entirely.

The exact bit assignments are shown in Table 154.

**Table 154 – StatusCode Bit Assignments**

Field	Bit Range	Description																		
Severity	30:31	<p>Indicates whether the <i>StatusCode</i> represents a good, bad or uncertain condition. These bits have the following meanings:</p> <table> <tr> <td>Good</td><td>00</td><td>Indicates that the operation was successful and the associated results may be used.</td></tr> <tr> <td>Success</td><td>01</td><td>Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.</td></tr> <tr> <td>Uncertain</td><td>01</td><td>Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.</td></tr> <tr> <td>Warning</td><td>10</td><td>Indicates that the operation failed and any associated results cannot be used.</td></tr> <tr> <td>Bad Failure</td><td>10</td><td>Indicates that the operation failed and any associated results cannot be used.</td></tr> <tr> <td>Reserved</td><td>11</td><td>Reserved for future use. All <i>Clients</i> should treat a <i>StatusCode</i> with this severity as "Bad".</td></tr> </table>	Good	00	Indicates that the operation was successful and the associated results may be used.	Success	01	Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.	Uncertain	01	Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.	Warning	10	Indicates that the operation failed and any associated results cannot be used.	Bad Failure	10	Indicates that the operation failed and any associated results cannot be used.	Reserved	11	Reserved for future use. All <i>Clients</i> should treat a <i>StatusCode</i> with this severity as "Bad".
Good	00	Indicates that the operation was successful and the associated results may be used.																		
Success	01	Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.																		
Uncertain	01	Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.																		
Warning	10	Indicates that the operation failed and any associated results cannot be used.																		
Bad Failure	10	Indicates that the operation failed and any associated results cannot be used.																		
Reserved	11	Reserved for future use. All <i>Clients</i> should treat a <i>StatusCode</i> with this severity as "Bad".																		
Reserved	29:28	Reserved for future use. Must always be zero.																		
SubCode	16:27	The code is a numeric value assigned to represent different conditions. Each code has a symbolic name and a numeric value. All descriptions in the UA specification refer to the symbolic name. [UA Part 6] maps the symbolic names onto a numeric value.																		
Structure Changed	15:15	<p>Indicates that the structure of the associated data value has changed since the last <i>Notification</i>. <i>Clients</i> should not process the data value unless they re-read the metadata.</p> <p>Servers must set this bit if the <i>DataTypeEncoding</i> used for a <i>Variable</i> changes. Clause 7.16 describes how the <i>DataTypeEncoding</i> is specified for a <i>Variable</i>.</p> <p>The bit is also set if the data type <i>Attribute</i> of the <i>Variable</i> changes. A <i>Variable</i> with data type <i>BaseDataType</i> does not require the bit to be set when the data type changes.</p> <p>Servers must also set this bit if the length of a fixed-length array <i>Variable</i> changes.</p> <p>This bit is provided to warn <i>Clients</i> that parse complex data values that their parsing routines could fail because the serialized form of the data value has changed.</p> <p>This bit has meaning only for <i>StatusCodes</i> returned as part of a data change <i>Notification</i>. <i>StatusCodes</i> used in other contexts must always set this bit to zero.</p>																		
Semantics Changed	14:14	<p>Indicates that the semantics of the associated data value have changed. <i>Clients</i> should not process the data value until they re-read the metadata associated with the <i>Variable</i>.</p> <p>Servers should set this bit if the metadata has changed in way that could cause application errors if the <i>Client</i> does not re-read the metadata. For example, a change to the engineering units could create problems if the <i>Client</i> uses the value to perform calculations.</p> <p>[UA Part 8] defines the conditions where a <i>Server</i> must set this bit for a DA <i>Variable</i>. Other specifications may define additional conditions. A <i>Server</i> may define other conditions that cause this bit to be set.</p> <p>This bit has meaning only for <i>StatusCodes</i> returned as part of a data change <i>Notification</i>. <i>StatusCodes</i> used in other contexts must always set this bit to zero.</p>																		
Reserved	12:13	Reserved for future use. Must always be zero.																		
InfoType	10:11	<p>The type of information contained in the info bits. These bits have the following meanings:</p> <table> <tr> <td>NotUsed</td><td>00</td><td>The info bits are not used and must be set to zero.</td></tr> <tr> <td>DataValue</td><td>01</td><td>The <i>StatusCode</i> and its info bits are associated with a data value returned from the Server.</td></tr> <tr> <td>Reserved</td><td>1X</td><td>Reserved for future use. The info bits must be ignored.</td></tr> </table>	NotUsed	00	The info bits are not used and must be set to zero.	DataValue	01	The <i>StatusCode</i> and its info bits are associated with a data value returned from the Server.	Reserved	1X	Reserved for future use. The info bits must be ignored.									
NotUsed	00	The info bits are not used and must be set to zero.																		
DataValue	01	The <i>StatusCode</i> and its info bits are associated with a data value returned from the Server.																		
Reserved	1X	Reserved for future use. The info bits must be ignored.																		
InfoBits	0:9	Additional information bits that qualify the <i>StatusCode</i> . The structure of these bits depends on the Info Type field.																		

Table 155 describes the structure of the *InfoBits* when the Info Type is set to *DataValue* (01).

**Table 155 – DataValue InfoBits**

Info Type	Bit Range	Description																					
LimitBits	8:9	The limit bits associated with the data value. The limits bits have the following meanings: <table> <thead> <tr> <th>Limit</th><th>Bits</th><th>Description</th></tr> </thead> <tbody> <tr> <td>None</td><td>00</td><td>The value is free to change.</td></tr> <tr> <td>Low</td><td>01</td><td>The value is at the lower limit for the data source.</td></tr> <tr> <td>High</td><td>10</td><td>The value is at the higher limit for the data source.</td></tr> <tr> <td>Constant</td><td>11</td><td>The value is constant and cannot change.</td></tr> </tbody> </table>	Limit	Bits	Description	None	00	The value is free to change.	Low	01	The value is at the lower limit for the data source.	High	10	The value is at the higher limit for the data source.	Constant	11	The value is constant and cannot change.						
Limit	Bits	Description																					
None	00	The value is free to change.																					
Low	01	The value is at the lower limit for the data source.																					
High	10	The value is at the higher limit for the data source.																					
Constant	11	The value is constant and cannot change.																					
Overflow	7	If this bit is set, not every detected change has been returned since the Server's queue buffer for the <i>MonitoredItem</i> reached its limit and had to purge out data.																					
Reserved	5:6	Reserved for future use. Must always be zero.																					
HistorianBits	0:4	These bits are set only when reading historical data. They indicate where the data value came from and provide information that affects how the <i>Client</i> uses the data value. The historian bits have the following meaning: <table> <tbody> <tr> <td>Raw</td><td>XXX00</td><td>A raw data value.</td></tr> <tr> <td>Calculated</td><td>XXX01</td><td>A data value which was calculated.</td></tr> <tr> <td>Interpolated</td><td>XXX10</td><td>A data value which was interpolated.</td></tr> <tr> <td>Reserved</td><td>XXX11</td><td>Undefined.</td></tr> <tr> <td>Partial</td><td>XX1XX</td><td>A data value which was calculated with an incomplete interval.</td></tr> <tr> <td>Extra Data</td><td>X1XXX</td><td>A raw data value that hides other data at the same timestamp.</td></tr> <tr> <td>Multi Value</td><td>1XXXX</td><td>Multiple values match the aggregate criteria (i.e. multiple minimum values at different timestamps within the same interval)  [UA Part 11] describes how these bits are used in more detail.</td></tr> </tbody> </table>	Raw	XXX00	A raw data value.	Calculated	XXX01	A data value which was calculated.	Interpolated	XXX10	A data value which was interpolated.	Reserved	XXX11	Undefined.	Partial	XX1XX	A data value which was calculated with an incomplete interval.	Extra Data	X1XXX	A raw data value that hides other data at the same timestamp.	Multi Value	1XXXX	Multiple values match the aggregate criteria (i.e. multiple minimum values at different timestamps within the same interval) [UA Part 11] describes how these bits are used in more detail.
Raw	XXX00	A raw data value.																					
Calculated	XXX01	A data value which was calculated.																					
Interpolated	XXX10	A data value which was interpolated.																					
Reserved	XXX11	Undefined.																					
Partial	XX1XX	A data value which was calculated with an incomplete interval.																					
Extra Data	X1XXX	A raw data value that hides other data at the same timestamp.																					
Multi Value	1XXXX	Multiple values match the aggregate criteria (i.e. multiple minimum values at different timestamps within the same interval) [UA Part 11] describes how these bits are used in more detail.																					

## Common StatusCodes

Table 156 defines the common *StatusCodes* for all *Service* results. [UA Part 6] maps the symbolic names to a numeric value.

**Table 156 – Common Service Result Codes**

Symbolic Id	Description
Good	The operation was successful.
Good_CompletesAsynchronously	The processing will complete asynchronously.
Good_CommunicationEvent	The communication layer has raised an event
Good_ShutdownEvent	The system is shutting down.
Good_CallAgain	The operation is not finished and needs to be called again.
Good_NonCriticalTimeout	A non-critical timeout occurred.
Good_SubscriptionTransferred	The subscription was transferred to another session.
Bad_UnexpectedError	An unexpected error occurred.
Bad_InternalError	An internal error occurred as a result of a programming or configuration error.
Bad_OutOfMemory	Not enough memory to complete the operation.
Bad_ResourceUnavailable	An operating system resource is not available.
Bad_CommunicationError	A low level communication error occurred.
Bad_ConnectionRejected	Could not establish a network connection to remote server.
Bad_ConnectionClosed	The network connection has been closed.
Bad_InvalidState	The operation cannot be completed because the object is closed, uninitialized or in some other invalid state.
Bad_EncodingError	Encoding halted because of invalid data in the objects being serialized.
Bad_DecodingError	Decoding halted because of invalid data in the stream.
Bad_EndOfStream	Cannot move beyond end of the stream.
Bad_NoDataAvailable	No data is currently available for reading from a non-blocking stream.
Bad_ExpectedStreamToBlock	The stream did not return all data requested (possibly because it is a non-blocking stream).
Bad_WaitingForResponse	The asynchronous operation is waiting for a response.
Bad_OperationAbandoned	The asynchronous operation was abandoned by the caller.
Bad_WouldBlock	Non blocking behaviour is required and the operation would block.
Bad_SyntaxError	A value had an invalid syntax.
Bad_ConnectionIdInvalid	The specified connection id was not valid.
Bad_UnknownResponse	An unrecognized response was received from the server.
Bad_Timeout	The operation timed out.
Bad_MaxConnectionsReached	The operation could not be finished because all available connections are in use.
Bad_ServiceUnsupported	The server does not support the requested service.
Bad_Shutdown	The operation was cancelled because the application is shutting down
Bad_Disconnect	The operation was cancelled because the network connection with the server was closed.
Bad_ServerNotConnected	The operation could not complete because the client is not connected to the server.
Bad_ServerHalted	The server has stopped and cannot process any requests.
Bad_InvalidArgument	One or more arguments are invalid. Each service defines parameter-specific <i>StatusCodes</i> and these <i>StatusCodes</i> must be used instead of this general error code. This error code must be used only by the communication stack and in services where it is defined in the list of valid <i>StatusCodes</i> for the service.
Bad_NothingToDo	There was nothing to do because the client passed a list of operations with no elements.
Bad_DataTypeidUnknown	The extension object cannot be (de)serialized because the data type id is not recognized.
Bad_ExtensibleParameterInvalid	The extensible parameter provided is not a valid for the service.
Bad_ExtensibleParameterUnsupported	The extensible parameter provided is valid but the server does not support it.
Bad_CertificateInvalid	The certificate is not valid.
Bad_CertificateExpired	The certificate is expired or not yet valid.
Bad_CertificateRevoked	The certificate has been revoked by the certification authority.
Bad_CertificateUntrusted	The certificate is valid; however, the server does not recognize it as a trusted certificate.
Bad_UserAccessDenied	User does not have permission to perform the requested operation.
Bad_IdentityTokenInvalid	The user identity token is not valid.
Bad_IdentityTokenRejected	The user identity token is valid but the server has rejected it.
Bad_SecureChannelIdInvalid	The specified secure channel is not longer valid.
Bad_SequenceNumberInvalid	The sequence number is less than that of a previous update request.

Bad_InvalidTimestamp	The timestamp is outside the range allowed by the server.
Bad_SignatureInvalid	The message signature is invalid.
Bad_NonceInvalid	The nonce does appear to be not a random value or it is not the correct length.
Bad_SessionIdInvalid	The session id is not valid.
Bad_SessionClosed	The session was closed by the client.
Bad_SessionNotActivated	The session cannot be used because ActivateSession has not been called.
Bad_SubscriptionIdInvalid	The subscription id is not valid.
Bad_NoSubscription	There is no subscription available for this session.
Bad_RequestHeaderInvalid	The header for the request is missing or invalid.
Bad_ContinuationPointInvalid	The continuation point is no longer valid.
Bad_TimestampsToReturnInvalid	The timestamps to return parameter is invalid.

Table 157 defines the common *StatusCodes* for all operation level results. [UA Part 6] maps the symbolic names to a numeric value. The common *Service* result codes can be also contained in the operation level.

**Table 157 – Common Operation Level Result Codes**

Symbolic Id	Description
Good_Overload	Sampling has slowed down due to resource limitations.
Good_Clamped	The value written was accepted but was clamped.
Uncertain	The value is uncertain but no specific reason is known
Bad	The value is bad but no specific reason is known
Bad_NoCommunication	Communication with the data source is defined, but not established, and there is no last known value available. This status/sub status is used for cached values before the first value is received.
Bad_WaitingForInitialData	Waiting for the server to obtain values from the underlying data source. After creating a <i>MonitoredItem</i> , it may take some time for the server to actually obtain values for these items. In such cases the server can optionally send a <i>Notification</i> with this status prior to the <i>Notification</i> with the first valid value.
Bad_NodeIdInvalid	The syntax of the node id is not valid.
Bad_NodeIdUnknown	The node id refers to a node that does not exist in the server address space.
Bad_AttributeIdInvalid	The attribute is not supported for the specified <i>Node</i> .
Bad_IndexRangeInvalid	The syntax of the index range parameter is invalid.
Bad_IndexRangeNoData	No data exists within the range of indexes specified.
Bad_DataEncodingInvalid	The data encoding is invalid.
Bad_DataEncodingUnsupported	The server does not support the requested data encoding for the node.
Bad_NoReadRights	The access level does not allow reading or subscribing to the <i>Node</i> .
Bad_NoWriteRights	The access level does not allow writing to the <i>Node</i> .
Bad_OutOfRange	The value was out of range.
Bad_NotSupported	The requested operation is not supported.
Bad_NotFound	A requested item was not found or a search operation ended without success.
Bad_ObjectDeleted	The object cannot be used because it has been deleted.
Bad_NotImplemented	Requested operation is not implemented.
Bad_MonitoringModelError	The monitoring mode is invalid.
Bad_MonitoredItemIdInvalid	The monitoring item id does not refer to a valid monitored item.
Bad_FilterNotAllowed	A monitoring filter cannot be used in combination with the attribute specified.
Bad_StructureMissing	A mandatory structured parameter was missing or null.

## 7.29 TimestampsToReturn

The *TimestampsToReturn* is an enumeration that specifies the *Timestamp Attributes* to be transmitted for *MonitoredItems* or *Nodes* in *HistoryRead*. The values of this parameter are defined in Table 158.

**Table 158 – TimestampsToReturn Values**

Value	Description
SOURCE_0	Return the source timestamp. If used in <i>HistoryRead</i> the source timestamp is used to determine which historical data values are returned.
SERVER_1	Return the <i>Server</i> timestamp. If used in <i>HistoryRead</i> the <i>Server</i> timestamp is used to determine which historical data values are returned.
BOTH_2	Return both the source and <i>Server</i> timestamps. If used in <i>HistoryRead</i> the source timestamp is used to determine which historical data values are returned.
NEITHER_3	Return neither timestamp. This is the default value for <i>MonitoredItems</i> if a <i>Variable</i> value is not being accessed. For <i>HistoryRead</i> this is not a valid setting.

## 7.30 UserTokenPolicy

The components of this parameter are defined in Table 159.

**Table 159 – UserTokenPolicy**

Name	Type	Description
UserTokenPolicy	structure	Specifies a <i>UserIdentityToken</i> that a <i>Server</i> will accept.
tokenType	Enum <i>UserIdentityTokenType</i>	The type of user identity token required. This value is an enumeration with one of the following values: DEFAULT_0                          No token is required. USERNAME_1                        A username/password token. CERTIFICATE_2                    An X509v3 certificate token. ISSUEDTOKEN_3                    Any WS-Security defined token.
issuerType	String	A URI indicating the type of token issuer. [UA Part 7] defines URIs for common issuer types. Vendors may specify their own issuer types that describe a vendor defined user database or identity token server. Clients that do not recognize the IssuerType should still be able to attempt a connection if the TokenType is CERTIFICATE or USERNAME. If the TokenType is ISSUEDTOKEN then the issuer type may be some version of WS-Trust.
issuerUrl	String	A URL of the token issuing service. This meaning of this value depends on the <i>issuerType</i>

### 7.31 ViewDescription

The components of this parameter are defined in Table 160.

**Table 160 – ViewDescription**

Name	Type	Description
ViewDescription	structure	Specifies a <i>View</i> .
viewId	NodeId	<i>NodeId</i> of the <i>View</i> to <i>Query</i> . A null value indicates the entire <i>AddressSpace</i> .
timestamp	UtcTime	The time date desired. The corresponding version is the one with the closest previous creation timestamp. Either the <i>Timestamp</i> or the <i>viewVersion</i> parameter may be set by a <i>Client</i> , but not both. If <i>ViewVersion</i> is set this parameter must be null.
viewVersion	UInt32	The version number for the <i>View</i> desired. When <i>Nodes</i> are added to or removed from a <i>View</i> , the value of a <i>View's ViewVersion Property</i> is updated. Either the <i>Timestamp</i> or the <i>viewVersion</i> parameter may be set by a <i>Client</i> , but not both. The <i>ViewVersion Property</i> is defined in [UA Part 3]. If <i>timestamp</i> is set this parameter must be 0. The current view is used if <i>timestamp</i> is null and <i>viewVersion</i> is 0.

## 8 Extensible parameter type definition

### 8.1 Overview

The extensible parameter types can only be extended by additional parts of this multi-part specification.

### 8.2 ExtensibleParameter

The *ExtensibleParameter* defines a data structure with two elements. The *parameterTypeId* specifies the data type encoding of the second element. Therefore the second element is specified as “--”. The *ExtensibleParameter* base type is defined in Table 161.

The following clauses define concrete extensible parameters that are common to OPC UA. Additional parts of this multi-part specification can define additional extensible parameter types.

**Table 161 – ExtensibleParameter Base Type**

Name	Type	Description
ExtensibleParameter	structure	Specifies the details of an extensible parameter type.
parameterTypeId	NodeId	Identifies the data type of the parameter that follows.
parameterData	--	The details for the extensible parameter type.

### 8.3 MonitoringFilter parameters

#### 8.3.1 Overview

The *CreateMonitoredItem* Service allows specifying a filter for each *MonitoredItem*. The *MonitoringFilter* is an extensible parameter whose structure depends on the type of item being monitored. The *parameterTypeIds* are defined in Table 162. Other types can be defined by additional parts of this multi-part specification or other standards based on OPC UA.

**Table 162 – MonitoringFilter parameterTypeIds**

Symbolic Id	Description
DataChangeFilter	The change in a data value that will cause a <i>Notification</i> to be generated.
EventFilter	If a <i>Notification</i> conforms to the <i>EventFilter</i> , the <i>Notification</i> is sent to the <i>Client</i> .

### 8.3.2 DataChangeFilter

The *DataChangeFilter* defines the conditions under which a data change notification should be reported and, optionally, a range or band for value changes where no *DataChange Notification* is generated. This range is called *Deadband*. The *DataChangeFilter* is defined in Table 163.

**Table 163 – DataChangeFilter**

Name	Type	Description								
DataChangeFilter	structure									
trigger	enum DataChangeTrigger	<p>Specifies the conditions under which a data change notification should be reported. It has the following values :</p> <ul style="list-style-type: none"> <li>STATUS_0      Report a notification ONLY if the <i>StatusCodes</i> associated with the value changes. See Table 157 for <i>StatusCodes</i> defined in this Part. [UA Part 8] specifies additional <i>StatusCodes</i> that are valid in particular for device data</li> <li>STATUS_VALUE_1      Report a notification if either the <i>StatusCode</i> or the value change. The <i>Deadband</i> filter can be used in addition for filtering value changes. <b>This is the default setting if no filter is set.</b></li> <li>STATUS_VALUE_TIMESTAMP_2      Report a notification if either <i>StatusCode</i>, value or the <i>SourceTimestamp</i> change. The <i>Deadband</i> filter can be used in addition for filtering value changes.</li> </ul> <p>If the DataChangeFilter is not applied to the monitored item, STATUS_VALUE is the default reporting behaviour.</p>								
deadbandType	UInt32	<p>A bit mask that defines the <i>Deadband</i> type and behaviour.</p> <table> <tr> <td><u>Bit Value</u></td> <td><u>deadbandType</u></td> </tr> <tr> <td>0x0000 0000</td> <td>No <i>Deadband</i> calculation should be applied.</td> </tr> <tr> <td>0x0000 0001</td> <td>AbsoluteDeadband (see below)</td> </tr> <tr> <td>0x0000 0002</td> <td>PercentDeadband (This type is specified in [UA Part 8]).</td> </tr> </table>	<u>Bit Value</u>	<u>deadbandType</u>	0x0000 0000	No <i>Deadband</i> calculation should be applied.	0x0000 0001	AbsoluteDeadband (see below)	0x0000 0002	PercentDeadband (This type is specified in [UA Part 8]).
<u>Bit Value</u>	<u>deadbandType</u>									
0x0000 0000	No <i>Deadband</i> calculation should be applied.									
0x0000 0001	AbsoluteDeadband (see below)									
0x0000 0002	PercentDeadband (This type is specified in [UA Part 8]).									
deadbandValue	Double	<p>The <i>Deadband</i> is applied only if</p> <ul style="list-style-type: none"> <li>* the <i>trigger</i> includes value changes and</li> <li>* the <i>deadbandType</i> is set appropriately.</li> </ul> <p>Deadband is generally ignored if the status of the data item changes.</p> <p><b>DeadbandType = AbsoluteDeadband:</b></p> <p>For this type the <i>deadbandValue</i> contains the absolute change in a data value that will cause a <i>Notification</i> to be generated. This parameter applies only to <i>Variables</i> with any integer or floating point data type.</p> <p>An exception that causes a <i>DataChange Notification</i> based on an AbsoluteDeadband is determined as follows:</p> <p><b>Exception if (absolute value of (last cached value - current value) &gt; AbsoluteDeadband)</b></p> <p>The last cached value is defined as the most recent value previously sent to the <i>Notification</i> channel.</p> <p>If the item is an array of values, the entire array is returned if any array element exceeds the AbsoluteDeadband.</p> <p><b>DeadbandType = PercentDeadband:</b></p> <p>This type is specified in [UA Part 8]</p>								

### 8.3.3 EventFilter

#### 8.3.3.1 General

The *EventFilter* provides for the filtering and content selection of *Event Subscriptions*.

If an *Event Notification* conforms to the filter defined by the *where* parameter of the *EventFilter*, then the *Notification* is sent to the *Client*.

Each *Event Notification* will include the field defined by the *select* array parameter of the *EventFilter*. [UA Part 3] describes the *Event* model and the base *EventTypes*. The fields of the base *EventTypes* and their representation in the *AddressSpace* are specified in [UA Part 5].

If an *EventType* does not support a selected field, a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* must be *Bad\_NotSupported*.

If the selected array is empty or one of the *NodeIds* is invalid, the *Server* will return an error. At least one field must be selected.

Table 164 defines the *EventFilter* structure.

**Table 164 – EventFilter structure**

Name	Type	Description
EventFilter	structure	
selectClauses []	NodeId	List of <i>NodeIds</i> for the <i>EventType</i> field to return with each <i>Event</i> in a <i>Notification</i> . At least one <i>EventType</i> field must be selected. If a field is not applicable to an <i>Event</i> being returned a null value will be returned for that field.
whereClause	ContentFilter	Limit the <i>Notifications</i> to those <i>Events</i> that match the criteria defined by this <i>ContentFilter</i> . The <i>ContentFilter</i> structure is described in Clause 7.2.

## 8.4 FilterOperand parameters

### 8.4.1 Overview

The *ContentFilter* structure specified in Clause 7.2 defines a collection of elements that makes up a filter criteria and contains different types of *FilterOperands*. The *FilterOperand* parameter is an extensible parameter. This parameter is defined in Table 165.

**Table 165 –FilterOperand parameterTypeIds**

Symbolic Id	Description
Element	Contains an index into the array of elements. This type is use to build a logic tree of sub-elements by linking the operand of one element to a sub-element.
Literal	Contains a literal value.
Attribute	Contains a <i>NodeId</i> Reference to an <i>Attribute</i> of a <i>Variable</i> or <i>Property</i> . This must be a <i>NodeId</i> from the type system.
Property	Contains a <i>NodeId</i> and a name of a <i>Property</i> of the <i>Node</i> . This must be a <i>NodeId</i> from the type system.

### 8.4.2 ElementOperand

The *ElementOperand* provides the linking to sub-elements within a *ContentFilter*. The link is in the form of an integer that is used to index into the array of elements contained in the *ContentFilter*. An index is considered valid if its value is greater than the element index it is part of and it does not *Reference* a non-existent element. *Clients* must construct filters in this way to avoid circular and invalid *References*. *Servers* should protect against invalid indexes by verifying the index prior to using it.

Table 166 defines the *ElementOperand* type.

**Table 166 – ElementOperand**

Name	Type	Description
ElementOperand	structure	ElementOperand value.
index	UInt32	Index into the element array.

### 8.4.3 LiteralOperand

Table 167 defines the *LiteralOperand* type.

**Table 167 – LiteralOperand**

Name	Type	Description
LiteralOperand	structure	LiteralOperand value.
value	BaseDataType	A literal value.

#### 8.4.4 AttributeOperand

Table 168 defines the *AttributeOperand* type.

**Table 168 – AttributeOperand**

Name	Type	Description
AttributeOperand	structure	Attribute of a <i>Node</i> in the address space.
nodeId	NodeId	<i>NodeId</i> of a <i>Node</i> from the type system.
alias	String	An optional parameter used to identify or refer to an alias. An alias is a symbolic name that can be used to alias this operand and use it in other location in the filter structure.
attributeId	IntegerId	Id of the <i>Attribute</i> . This must be a valid <i>Attribute</i> id. The <i>IntegerId</i> is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA Part 6].
indexRange	NumericRange	This parameter is used to identify a single element of a structure, bit mask or an array, or a single range of indexes for arrays or bit masks. The first element is identified by index 0 (zero). The <i>NumericRange</i> type is defined in Clause 7.14. This parameter is not used if the specified <i>Attribute</i> is not an array, a bit mask or a structure. However, if the specified <i>Attribute</i> is an array, a bit mask or a structure, and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used.

#### 8.4.5 PropertyOperand

Table 169 defines the *PropertyOperand* type.

**Table 169 – PropertyOperand**

Name	Type	Description
PropertyOperand	structure	Property in the address space.
nodeId	NodeId	<i>NodeId</i> of a <i>Node</i> in the type system.
alias	String	An optional parameter used to identify or refer to an alias. An alias is a symbolic name that can be used to alias this operand and use it in other location in the filter structure.
property	QualifiedName	<i>Property</i> . Implicit Reference to the value <i>Attribute</i> of the <i>Property</i>
indexRange	NumericRange	This parameter is used to identify a single element of a structure, bit mask or an array, or a single range of indexes for arrays or bit masks. The first element is identified by index 0 (zero). The <i>NumericRange</i> type is defined in Clause 7.14. This parameter is not used if the specified <i>Attribute</i> is not an array, a bit mask or a structure. However, if the specified <i>Attribute</i> is an array, a bit mask or a structure, and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used.

### 8.5 NotificationData parameters

#### 8.5.1 Overview

The *NotificationMessage* structure used in the *Subscription Service* set allows specifying different types of *NotificationData*. The *NotificationData* parameter is an extensible parameter whose structure depends on the type of *Notification* being sent. This parameter is defined in Table 170. Other types can be defined by additional parts of this multi-part specification or other standards based on OPC UA.

**Table 170 – NotificationData parameterTypeIds**

Symbolic Id	Description
DataChange	<i>Notification data parameter used for data change Notifications.</i>
Event	<i>Notification data parameter used for Event Notifications.</i>

### 8.5.2 DataChangeNotification parameter

Table 171 defines the *NotificationData* parameter used for data change notifications. This structure contains the monitored data items that are to be reported. Monitored data items are reported under two conditions:

- a) If the *MonitoringMode* is set to REPORTING and a change in value or its status (represented by its *StatusCode*) is detected.
- b) If the *MonitoringMode* is set to SAMPLING, the *MonitoredItem* is linked to a triggering item and the triggering item triggers.

See Clause 5.12 for a description of the *MonitoredItem* Service set, and in particular the *MonitoringItemModel* and the *TriggeringModel*.

After creating a *MonitoredItem* the current value or status of the monitored *Attribute* must be queued without applying the filter. If the current value is not available after the first sampling interval the first *Notification* must be queued after getting the initial value or status from the data source.

**Table 171 – DataChangeNotification**

Name	Type	Description
DataChangeNotification	structure	Data change <i>Notification</i> data
monitoredItems []	MonitoredItem Notification	The list of <i>MonitoredItems</i> for which a change has been detected.
clientHandle	IntegerId	<i>Client-supplied handle for the MonitoredItem. The IntegerId type is defined in Clause 7.9</i>
value	DataValue	The <i>StatusCodes</i> , value and timestamp(s) of the monitored <i>Attribute</i> depending on the sampling and queuing configuration. If the <i>StatusCodes</i> indicates an error then the value and timestamp(s) are to be ignored. If not every detected change has been returned since the Server's queue buffer for the <i>MonitoredItem</i> reached its limit and had to purge out data, the <i>Overflow</i> bit in the <i>DataValue InfoBits</i> of the <i>statusCodes</i> is set. <i>DataValue</i> is a common type defined in Clause 7.4.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information. The size and order of this list matches the size and order of the <i>monitoredItem</i> parameter. There is one entry in this list for each <i>Node</i> contained in the <i>monitoredItem</i> parameter. This list is empty if diagnostics information was not requested or is not available for any of the <i>MonitoredItems</i> . <i>DiagnosticInfo</i> is a common type defined in Clause 7.5.

### 8.5.3 EventNotification parameter

Table 172 defines the *NotificationData* parameter used for *Event* notifications.

The *EventNotification* defines a table structure that is used to return *Event* fields to a *Client Subscription*. The structure is in the form of a table consisting of one or more *Events*, each containing an array of one or more fields. The selection and order of the fields returned for each *Event* is identical to the selected parameter of the *EventFilter*.

**Table 172 –EventNotification**

Name	Type	Description
EventNotification	structure	Event Notification data
events []	EventFieldList	The list of <i>Events</i> being delivered
eventFields []	BaseDataType	List of selected <i>Event</i> fields. This will be a one to one match with the fields selected in the <i>EventFilter</i> . If an <i>EventType</i> does not support a selected field, a <i>StatusCodes</i> value will be returned for any <i>Event</i> field that cannot be returned. The value of the <i>StatusCodes</i> must be <i>Bad_NotSupported</i> . Other <i>StatusCodes</i> may indicate other problems such as <i>Bad_UserAccessDenied</i> .

## 8.6 NodeAttributes parameters

### 8.6.1 Overview

The *AddNodes* Service allows specifying the *Attributes* for the *Nodes* to add. The *NodeAttributes* is an extensible parameter whose structure depends on the type of the *Attribute* being added. It identifies the *NodeClass* that defines the structure of the *Attributes* that follow. The *parameterTypeIds* are defined in Table 173.

**Table 173 – NodeAttributes parameterTypeIds**

Symbolic Id	Description
ObjectAttributes	Defines the <i>Attributes</i> for the <i>Object NodeClass</i> .
VariableAttributes	Defines the <i>Attributes</i> for the <i>Variable NodeClass</i> .
MethodAttributes	Defines the <i>Attributes</i> for the <i>Method NodeClass</i> .
ObjectTypeAttributes	Defines the <i>Attributes</i> for the <i>ObjectType NodeClass</i> .
VariableTypeAttributes	Defines the <i>Attributes</i> for the <i>VariableType NodeClass</i> .
ReferenceTypeAttributes	Defines the <i>Attributes</i> for the <i>ReferenceType NodeClass</i> .
DataTypeAttributes	Defines the <i>Attributes</i> for the <i>DataType NodeClass</i> .
ViewAttributes	Defines the <i>Attributes</i> for the <i>View NodeClass</i> .

### 8.6.2 ObjectAttributes parameter

Table 174 defines the *ObjectAttributes* parameter.

**Table 174 – ObjectAttributes**

Name	Type	Description
ObjectAttributes	structure	Defines the <i>Attributes</i> for the <i>Object NodeClass</i>
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
eventNotifier	Byte	See [UA Part 3] for the description of this <i>Attribute</i> .

### 8.6.3 VariableAttributes parameter

Table 175 defines the *VariableAttributes* parameter.

**Table 175 – VariableAttributes**

Name	Type	Description
VariableAttributes	structure	Defines the <i>Attributes</i> for the <i>Variable NodeClass</i>
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
value	Defined by the <i>DataType Attribute</i>	See [UA Part 3] for the description of this <i>Attribute</i> .
dataType	NodeId	See [UA Part 3] for the description of this <i>Attribute</i> .
arraySize	Int32	See [UA Part 3] for the description of this <i>Attribute</i> .
accessLevel	Byte	See [UA Part 3] for the description of this <i>Attribute</i> .
userAccessLevel	Byte	See [UA Part 3] for the description of this <i>Attribute</i> .
minimumSamplingInterval	Int32	See [UA Part 3] for the description of this <i>Attribute</i> .
historizing	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .

#### 8.6.4 MethodAttributes parameter

Table 176 defines the *MethodAttributes* parameter.

**Table 176 – MethodAttributes**

Name	Type	Description
BaseAttributes	structure	Defines the <i>Attributes</i> for the <i>Method NodeClass</i> .
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
executable	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .
userExecutable	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .

#### 8.6.5 ObjectTypeAttributes parameter

Table 177 defines the *ObjectTypeAttributes* parameter.

**Table 177 – ObjectTypeAttributes**

Name	Type	Description
ObjectTypeAttributes	structure	Defines the <i>Attributes</i> for the <i>ObjectType NodeClass</i> .
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
isAbstract	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .

#### 8.6.6 VariableTypeAttributes parameter

Table 178 defines the *VariableTypeAttributes* parameter.

**Table 178 – VariableTypeAttributes**

Name	Type	Description
VariableTypeAttributes	structure	Defines the <i>Attributes</i> for the <i>VariableType NodeClass</i> .
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
value	Defined by the <i>DataType Attribute</i>	See [UA Part 3] for the description of this <i>Attribute</i> .
dataType	NodeID	See [UA Part 3] for the description of this <i>Attribute</i> .
arraySize	Int32	See [UA Part 3] for the description of this <i>Attribute</i> .
isAbstract	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .

#### 8.6.7 ReferenceTypeAttributes parameter

Table 179 defines the *ReferenceTypeAttributes* parameter.

**Table 179 – ReferenceTypeAttributes**

Name	Type	Description
ReferenceTypeAttributes	structure	Defines the <i>Attributes</i> for the <i>ReferenceType NodeClass</i> .
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
isAbstract	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .
symmetric	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .
inverseName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .

### 8.6.8 **DataTypeAttributes** parameter

Table 176 defines the *DataTypeAttributes* parameter.

**Table 180 – DataTypeAttributes**

Name	Type	Description
DataTypeAttributes	structure	Defines the <i>Attributes</i> for the <i>NodeType NodeClass</i>
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
isAbstract	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .

### 8.6.9 **ViewAttributes** parameter

Table 181 defines the *ViewAttributes* parameter.

**Table 181 – ViewAttributes**

Name	Type	Description
ViewAttributes	structure	Defines the <i>Attributes</i> for the <i>View NodeClass</i>
displayName	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
description	LocalizedText	See [UA Part 3] for the description of this <i>Attribute</i> .
containsNoLoops	Boolean	See [UA Part 3] for the description of this <i>Attribute</i> .
eventNotifier	Byte	See [UA Part 3] for the description of this <i>Attribute</i> .

## 8.7 UserIdentityToken parameters

### 8.7.1 Overview

The *UserIdentityToken* structure used in the Server Service Set allows *Clients* to specify the identity of the user they are acting on behalf of. The exact mechanism used to identify users depends on the system configuration. The different types of identity tokens are based on the most common mechanisms that are used in systems today. Table 182 defines the current set of user identity tokens.

**Table 182 – UserIdentityToken parameterTypeIds**

Symbolic Id	Description
UserName	A user identified by user name and password.
X509v3	A user identified by an X509v3 <i>Certificate</i> .
WSS	A user identified by a WS-SecurityToken.

### 8.7.2 UserName identity tokens

The *UserName* identity token is used to pass simple username/password credentials to the Server. The password may be passed in its literal form (in which case the *Message* must be encrypted) or it may be hashed.

The hash algorithm depends on the application; however, UA *Profiles* may define standard algorithms.

Table 183 defines the User Name Identity Token parameter.

**Table 183 – UserName Identity Token**

Name	Type	Description
UserName	structure	UserName value.
userName	String	A string that identifies the user.
password	String	The password for the user (may be hashed)
hashAlgorithm	String	The hash algorithm used. If not specified, the password is passed as plain text.

### 8.7.3 X509v3 identity tokens

The X.509 Identity Token is used to pass an X509v3 *Certificate* which is issued by the user.

Table 184 defines the X509IdentityToken parameter.

**Table 184 – X509 Identity Token**

Name	Type	Description
X509v3	structure	X509v3 value.
CertificateData	ByteString	The X509 <i>Certificate</i> in DER format.

### 8.7.4 WSS identity tokens

The WSS *IdentityToken* is used to pass WS-Security compliant *SecurityTokens* to the Server.

WS-Security defines a number of token profiles that may be used to represent different types of *SecurityTokens*. For example, Kerberos and SAML tokens have WSS token profiles and must be exchanged in UA as XML Security Tokens.

The WSS X509 and UserName tokens should not be exchanged as XML security tokens. UA applications should use the appropriate UA identity tokens to pass the information contained in these types of WSS *SecurityTokens*.

Table 185 defines the WSS Identity Token parameter.

**Table 185 – Issued Identity Token**

Name	Type	Description
WSS	structure	WSS value.
tokenData	XmlElement	The XML representation of the token.

## Appendix A: BNF definitions

### A.1 Overview over BNF

The BNF (Backus-Naur form) used in this Appendix uses `<` and `>` to mark symbols, `[' and `']` to identify optional paths and `|` to identify alternatives. The `(` and `)` symbols are used it indicate sets.

### A.2 BNF of RelativePath

The following BNF describes the syntax of the *RelativePath* parameter used in the TranslateBrowsePathToNodeIds and the QueryFirst Services.

```

<relative-path> ::= <reference-type> <browse-name> [<relative-path>]

<reference-type> ::= '/' | '.' | '<' [ '!' ] <browse-name> '>'

<browse-name> ::= [<namespace-index> ':' ] <name>

<namespace-index> ::= <digit> [<digit>]

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<name> ::= (<name-char> | '&' <reserved-char>) [<name>]

<reserved-char> ::= '/' | '.' | '<' | '>' | ':' | '!' | '&'

<name-char> ::= All valid characters for a String (see [UA Part 3]) excluding reserved-chars.

```

### A.3 BNF of NumericRange

The following BNF describes the syntax of the NumericRange parameter type.

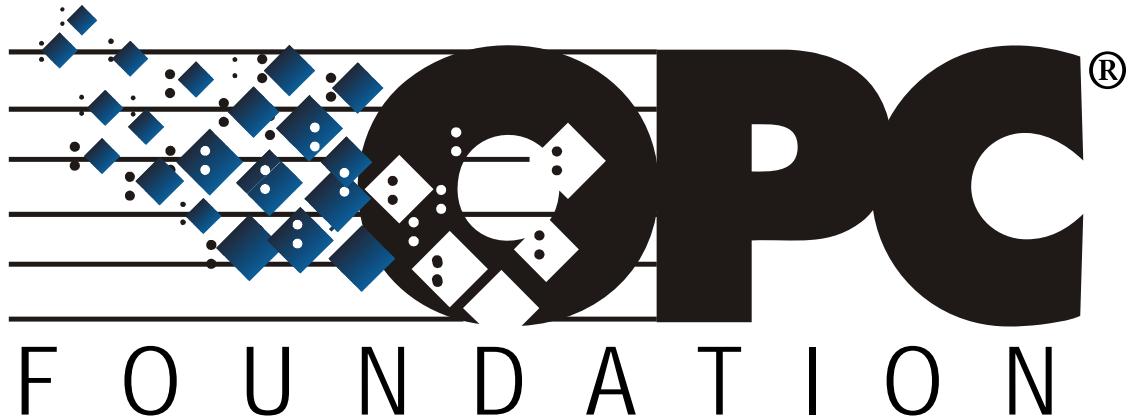
```

<numeric-range> ::= <index> [ ':' <index> ]

<index> ::= <digit> [<digit>]

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

```



# **OPC Unified Architecture**

## **Specification**

### **Part 5: Information Model**

**Version 1.00**

**July 28, 2006**



Specification Type	<u>Industry Standard Specification</u>		
Title:	OPC Unified Architecture	Date:	July 28, 2006
<u>Information Model</u>			
Version:	<u>Release 1.00</u>	Software Source:	<u>MS-Word</u> <u>OPC UA Part 5 - Information Model 1.00 Specification.doc</u>
Author:	<u>OPC Foundation</u>	Status:	<u>Release</u>

## CONTENTS

	Page
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, and conventions.....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 2 terms .....	2
3.3 OPC UA Part 3 terms .....	2
3.4 OPC UA Part 4 terms .....	2
3.5 OPC UA Information Model terms .....	2
3.6 Abbreviations and symbols.....	2
3.7 Conventions for Node descriptions .....	3
4 Nodelds and BrowseNames .....	4
4.1 Nodelds .....	4
4.2 BrowseNames.....	4
5 Common Attributes .....	5
5.1 General .....	5
5.2 Objects .....	5
5.3 Variables .....	5
5.4 VariableTypes.....	5
6 Standard ObjectTypes.....	6
6.1 General .....	6
6.2 BaseObjectType .....	6
6.3 ObjectTypes for the Server Object.....	7
6.3.1 ServerType .....	7
6.3.2 ServerCapabilitiesType.....	8
6.3.3 ServerDiagnosticsType.....	9
6.3.4 SessionsDiagnosticsSummaryType.....	10
6.3.5 SessionDiagnosticsObjectType.....	10
6.3.6 VendorServerInfoType.....	11
6.3.7 ServerRedundancyType .....	11
6.3.8 TransparentRedundancyType .....	11
6.3.9 NonTransparentRedundancyType .....	12
6.4 ObjectTypes used as EventTypes.....	12
6.4.1 General.....	12
6.4.2 BaseEventType .....	13
6.4.3 AuditEventType.....	15
6.4.4 AuditSecurityEventType.....	16
6.4.5 AuditChannelEventType .....	16
6.4.6 AuditOpenSecureChannelEventType .....	16
6.4.7 AuditCloseSecureChannelEventType .....	17
6.4.8 AuditSessionEventType .....	18
6.4.9 AuditCreateSessionEventType.....	18
6.4.10 AuditActivateSessionEventType.....	19
6.4.11 AuditImpersonateUserEventType .....	19
6.4.12 AuditNodeManagementEventType .....	20
6.4.13 AuditAddNodesEventType .....	20

6.4.14	AuditDeleteNodesEventType .....	21
6.4.15	AuditAddReferencesEventType .....	21
6.4.16	AuditDeleteReferencesEventType .....	22
6.4.17	AuditUpdateEventType .....	22
6.4.18	SystemEventType .....	23
6.4.19	DeviceFailureEventType .....	23
6.4.20	BaseModelChangeEventType .....	23
6.4.21	GeneralModelChangeEventType .....	24
6.4.22	PropertyChangeEvent-Type .....	24
6.5	ModellingRuleType .....	24
6.6	FolderType .....	25
6.7	DataTypeEncodingType .....	25
6.8	DataTypeSystemType .....	25
7	Standard VariableTypes .....	26
7.1	General .....	26
7.2	BaseVariableType .....	26
7.3	.PropertyType .....	26
7.4	BaseDataVariableType .....	26
7.5	ServerVendorCapabilityType .....	27
7.6	DataTypeDictionaryType .....	27
7.7	DataTypeDescriptionType .....	28
7.8	ServerStatusType .....	28
7.9	BuildInfoType .....	28
7.10	ServerDiagnosticsSummaryType .....	29
7.11	SamplingRateDiagnosticsArrayType .....	29
7.12	SamplingRateDiagnosticsType .....	30
7.13	SubscriptionDiagnosticsArrayType .....	30
7.14	SubscriptionDiagnosticsType .....	31
7.15	SessionDiagnosticsArrayType .....	31
7.16	SessionDiagnosticsVariableType .....	32
7.17	SessionSecurityDiagnosticsArrayType .....	33
7.18	SessionSecurityDiagnosticsType .....	33
8	Standard Objects and their Variables .....	34
8.1	General .....	34
8.2	Objects used to organise the AddressSpace structure .....	34
8.2.1	Overview .....	34
8.2.2	Root .....	34
8.2.3	Views .....	35
8.2.4	Objects .....	35
8.2.5	Types .....	36
8.2.6	ObjectTypes .....	37
8.2.7	VariableTypes .....	38
8.2.8	ReferenceTypes .....	39
8.2.9	DataTypes .....	39
8.2.10	OPC Binary .....	41
8.2.11	XML Schema .....	41
8.3	Server Object and its containing Objects .....	41
8.3.1	General .....	41
8.3.2	Server Object .....	43

8.4	ModellingRule Objects .....	43
8.4.1	None .....	43
8.4.2	New .....	43
8.4.3	Shared .....	44
9	Standard Methods .....	44
10	Standard ReferenceTypes .....	44
10.1	References .....	44
10.2	HierarchicalReferences .....	44
10.3	NonHierarchicalReferences .....	45
10.4	Aggregates .....	45
10.5	Organizes .....	45
10.6	HasComponent .....	46
10.7	HasOrderedComponent .....	46
10.8	HasProperty .....	46
10.9	HasSubtype .....	46
10.10	HasModellingRule .....	47
10.11	HasTypeDefinition .....	47
10.12	HasEncoding .....	47
10.13	HasDescription .....	47
10.14	HasEventSource .....	48
10.15	HasNotifier .....	48
10.16	GeneratesEvent .....	48
10.17	ExposestArray .....	48
11	Standard DataTypes .....	49
11.1	Overview .....	49
11.2	DataTypes defined in [UA Part 3] .....	50
11.3	DataTypes defined in [UA Part 4] .....	52
11.4	RedundancySupport .....	53
11.5	ServerState .....	53
11.6	RedundantServerDataType .....	54
11.7	SamplingRateDiagnosticsDataType .....	54
11.8	ServerDiagnosticsSummaryDataType .....	55
11.9	ServerStatusDataType .....	55
11.10	SessionDiagnosticsDataType .....	56
11.11	SessionSecurityDiagnosticsDataType .....	57
11.12	ServiceCounterDataType .....	58
11.13	SubscriptionDiagnosticsDataType .....	58
11.14	ChangeStructureDataType .....	59
11.15	PropertyChangeStructureDataType .....	60
Appendix A	: Design decisions when modelling the server information .....	61
A.1	Overview .....	61
A.2	ServerType and Server Object .....	61
A.3	Typed complex Objects beneath the Server Object .....	61
A.4	Properties vs. DataVariables .....	61
A.5	Complex Variables using complex DataTypes .....	62
A.6	Complex Variables having an array .....	62
A.7	Adding ReferenceTypes .....	62
A.8	Redundant information .....	62

A.9 Usage of the BaseDataVariableType .....	63
A.10 Subtyping .....	63
A.11 Extensibility mechanism .....	63

**FIGURES**

Figure 1 – Standard AddressSpace Structure.....	34
Figure 2 – Views Organization .....	35
Figure 3 – Objects Organization .....	36
Figure 4 – ObjectTypes Organization .....	37
Figure 5 – VariableTypes Organization .....	38
Figure 6 – ReferenceType Definitions .....	39
Figure 7 – DataTypes Organization.....	40
Figure 8 – Excerpt of Diagnostic Information of the Server .....	42

**TABLES**

Table 1 – Type Definition Table .....	3
Table 2 – Common Node Attributes .....	5
Table 3 – Common Object Attributes .....	5
Table 4 – Common Variable Attributes .....	5
Table 5 – Common VariableType Attributes .....	5
Table 6 – BaseObjectType Definition .....	6
Table 7 – ServerType Definition.....	7
Table 8 – ServerCapabilitiesType Definition.....	8
Table 9 – ServerDiagnosticsType Definition .....	9
Table 10 – SessionsDiagnosticsSummaryType Definition.....	10
Table 11 – SessionDiagnosticsObjectType Definition .....	10
Table 12 – VendorServerInfoType Definition .....	11
Table 13 – ServerRedundancyType Definition.....	11
Table 14 – TransparentRedundancyType Definition .....	11
Table 15 – NonTransparentRedundancyType Definition .....	12
Table 16 – BaseEventType Definition .....	13
Table 17 – AuditEventType Definition .....	15
Table 18 – AuditSecurityEventType Definition .....	16
Table 19 – AuditChannelEventType Definition.....	16
Table 20 – AuditOpenSecureChannelEventType Definition.....	16
Table 21 – AuditCloseSecureChannelEventType Definition .....	17
Table 22 – AuditSessionEventType Definition .....	18
Table 23 – AuditCreateSessionEventType Definition .....	18
Table 24 – AuditActivateSessionEventType Definition .....	19
Table 25 – AuditImpersonateUserEventType Definition .....	19
Table 26 – AuditNodeManagementEventType Definition.....	20
Table 27 – AuditAddNodesEventType Definition .....	20
Table 28 – AuditDeleteNodesEventType Definition.....	21
Table 29 – AuditAddReferencesEventType Definition .....	21
Table 30 – AuditDeleteReferenceEventType Definition.....	22
Table 31 – AuditUpdateEventType Definition .....	22
Table 32 – SystemEventType Definition.....	23
Table 33 – DeviceFailureEventType Definition .....	23
Table 34 – BaseModelChangeEventType Definition .....	23
Table 35 – GeneralModelChangeEventType Definition .....	24
Table 36 – PropertyChangeEvent Type Definition .....	24
Table 37 – ModellingRuleType Definition .....	24
Table 38 – FolderType Definition .....	25
Table 39 – DataTypeEncodingType Definition .....	25
Table 40 – DataTypeSystemType Definition .....	25
Table 41 – BaseVariableType Definition .....	26
Table 42 – PropertyType Definition.....	26

Table 43 – BaseDataVariableType Definition .....	27
Table 44 – ServerVendorCapabilityType Definition.....	27
Table 45 – DataTypeDictionaryType Definition .....	27
Table 46 – DataTypeDescriptionType Definition.....	28
Table 47 – ServerStatusType Definition .....	28
Table 48 – BuildInfoType Definition .....	28
Table 49 – ServerDiagnosticsSummaryType Definition.....	29
Table 50 – SamplingRateDiagnosticsArrayType Definition.....	29
Table 51 – SamplingRateDiagnosticsType Definition .....	30
Table 52 – SubscriptionDiagnosticsArrayType Definition .....	30
Table 53 – SubscriptionDiagnosticsType Definition .....	31
Table 54 – SessionDiagnosticsArrayType Definition.....	31
Table 55 – SessionDiagnosticsVariableType Definition .....	32
Table 56 – SessionSecurityDiagnosticsArrayType Definition .....	33
Table 57 – SessionSecurityDiagnosticsType Definition.....	33
Table 58 – Root Definition .....	34
Table 59 – Views Definition .....	35
Table 60 – Objects Definition.....	36
Table 61 – Types Definition .....	36
Table 62 – ObjectTypes Definition .....	37
Table 63 – VariableTypes Definition .....	38
Table 64 – ReferenceTypes Definition .....	39
Table 65 – DataTypes Definition .....	40
Table 66 – OPC Binary Definition .....	41
Table 67 – XML Schema Definition .....	41
Table 68 – Server Definition .....	43
Table 69 – None Definition .....	43
Table 70 – New Definition .....	43
Table 71 – Shared Definition .....	44
Table 72 – References ReferenceType .....	44
Table 73 – HierarchicalReferences ReferenceType .....	44
Table 74 – NonHierarchicalReferences ReferenceType.....	45
Table 75 – Aggregates ReferenceType .....	45
Table 76 – Organizes ReferenceType .....	45
Table 77 – HasComponent ReferenceType .....	46
Table 78 – HasOrderedComponent ReferenceType.....	46
Table 79 – HasProperty ReferenceType.....	46
Table 80 – HasSubtype ReferenceType .....	46
Table 81 – HasModellingRule ReferenceType .....	47
Table 82 – HasTypeDefinition ReferenceType.....	47
Table 83 – HasEncoding ReferenceType .....	47
Table 84 – HasDescription ReferenceType .....	47
Table 85 – HasEventSource ReferenceType .....	48

Table 86 – HasNotifier ReferenceType .....	48
Table 87 – GeneratesEvent ReferenceType .....	48
Table 88 – ExposesItsArray ReferenceType.....	49
Table 89 – [UA Part 3] DataType Definitions .....	50
Table 90 – BaseDataType Definition.....	51
Table 91 – Number Definition .....	51
Table 92 – Integer Definition .....	52
Table 93 – BaseDataType Definition.....	52
Table 94 – [UA Part 4] DataType Definitions .....	52
Table 95 – RedundancySupport Values .....	53
Table 96 – RedundancySupport Definition .....	53
Table 97 – ServerState Values .....	53
Table 98 – ServerState Definition .....	54
Table 99 – RedundantServerDataType Structure .....	54
Table 100 – RedundantServerDataType Definition .....	54
Table 101 – SamplingRateDiagnosticsDataType Structure .....	54
Table 102 – SamplingRateDiagnosticsDataType Definition.....	54
Table 103 – ServerDiagnosticsSummaryDataType Structure .....	55
Table 104 – ServerDiagnosticsSummaryDataType Definition.....	55
Table 105 – ServerStatusDataType Structure .....	55
Table 106 – ServerStatusDataType Definition .....	56
Table 107 – SessionDiagnosticsDataType Structure .....	56
Table 108 – SessionDiagnosticsDataType Definition .....	57
Table 109 – SessionSecurityDiagnosticsDataType Structure .....	57
Table 110 – SessionSecurityDiagnosticsDataType Definition.....	58
Table 111 – ServiceCounterDataType Structure.....	58
Table 112 – ServiceCounterDataType Definition .....	58
Table 113 – SubscriptionDiagnosticsDataType Structure.....	58
Table 114 – SubscriptionDiagnosticsDataType Definition .....	59
Table 115 – ChangeStructureDataType Structure .....	59
Table 116 – ChangeStructureDataType Definition .....	59
Table 117 – PropertyChangeStructureDataType Structure.....	60
Table 118 – PropertyChangeStructureDataType Definition .....	60

## OPC FOUNDATION

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these

materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>



## 1 Scope

This specification defines the Information Model of the OPC Unified Architecture. The Information Model describes standardised *Nodes* of a server's *AddressSpace*. These *Nodes* are standardised types as well as standardised instances used for diagnostics or as entry points to server specific *Nodes*. Thus, the Information Model defines the *AddressSpace* of an empty OPC UA server. However, it is not expected that all servers will provide all of these *Nodes*.

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 6] OPC UA Specification: Part 6 – Mapping, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part6/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

## 3 Terms, definitions, and conventions

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Attribute
- 3) Event
- 4) Information Model
- 5) Method
- 6) MonitoredItem
- 7) Node
- 8) NodeClass
- 9) Notification
- 10) Object
- 11) ObjectType
- 12) Profile
- 13) Reference
- 14) ReferenceType
- 15) Service
- 16) Service Set
- 17) Subscription
- 18) Variable

19) View

### **3.2 OPC UA Part 2 terms**

There are no [UA Part 2] terms used in this part.

### **3.3 OPC UA Part 3 terms**

The following terms defined in [UA Part 3] apply.

- 1) DataVariable
- 2) EventType
- 3) Hierarchical Reference
- 4) InstanceDeclaration
- 5) ModellingRule
- 6) Property
- 7) SourceNode
- 8) TargetNode
- 9) TypeDefinitionNode
- 10) VariableType

### **3.4 OPC UA Part 4 terms**

There are no [UA Part 4] terms used in this part.

### **3.5 OPC UA Information Model terms**

There are no additional terms defined in this document.

### **3.6 Abbreviations and symbols**

UA	Unified Architecture
XML	Extensible Markup Language

### 3.7 Conventions for Node descriptions

*Node* definitions are specified using tables. Blank lines may be inserted for readability.

*Attributes* are defined by providing the *Attribute* name and a value, or a description of the value.

*References* are defined by providing the *ReferenceType* name, the *BrowseName* of the *TargetNode* and its *NodeClass*.

- If the *TargetNode* is a component of the *Node* being defined in the table the *Attributes* of the composed *Node* are defined in the same row of the table.
- The *DataType* is only specified for *Variables*; “[<number>]” indicates an array. For all arrays the *ArraySize* is set as identified by <number>. If no <number> is set, the *ArraySize* is set to 0, indicating an unknown size.
- The *TypeDefinition* is specified for *Objects* and *Variables*.
- The *TypeDefinition* column specifies a symbolic name for a *NodeId*, i.e. the specified *Node* points with a *HasTypeDefinition* *Reference* to the corresponding *Node*.
- The *ModellingRule* of the referenced component is provided by specifying the symbolic name of the rule in the *ModellingRule* column. In the *AddressSpace*, the *Node* must use a *HasModellingRule* *Reference* to point to the corresponding *ModellingRule Object*.

If the *NodeId* of a *DataType* must be provided, the symbolic name of the *Node* representing the *DataType* is used.

*Nodes* of all other *NodeClasses* cannot be defined in the same table; therefore only the used *ReferenceType*, their *NodeClass* and their *BrowseName* are specified. A reference to another Clause of this document points to their definition.

Table 1 illustrates the table. If no components are provided, the *DataType*, *TypeDefinition* and *ModellingRule* columns may be omitted and only a *Comment* column is introduced to point to the *Node* definition.

**Table 1 – Type Definition Table**

Attribute	Value				
Attribute name	Attribute value. If it is an optional Attribute that is not set “--” will be used.				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
ReferenceType name	NodeClass of the TargetNode.	BrowseName of the target Node. If the Reference is to be instantiated by the server, then the value of the target Node's BrowseName is “--”.	Attributes of the referenced Node, only applicable for Variables and Objects.		Referenced ModellingRule of the referenced Object.
Notes –	1) Notes referencing footnotes of the table content.				

Components of *Nodes* can be complex, i.e. containing components by themselves. The *TypeDefinition*, *NodeClass*, *DataType* and *ModellingRule* can be derived from the type definitions, and the symbolic name can be created as defined in Clause 4.1. Therefore those containing components are not explicitly specified; they are implicitly specified by the type definitions.

## 4 NodeIds and BrowseNames

### 4.1 NodeIds

The *NodeIds* of all *Nodes* described in this document are only symbolic names. [UA Part 6] defines the actual *NodeIds*.

The symbolic name of each *Node* defined in this document is its *BrowseName*, or, when it is part of another *Node*, the *BrowseName* of the other *Node*, a “.”, and the *BrowseName* of itself. In this case “part of” means that the whole has a *HasProperty* or *HasComponent Reference* to its part. Since all *Nodes* not being part of another *Node* have a unique name in this document, the symbolic name is unique. For example, the *ServerType* defined in Clause 6.3.1 has the symbolic name “ServerType”. One of its *InstanceDeclarations* would be identified as “ServerType.ServerCapabilities”. Since this *Object* is complex, another *InstanceDeclaration* of the *ServerType* is “ServerType.ServerCapabilities.MinSupportedSampleRate”. The *Server Object* defined in Clause 8.3.2 is based on the *ServerType* and has the symbolic name “Server”. Therefore, the instance based on the *InstanceDeclaration* described above has the symbolic name “ServerType.ServerCapabilities.MinSupportedSampleRate”.

The *NamespaceIndex* for all *NodeIds* defined in this specification is 0. The namespace for this *NamespaceIndex* is specified in [UA Part 3].

### 4.2 BrowseNames

The text part of the *BrowseNames* for all *Nodes* defined in this part is specified in the tables defining the *Nodes*. The *NamespaceIndex* for all *BrowseNames* defined in this part is 0.

## 5 Common Attributes

### 5.1 General

For all *Nodes* specified in this part, the *Attributes* named in Table 2 must be set as specified in the table.

**Table 2 – Common Node Attributes**

Attribute	Value
DisplayName	The <i>DisplayName</i> is a <i>LocalizedText</i> . Each server must provide the <i>DisplayName</i> identical to the <i>BrowseName</i> of the <i>Node</i> for the LocaleId “en”. Whether the server provides translated names for other LocaleIds is vendor specific.
Description	Optionally a vendor specific description is provided
NodeClass	Must reflect the <i>NodeClass</i> of the <i>Node</i>
NodeId	The <i>NodeId</i> is described by <i>BrowseNames</i> as defined in Clause 4.1 and defined in [UA Part 6].

### 5.2 Objects

For all *Objects* specified in this part, the *Attributes* named in Table 3 must be set as specified in the table.

**Table 3 – Common Object Attributes**

Attribute	Value
EventNotifier	Whether the <i>Node</i> can be used to subscribe to <i>Events</i> or not is vendor specific

### 5.3 Variables

For all *Variables* specified in this part, the *Attributes* named in Table 4 must be set as specified in the table.

**Table 4 – Common Variable Attributes**

Attribute	Value
MinimumSamplingInterval	Optionally, a vendor-specific minimum sampling interval is provided
AccessLevel	The access level for <i>Variables</i> used for type definitions is vendor-specific, for all other <i>Variables</i> defined in this part, the access level must allow a current read; other settings are vendor specific.
UserAccessLevel	The value for the <i>UserAccessLevel Attribute</i> is vendor-specific. It is assumed that all <i>Variables</i> can be accessed by at least one user.
Value	For <i>Variables</i> used as <i>InstanceDeclarations</i> , the value is vendor-specific; otherwise it must represent the value described in the text.

### 5.4 VariableTypes

For all *VariableTypes* specified in this part, the *Attributes* named in Table 5 must be set as specified in the table.

**Table 5 – Common VariableType Attributes**

Attributes	Value
Value	Optionally a vendor-specific default value can be provided

## 6 Standard ObjectTypes

### 6.1 General

Typically, the components of an *ObjectType* are fixed and can be extended by subtyping. However, since each *Object* of an *ObjectType* can be extended with additional components, UA allows extending the standard *ObjectTypes* defined in this document with additional components. Thereby, it is possible to express the additional information in the type definition that would already be contained in each *Object*. Some *ObjectTypes* already provide entry points for server specific extensions. However, it is not allowed to restrict the components of the standard *ObjectTypes* defined in this Part. An example of extending the *ObjectTypes* is putting the standard *Property NodeVersion* defined in [UA Part 3] into the *BaseObjectType*, stating that each *Object* of the server will provide a *NodeVersion*.

### 6.2 BaseObjectType

The *BaseObjectType* is used as type definition whenever there is an *Object* having no more concrete type definition available. Servers should avoid using this *ObjectType* and use a more specific type, if possible. This *ObjectType* is the base *ObjectType* and all other *ObjectTypes* must either directly or indirectly inherit from it. However, it may not be possible for servers to provide all *HasSubtype References* from this *ObjectType* to its subtypes, and therefore it is not required to provide this information.

There are no *References* except for *HasSubtype References* specified for this *ObjectType*. It is formally defined in Table 6.

**Table 6 – BaseObjectType Definition**

Attribute	Value				
BrowseName	BaseObjectType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasSubtype	ObjectType	ServerType	Defined in Clause 6.3.1		
HasSubtype	ObjectType	ServerCapabilitiesType	Defined in Clause 6.3.2		
HasSubtype	ObjectType	ServerDiagnosticsType	Defined in Clause 6.3.3		
HasSubtype	ObjectType	SessionsDiagnosticsSummaryType	Defined in Clause 6.3.4		
HasSubtype	ObjectType	SessionDiagnosticsObjectType	Defined in Clause 6.3.5		
HasSubtype	ObjectType	VendorServerInfoType	Defined in Clause 6.3.6		
HasSubtype	ObjectType	ServerRedundancyType	Defined in Clause 6.3.7		
HasSubtype	ObjectType	BaseEventType	Defined in Clause 6.4.2		
HasSubtype	ObjectType	ModellingRuleType	Defined in Clause 6.5		
HasSubtype	ObjectType	FolderType	Defined in Clause 6.6		
HasSubtype	ObjectType	DataTypeEncodingType	Defined in Clause 6.7		
HasSubtype	ObjectType	DataTypeSystemType	Defined in Clause 6.8		

### 6.3 ObjectTypes for the Server Object

#### 6.3.1 ServerType

This *ObjectType* defines the capabilities supported by the UA server. It is formally defined in Table 7.

**Table 7 – ServerType Definition**

Attribute	Value							
BrowseName	ServerType							
IsAbstract	False							
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule			
Subtype of the BaseObjectType defined in Clause 6.2								
HasProperty	Variable	ServerArray	String[]	.PropertyType	New			
HasProperty	Variable	NamespaceArray	String[]	.PropertyType	New			
HasComponent	Variable	ServerStatus <sup>1</sup>	ServerStatusDataType	ServerStatusType	New			
HasProperty	Variable	ServiceLevel	Byte	.PropertyType	New			
HasComponent	Object	ServerCapabilities <sup>1</sup>	-	ServerCapabilitiesType	New			
HasComponent	Object	ServerDiagnostics <sup>1</sup>	-	ServerDiagnosticsType	New			
HasComponent	Object	VendorServerInfo	-	VendorServerInfoType	New			
HasComponent	Object	ServerRedundancy <sup>1</sup>	-	ServerRedundancyType	New			
GeneratesEvent	ObjectType	AuditEventType	Defined in Clause 6.4.3					
Notes –								
1) Containing <i>Objects</i> and <i>Variables</i> of these <i>Objects</i> and <i>Variables</i> are defined by their <i>BrowseName</i> defined in the corresponding <i>TypeDefinitionNode</i> . The <i>NodeId</i> is defined by the composed symbolic name described in Clause 4.1.								

The *ServerArray Variable* defines an array of server URIs. This *Variable* is also referred to as the *server table*. Each URI in this array represents a globally-unique logical name for a server within the scope of the network in which it is installed. Each OPC UA server instance has a single URI that is used in the *server table* of other OPC UA servers. Index 0 is reserved for the URI of the local server. Values above 0 are used to identify remote servers and are specific to a server. [UA Part 6] describes discovery mechanism that can be used to resolve URIs into URLs.

The indexes into this table are referred to as *server indexes* or *server names*. They are used in OPC UA Services to identify *TargetNodes* of *References* that reside in remote servers. Clients may read the entire table or they may read individual entries in the table. The server must not modify or delete entries of this table while any client has an open session to the server, because clients may cache this table. A server may add entries to the table even if clients are connected to the server.

The *NamespaceArray Variable* defines an array of namespace URIs. This *Variable* is also referred as *namespace table*. The indexes into this table are referred to as *NamespaceIndexes*. *NamespaceIndexes* are used in *Nodelds* in OPC UA Services, rather than the longer namespace URI. Index 0 is reserved for the OPC UA namespace, and index 1 is reserved for the local server. Clients may read the entire table or they may read individual entries in the table. The server must not modify or delete entries of this table while any client has an open session to the server, because clients may cache this table. A server may add entries to the table even if clients are connected to the server. It is recommended that servers not change the indexes of this table but only add entries, because the client may cache *Nodelds* using the indexes. Nevertheless, it may not always be possible for servers to avoid changing indexes in this table. Clients that cache *NamespaceIndexes* of *Nodelds* should always check when starting a session to verify that the cached *NamespaceIndexes* have not changed.

The *ServerStatus Variable* contains elements that describe the status of the server. See Clause 11.9 for a description of its elements.

The *ServiceLevel Variable* describes the ability of the server to provide its data to the client. The value range is from 0 to 255, where 0 indicates the worst and 255 indicates the best. The concrete values are vendor-specific. The intent is to provide the clients an indication of availability among redundant servers.

The *ServerCapabilities Object* defines the capabilities supported by the UA server. See Clause 6.3.2 for its description.

The *ServerDiagnostics Object* defines diagnostic information about the UA server. See Clause 6.3.3 for its description

The *VendorServerInfo Object* represents the browse entry point for vendor-defined server information. This Object is required to be present even if there are no vendor-defined Objects beneath it. See Clause 6.3.6 for its description.

The *ServerRedundancy Object* describes the redundancy capabilities provided by the server. This Object is required even if the server does not provide any redundancy support. If the server supports redundancy, then a subtype of *ServerRedundancyType* is used to describe its capabilities. Otherwise, it provides an Object of type *ServerRedundancyType* with an empty array of *RedundancySupportArray*. See Clause 6.3.7 for the description of *ServerRedundancyType*.

### 6.3.2 ServerCapabilitiesType

This *ObjectType* defines the capabilities supported by the UA server. It is formally defined in Table 8.

**Table 8 – ServerCapabilitiesType Definition**

Attribute	Value				
BrowseName	<i>ServerCapabilitiesType</i>				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the <i>BaseObjectType</i> defined in Clause 6.2					
HasProperty	Variable	ServerProfileArray	String[]	.PropertyType	New
HasProperty	Variable	IdTypeArray	IdType[]	.PropertyType	New
HasProperty	Variable	LocaleIdArray	LocaleId[]	PropertyParams	New
HasProperty	Variable	MinSupportedSampleRate	UInt32	PropertyParams	New
HasProperty	Variable	MaxParallelContinuationPointsPerSession	UInt16	PropertyParams	New
HasComponent	Variable	Vendor specific Variables of a subtype of the <i>ServerVendorCapabilityType</i> defined in Clause 7.5			New

The *ServerProfileArray Variable* defines the conformance profile of the server. See [UA Part 7] for the definitions of server profiles.

The *IdTypeArray Variable* is an array of IdTypes that are supported by the server. This is how the different types of *NodeIds* supported by the server are defined. IdTypes are defined in [UA Part 3].

The *LocaleIdArray Variable* is an array of LocaleIds that are known to be supported by the server. The server may not be aware of all LocaleIds that it supports because it may provide access to underlying servers, systems or devices that do not report the LocaleIds that they support.

The *MinSupportedSampleRate Variable* defines the minimum supported sample rate, including 0, that is supported by the server.

The *MaxParallelContinuationPointsPerSession Variable* is an integer specifying the maximum number of parallel continuation points of the *Browse Service* that the server can support per session. The value specifies the maximum the server can support under normal circumstances, so there is no guarantee the server can always support the maximum. The client should not open more *Browse* calls with open continuation points than exposed in this *Variable*. The value 0 indicates that the server does not restrict the number of parallel continuation points the client should use.

The remaining components of the *ServerCapabilitiesType* define the server-specific capabilities of the server. Each is defined using a *HasComponent Reference* whose target is an instance of a vendor-defined subclass of the abstract *ServerVendorCapabilityType* (see Clause 7.5). Each

subtype of this type defines a specific server capability. The *NodeIds* for these *Variables* and their *VariableTypes* are server-defined.

### 6.3.3 ServerDiagnosticsType

This *ObjectType* defines diagnostic information about the UA server. This *ObjectType* is formally defined in Table 9.

**Table 9 – ServerDiagnosticsType Definition**

Attribute	Value			
BrowseName	ServerDiagnosticsType			
IsAbstract	False			
References	NodeClass	BrowseName	DataType / TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2				
HasComponent	Variable	ServerDiagnosticsSummary	ServerDiagnosticsSummaryDataType ServerDiagnosticsSummaryType	New
HasComponent	Variable	SamplingRateDiagnosticsArray	SamplingRateDiagnosticsDataType[] SamplingRateDiagnosticsArrayType	New
HasComponent	Variable	SubscriptionDiagnosticsArray	SubscriptionDiagnosticsDataType[] SubscriptionDiagnosticsArrayType	New
HasComponent	Object	SessionsDiagnosticsSummary	-- SessionsDiagnosticsSummaryType	New
HasProperty	Variable	EnabledFlag	Boolean .PropertyType	New

The *ServerDiagnosticSummary Variable* contains diagnostic summary information for the server, as defined in Clause 11.8.

The *SamplingRateArray Variable* is an array of diagnostic information per sampling rate as defined in Clause 11.7. There is one entry for each sampling rate currently used by the server. Its *TypeDefinitionNode* is the *VariableType SamplingRateDiagnosticsArrayType*, providing a *Variable* for each entry in the array, as defined in Clause 7.11.

The *SubscriptionArray Variable* is an array of Subscription diagnostic information per subscription, as defined in Clause 11.13. There is one entry for each Notification channel actually established in the server. Its *TypeDefinitionNode* is the *VariableType SubscriptionDiagnosticsArrayType*, providing a *Variable* for each entry in the array as defined in Clause 7.13. Because those *Variables* are also used as *Variables* referenced by other *Variables* they must have the *ModellingRule Shared*.

The *SessionsDiagnostics Object* contains diagnostic information per session, as defined in Clause 6.3.4.

The *EnabledFlag Variable* identifies whether or not diagnostic information is collected by the server. It can also be used by a client to enable or disable the collection of diagnostic information of the server. The following settings of the Boolean value apply: TRUE indicates that the server collects diagnostic information, and setting the value to TRUE leads to resetting and enabling the collection. FALSE indicates that no statistic information is collected, and setting the value to FALSE disables the collection without resetting the statistic values.

### 6.3.4 SessionsDiagnosticsSummaryType

This *ObjectType* defines diagnostic information about the sessions of the UA server. This *ObjectType* is formally defined in Table 10.

**Table 10 – SessionsDiagnosticsSummaryType Definition**

Attribute	Value			
BrowseName	SessionsDiagnosticsSummaryType			
IsAbstract	False			
References	NodeClass	BrowseName	DataType / TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2				
HasComponent	Variable	SessionDiagnosticsArray	SessionDiagnosticsDataType[] SessionDiagnosticsArrayType	New
HasComponent	Variable	SessionSecurityDiagnosticsArray	SessionSecurityDiagnosticsDataType[] SessionSecurityDiagnosticsArrayType	New
HasComponent	Object	For each session of the server one Object has to be provided	-- SessionDiagnosticsObjectType	--

The SessionDiagnosticsArray *Variable* provides an array with an entry for each session in the server having general diagnostic information about a session.

The SessionSecurityDiagnosticsArray *Variable* provides an array with an entry for each active session in the server having security-related diagnostic information about a session. Since this information is security-related, it should not be made accessible to all users, but only to authorised users.

For each session of the server, this *Object* also provides an *Object* representing the session. It has the ClientName of the session as *BrowseName* and is of the *ObjectType* SessionDiagnosticsObjectType, as defined in Clause 6.3.5. However, to identify the *Object* representing the session the client runs on, a special *NodeId* is assigned, as defined in Clause 8.3.2.

### 6.3.5 SessionDiagnosticsObjectType

This *ObjectType* defines diagnostic information about a session of the UA server. This *ObjectType* is formally defined in Table 11.

**Table 11 – SessionDiagnosticsObjectType Definition**

Attribute	Value			
BrowseName	SessionDiagnosticsObjectType			
IsAbstract	False			
References	NodeClass	BrowseName	DataType / TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2				
HasComponent	Variable	SessionDiagnostics	SessionDiagnosticsDataType SessionDiagnosticsVariableType	New
HasComponent	Variable	SessionSecurityDiagnostics	SessionSecurityDiagnosticsDataType SessionSecurityDiagnosticsType	New
HasComponent	Variable	SubscriptionDiagnosticsArray	SubscriptionDiagnosticsDataType[] SubscriptionDiagnosticsArrayType	New
GeneratesEvent	ObjectType	AuditSessionEventType	Defined in Clause 6.4.8	

The SessionDiagnostics *Variable* contains general diagnostic information about the session; the SessionSecurityDiagnostics *Variable* contains security-related diagnostic information. Because the information of the second *Variable* is security-related, it should not be made accessible to all users, but only to authorised users.

The SubscriptionDiagnosticsArray *Variable* is an array of Subscription diagnostic information per opened subscription, as defined in Clause 11.13. Its *TypeDefinitionNode* is the *VariableType*

`SubscriptionDiagnosticsArrayType` providing a *Variable* for each entry in the array, as defined in Clause 7.13.

### 6.3.6 VendorServerInfoType

This *ObjectType* defines a placeholder *Object* for vendor-specific information about the UA server. This *ObjectType* defines an empty *ObjectType* that has no components. It must be subtyped by vendors to define their vendor-specific information. This *ObjectType* is formally defined in Table 12.

**Table 12 – VendorServerInfoType Definition**

Attribute	Value										
BrowseName	VendorServerInfoType										
IsAbstract	False										
References	<table border="1"> <thead> <tr> <th>NodeClass</th> <th>BrowseName</th> <th>DataType</th> <th>TypeDefinition</th> <th>Modelling Rule</th> </tr> </thead> <tbody> <tr> <td>Subtype of the BaseObjectType defined in Clause 6.2</td><td></td><td></td><td></td><td></td></tr> </tbody> </table>	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule	Subtype of the BaseObjectType defined in Clause 6.2				
NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule							
Subtype of the BaseObjectType defined in Clause 6.2											
Subtype of the BaseObjectType defined in Clause 6.2											

### 6.3.7 ServerRedundancyType

This *ObjectType* defines the redundancy capabilities supported by the UA server. It is formally defined in Table 13.

**Table 13 – ServerRedundancyType Definition**

Attribute	Value																									
BrowseName	ServerRedundancyType																									
IsAbstract	False																									
References	<table border="1"> <thead> <tr> <th>NodeClass</th> <th>BrowseName</th> <th>DataType</th> <th>TypeDefinition</th> <th>Modelling Rule</th> </tr> </thead> <tbody> <tr> <td>Subtype of the BaseObjectType defined in Clause 6.2</td><td></td><td></td><td></td><td></td></tr> <tr> <td>HasProperty</td><td>Variable</td><td>RedundancySupport</td><td>RedundancySupport</td><td>.PropertyType</td></tr> <tr> <td>HasSubtype</td><td>ObjectType</td><td>TransparentRedundancyType</td><td>Defined in Clause 6.3.8</td><td></td></tr> <tr> <td>HasSubtype</td><td>ObjectType</td><td>NonTransparentRedundancyType</td><td>Defined in Clause 6.3.9</td><td></td></tr> </tbody> </table>	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule	Subtype of the BaseObjectType defined in Clause 6.2					HasProperty	Variable	RedundancySupport	RedundancySupport	.PropertyType	HasSubtype	ObjectType	TransparentRedundancyType	Defined in Clause 6.3.8		HasSubtype	ObjectType	NonTransparentRedundancyType	Defined in Clause 6.3.9	
NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule																						
Subtype of the BaseObjectType defined in Clause 6.2																										
HasProperty	Variable	RedundancySupport	RedundancySupport	.PropertyType																						
HasSubtype	ObjectType	TransparentRedundancyType	Defined in Clause 6.3.8																							
HasSubtype	ObjectType	NonTransparentRedundancyType	Defined in Clause 6.3.9																							
Subtype of the BaseObjectType defined in Clause 6.2																										
HasProperty	Variable	RedundancySupport	RedundancySupport	.PropertyType	New																					
HasSubtype	ObjectType	TransparentRedundancyType	Defined in Clause 6.3.8																							
HasSubtype	ObjectType	NonTransparentRedundancyType	Defined in Clause 6.3.9																							

The *RedundancySupport Variable* indicates what redundancy is supported by the server. Its values are defined in Clause 11.4.

### 6.3.8 TransparentRedundancyType

This *ObjectType* is a subtype of *ServerRedundancyType* and is used to identify the capabilities of the UA server for server-controlled redundancy with a transparent switchover for the client. It is formally defined in Table 14.

**Table 14 – TransparentRedundancyType Definition**

Attribute	Value																									
BrowseName	TransparentRedundancyType																									
IsAbstract	False																									
References	<table border="1"> <thead> <tr> <th>Node Class</th> <th>BrowseName</th> <th>DataType</th> <th>TypeDefinition</th> <th>Modelling Rule</th> </tr> </thead> <tbody> <tr> <td colspan="2">Inherit the Properties of the ServerRedundancyType defined in Clause 6.3.7, i.e. it has HasProperty References to the same Nodes.</td><td></td><td></td><td></td></tr> <tr> <td>HasProperty</td><td>Variable</td><td>CurrentServerId</td><td>String</td><td>.PropertyType</td></tr> <tr> <td>HasProperty</td><td>Variable</td><td>RedundantServerArray</td><td>RedundantServerDataType[]</td><td>.PropertyType</td></tr> <tr> <td>HasProperty</td><td>Variable</td><td></td><td></td><td>New</td></tr> </tbody> </table>	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule	Inherit the Properties of the ServerRedundancyType defined in Clause 6.3.7, i.e. it has HasProperty References to the same Nodes.					HasProperty	Variable	CurrentServerId	String	.PropertyType	HasProperty	Variable	RedundantServerArray	RedundantServerDataType[]	.PropertyType	HasProperty	Variable			New
Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule																						
Inherit the Properties of the ServerRedundancyType defined in Clause 6.3.7, i.e. it has HasProperty References to the same Nodes.																										
HasProperty	Variable	CurrentServerId	String	.PropertyType																						
HasProperty	Variable	RedundantServerArray	RedundantServerDataType[]	.PropertyType																						
HasProperty	Variable			New																						
Inherit the Properties of the ServerRedundancyType defined in Clause 6.3.7, i.e. it has HasProperty References to the same Nodes.																										
HasProperty	Variable	CurrentServerId	String	PropertyParams	New																					
HasProperty	Variable	RedundantServerArray	RedundantServerDataType[]	PropertyParams	New																					

The *RedundancySupport Variable* is inherited from the *ServerRedundancyType*.

Although, in a transparent switchover scenario, all redundant servers serve under the same URI to the client, it may be required to track the exact data source on the client. Therefore, the *CurrentServerId Variable* contains an identifier of the currently-used server in the redundant set. This server is valid only inside a session; if a client opens several sessions, different servers of the redundant set of servers may serve it in different sessions. The value of the *CurrentServerId* may

change due to failover or load balancing, so a client that needs to track its data source must subscribe to this *Variable*.

As diagnostic information, the *RedundantServerArray* contains an array of available servers in the redundant set, including their service levels (see Clause 11.6). This array may change during a session.

### 6.3.9 NonTransparentRedundancyType

This *ObjectType* is a subtype of *ServerRedundancyType* and is used to identify the capabilities of the UA server for non-transparent redundancy. It is formally defined in Table 15.

**Table 15 – NonTransparentRedundancyType Definition**

Attribute	Value				
BrowseName	NonTransparentRedundancyType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>ServerRedundancyType</i> defined in Clause 6.3.7, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	ServerURIArray	String[]	.PropertyType	New

The *ServerURIArray Variable* is an array with the URI of all redundant servers of the UA server. See [UA Part 1] for the definition of redundancy in UA. Since, in a non-transparent redundancy environment, the client is responsible to subscribe to the redundant servers, it may or may not open a session to one or more redundant servers of this array.

The redundancy support provided by the server is defined in the *RedundancySupport* (defined in the supertype). The client is allowed to access the redundant sever only as described there, however, "hot" switchover implies the support of "warm" switchover and "warm" switchover implies the support of "cold" switchover.

If the server supports only a "cold" switchover, the *ServiceLevel Variable* of the *Server Object* should be considered to identify the primary server. In this scenario, only the primary server may be able to access the underlying system, because the underlying system may support access only from a single server. In this case, all other servers will be identified with a *ServiceLevel* of zero.

## 6.4 ObjectTypes used as EventTypes

### 6.4.1 General

OPC UA defines standard *EventTypes*. They are represented in the *AddressSpace* as *ObjectTypes*. The *EventTypes* are already defined in [UA Part 3]. The following subsections specify their representation in the *AddressSpace*.

All fields that can be part of an *Event Notification* are defined as *Variables* of the *EventType*. These *Variables* have their *ModellingRule* defined as *New*. Typically, *Properties* are used for this purpose. It is also allowed to use *DataVariables* related to the *EventType* with a *HasComponent Reference* to model complex *Variables*. Thus a client can subscribe to fields that are only a part of a complex *Variable* by choosing the contained *DataVariable* of the complex *Variable*. To permit the logical grouping of *Variables*, the *EventType* may contain *Objects* that group *Variables*. Those *Variables* can also be used as fields of the *Event* to subscribe to. To have unambiguous names for the fields of an *Event* the client should consider using the path instead of only the *DisplayName* of such *Variables*.

The fields that are returned to a client must be specified as part of the *Event filter*. All servers supporting a given *EventType* must provide all listed *Variables*.

### 6.4.2 BaseEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 16.

**Table 16 – BaseEventType Definition**

Attribute	Value				
BrowseName	BaseEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the <i>BaseObjectType</i> defined in Clause 6.2					
HasSubtype	ObjectType	AuditEventType	Defined in Clause 6.4.3		
HasSubtype	ObjectType	SystemEventType	Defined in Clause 6.4.18		
HasSubtype	ObjectType	BaseModelChangeEvent	Defined in Clause 6.4.20		
HasSubtype	ObjectType	PropertyChangeEventType	Defined in Clause 6.4.22		
HasProperty	Variable	EventId	ByteString	.PropertyType	New
HasProperty	Variable	EventType	NodeId	.PropertyType	New
HasProperty	Variable	SourceNode	NodeId	PropertyParams	New
HasProperty	Variable	SourceName	String	PropertyParams	New
HasProperty	Variable	Time	UtcTime	PropertyParams	New
HasProperty	Variable	ReceiveTime	UtcTime	PropertyParams	New
HasProperty	Variable	Message	LocalizedText	PropertyParams	New
HasProperty	Variable	Severity	UInt16	PropertyParams	New

The *EventId* is generated by the server to uniquely identify a particular *Event Notification*. The server is responsible to ensure that each *Event* has its unique *EventId*. It may do this, for example, by putting GUIDs into the *ByteString*. Clients can use the *EventId* to assist in minimizing or eliminating gaps and overlaps that may occur during a redundancy failover.

The *EventType* describes the specific type of *Event*.

The *SourceNode* identifies the *Node* that the *Event* originated from. If the *Event* is not specific to a *Node* the *NodeId* is set to null. Some subtypes of this *BaseEventType* may define additional rules for *SourceNode*.

The *SourceName* provides a description of the source of the *Event*. This could be the *DisplayName* of the *Event* source – if the *Event* is specific to a *Node* – or some server-specific notation.

The *Time* provides the time the *Event* occurred. This value is set as close to the event generator as possible. It often comes from the underlying system or device. Once set, intermediate UA Servers must not alter the value.

The *ReceiveTime* provides the time the UA Server received the *Event* from the underlying device of another Server. *ReceiveTime* is analogous to *ServerTimestamp* defined in [UA Part 4], i.e. in the case where the OPC UA Server gets an *Event* from another OPC UA Server, each Server applies its own *ReceiveTime*. That implies that a *Client* may get the same *Event* – having the same *EventId* – from different Servers having different values of the *ReceiveTime*.

The *Message Variable* provides a human-readable and localizable text description of the *Event*. The server may return any appropriate text to describe the *Event*. A null string is not a valid value; if the server does not have a description, it must return the string part of the *BrowseName* of the *Node* associated with the *Event*.

The *Severity* is an indication of the urgency of the *Event*. This is also commonly called “priority”. Values will range from 1 to 1000, with 1 being the lowest severity and 1000 being the highest. Typically, a severity of 1 would indicate an *Event* which is informational in nature, while a value of 1000 would indicate an *Event* of catastrophic nature, which could potentially result in severe financial loss or loss of life.

It is expected that very few server implementations will support 1000 distinct severity levels. Therefore, server developers are responsible for distributing their severity levels across the 1 – 1000 range in such a manner that clients can assume a linear distribution. For example, a client wishing to present five severity levels to a user should be able to do the following mapping:

Client Severity	OPC Severity
HIGH	801 – 1000
MEDIUM HIGH	601 – 800
MEDIUM	401 – 600
MEDIUM LOW	201 – 400
LOW	1 – 200

In many cases a strict linear mapping of underlying source severities to the OPC Severity range is not appropriate. The server developer will instead intelligently map the underlying source severities to the 1 – 1000 OPC Severity range in some other fashion. In particular, it is recommended that server developers map *Events* of high urgency into the OPC severity range of 667 – 1000, *Events* of medium urgency into the OPC severity range of 334 – 666 and *Events* of low urgency into OPC severities of 1 – 333.

For example, if a source supports 16 severity levels that are clustered such that severities 0 – 2 are considered to be LOW, 3 – 7 are MEDIUM and 8 – 15 are HIGH, then an appropriate mapping might be as follows:

OPC Range	Source Severity	OPC Severity
HIGH (667 – 1000)	15	1000
	14	955
	13	910
	12	865
	11	820
	10	775
	9	730
	8	685
MEDIUM (334 – 666)	7	650
	6	575
	5	500
	4	425
	3	350
LOW (1 – 333)	2	300
	1	150
	0	1

Some servers may not support any *Events* which are catastrophic in nature, so they may choose to map all of their severities into a subset of the 1 – 1000 range (for example, 1 – 666). Other servers may not support any *Events* which are merely informational, so they may choose to map all of their severities into a different subset of the 1 – 1000 range (for example, 334 – 1000).

The purpose of this approach is to allow clients to use severity values from multiple servers from different vendors in a consistent manner.

### 6.4.3 AuditEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 17.

**Table 17 – AuditEventType Definition**

Attribute	Value				
BrowseName	AuditEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>BaseEventType</i> defined in Clause 6.4.2, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasSubtype	ObjectType	AuditSecurityEventType	Defined in Clause 6.4.4		
HasSubtype	ObjectType	AuditNodeManagementEventType	Defined in Clause 6.4.12		
HasSubtype	ObjectType	AuditUpdateEventType	Defined in Clause 6.4.17		
HasProperty	Variable	ActionTimeStamp	UtcTime	.PropertyType	New
HasProperty	Variable	Status	Boolean	.PropertyType	New
HasProperty	Variable	ServerId	String	.PropertyType	New
HasProperty	Variable	ClientAuditEntryId	String	PropertyParams	New
HasProperty	Variable	ClientUserId	String	PropertyParams	New

This *EventType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Clause 6.4.2.

The *ActionTimeStamp* identifies the time the user initiated the action that resulted in the *AuditEvent* being generated. It differs from the *Time Property* because this is the time the server generated the *AuditEvent* documenting the action.

The *Status Property* identifies whether the requested action could be performed (set *Status* to TRUE) or not (set *Status* to FALSE).

The *ServerId* uniquely identifies the server generating the *Event*. It identifies the server uniquely even in a server-controlled transparent redundancy scenario where several servers may use the same URI.

The *ClientAuditEntryId* contains the human-readable *AuditEntryId* defined in [UA Part 3].

The *ClientUserId* identifies the user of the client requesting an action. This is obtained from the system via the information received as part of the session establishment or the *ImpersonateUser Service*. The *ClientUserId* can be obtained from the *UserIdentityToken*. This token can contain the information in multiple formats depending on the type of User Identity that is passed to the service. If the *UserIdentityToken* that was passed was defined as a *UserName*, then the structure contains an explicit string that is the user. If the passed *UserIdentityToken* was defined as X509v3, then the *CertificateData* byte string contains an element that is the user string which can be extracted from the subject key in this structure. If the passed *UserIdentityToken* was defined as WSS, then the user string can be extracted from the WS-Security XML token.

#### 6.4.4 AuditSecurityEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 18.

**Table 18 – AuditSecurityEventType Definition**

Attribute	Value				
BrowseName	AuditSecurityEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditEventType</i> defined in Clause 6.4.3, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasSubtype	ObjectType	AuditChannelEventType	Defined in Clause 6.4.5		
HasSubtype	ObjectType	AuditSessionEventType	Defined in Clause 6.4.8		

This *EventType* inherits all *Properties* of the *AuditEventType*. Their semantic is defined in Clause 6.4.3. There are no additional *Properties* defined for this *EventType*.

#### 6.4.5 AuditChannelEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 19.

**Table 19 – AuditChannelEventType Definition**

Attribute	Value				
BrowseName	AuditChannelEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditSecurityEventType</i> defined in Clause 6.4.4, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasSubtype	ObjectType	AuditOpenSecureChannelEventType	Defined in Clause 6.4.6		
HasSubtype	ObjectType	AuditCloseSecureChannelEventType	Defined in Clause 6.4.7		

This *EventType* inherits all *Properties* of the *AuditSecurityEventType*. Their semantic is defined in Clause 6.4.4. There are no additional *Properties* defined for this *EventType*. The *SourceNode* for *Events* of this type should be assigned to the *Server Object*. The *SourceName* for *Events* of this type should be “SecureChannel/” and the *Service* that generates the *Event* (e.g. *OpenSecureChannel*, *CloseSecureChannel* or *GetSecurityPolicies*).

#### 6.4.6 AuditOpenSecureChannelEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 20.

**Table 20 – AuditOpenSecureChannelEventType Definition**

Attribute	Value				
BrowseName	AuditOpenSecureChannelEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditChannelEventType</i> defined in Clause 6.4.5, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	ClientCertificate	ByteString	.PropertyType	New
HasProperty	Variable	RequestType	SecurityTokenRequestType	.PropertyType	New
HasProperty	Variable	SecurityPolicy	String	.PropertyType	New
HasProperty	Variable	UserIdentityToken	UserIdentityToken	.PropertyType	New
HasProperty	Variable	SecureChannelId	String	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditChannelEventType*. Their semantic is defined in Clause 6.4.5. The *SourceName* for *Events* of this type should be “SecureChannel/OpenSecureChannel”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *ClientCertificate* reflects the *clientCertificate* parameter of the *OpenSecureChannel* *Service* call.

The *RequestType* reflects the *requestType* parameter of the *OpenSecureChannel* *Service* call.

The *SecurityPolicy* reflects the *requestedSecurityPolicy* parameter of the *OpenSecureChannel* *Service* call.

The *UserIdentityToken* reflects the *userIdentityToken* parameter of the *OpenSecureChannel* *Service* call.

The *SecureChannelId* reflects the *secureChannelId* parameter of the *OpenSecureChannel* *Service* call.

#### 6.4.7 AuditCloseSecureChannelEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 21.

**Table 21 – AuditCloseSecureChannelEventType Definition**

Attribute	Value					
BrowseName	AuditCloseSecureChannelEventType					
IsAbstract	True					
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule	
Inherit the <i>Properties</i> of the <i>AuditChannelEventType</i> defined in Clause 6.4.5, i.e. it has <i>HasProperty</i> <i>References</i> to the same <i>Nodes</i> .						
HasProperty	Variable	SecureChannelId	String	.PropertyType	New	

This *EventType* inherits all *Properties* of the *AuditChannelEventType*. Their semantic is defined in Clause 6.4.5. The *SourceName* for *Events* of this type should be “SecureChannel/CloseSecureChannel”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *SecureChannelId* reflects the *secureChannelId* parameter of the *CloseSecureChannel* *Service* call.

#### 6.4.8 AuditSessionEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 22.

**Table 22 – AuditSessionEventType Definition**

Attribute	Value				
BrowseName	AuditSessionEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditEventType</i> defined in Clause 6.4.4, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasSubtype	ObjectType	AuditCreateSessionEventType	Defined in Clause 6.4.9		
HasSubtype	ObjectType	AuditActivateSessionEventType	Defined in Clause 6.4.10		
HasSubtype	ObjectType	AuditImpersonateUserEventType	Defined in Clause 6.4.11		
HasProperty	Variable	SessionId	String	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditEventType*. Their semantic is defined in Clause 6.4.4. There are no additional *Properties* defined for this *EventType*.

If the Event is generated by a *TransferSubscriptions* Service call, the *SourceNode* should be assigned to the *SessionDiagnostics Object* that represents the session. The *SourceName* for *Events* of this type should be “Session/TransferSubscriptions”.

Otherwise, the *SourceNode* for *Events* of this type should be assigned to the *Server Object*. The *SourceName* for *Events* of this type should be “Session/” and the *Service* that generates the *Event* (e.g. *CreateSession*, *ActiveSession*, *ImpersonateUser* or *CloseSession*).

The *SessionId* should contain the *SessionId* of the session that the *Service* call was issued on. If no session context exists (e.g. for a failed *CreateSession* Service call) the *SessionId* is set to 0.

#### 6.4.9 AuditCreateSessionEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 23.

**Table 23 – AuditCreateSessionEventType Definition**

Attribute	Value				
BrowseName	AuditCreateSessionEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditSessionEventType</i> defined in Clause 6.4.8, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	UserIdentityToken	UserIdentityToken	.PropertyType	New
HasProperty	Variable	SecureChannelId	String	PropertyParams	New

This *EventType* inherits all *Properties* of the *AuditSessionEventType*. Their semantic is defined in Clause 6.4.8. The *SourceName* for *Events* of this type should be “Session/CreateSession”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *UserIdentityToken* reflects the *userIdentityToken* parameter of the *CreateSession* *Service* call.

The *SecureChannelId* reflects the *secureChannelId* parameter of the *CreateSession* *Service* call.

#### 6.4.10 AuditActivateSessionEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 24.

**Table 24 – AuditActivateSessionEventType Definition**

Attribute	Value					
BrowseName	AuditActivateSessionEventType					
IsAbstract	True					
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule	
Inherit the <i>Properties</i> of the <i>AuditSessionEventType</i> defined in Clause 6.4.8, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .						
HasProperty	Variable	ClientSoftwareCertificates	SignedSoftwareCertificate[]	.PropertyType	New	

This *EventType* inherits all *Properties* of the *AuditSessionEventType*. Their semantic is defined in Clause 6.4.8. The *SourceName* for *Events* of this type should be “Session/ActivateSession”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *ClientSoftwareCertificates* reflects the *clientSoftwareCertificates* parameter of the *ActivateSession* Service call.

#### 6.4.11 AuditImpersonateUserEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 25.

**Table 25 – AuditImpersonateUserEventType Definition**

Attribute	Value					
BrowseName	AuditImpersonateUserEventType					
IsAbstract	True					
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule	
Inherit the <i>Properties</i> of the <i>AuditSessionEventType</i> defined in Clause 6.4.8, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .						
HasProperty	Variable	UserIdentityToken	UserIdentityToken	.PropertyType	New	

This *EventType* inherits all *Properties* of the *AuditSessionEventType*. Their semantic is defined in Clause 6.4.8. The *SourceName* for *Events* of this type should be “Session/ImpersonateUser”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *UserIdentityToken* reflects the *isерIdentityToken* parameter of the *ImpersonateUser* Service call.

#### 6.4.12 AuditNodeManagementEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 26.

**Table 26 – AuditNodeManagementEventType Definition**

Attribute	Value				
BrowseName	AuditNodeManagementEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditEventType</i> defined in Clause 6.4.3, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasSubtype	ObjectType	AuditAddNodesEventType			
HasSubtype	ObjectType	AuditDeleteNodesEventType			
HasSubtype	ObjectType	AuditAddReferencesEventType			
HasSubtype	ObjectType	AuditDeleteReferencesEventType			

This *EventType* inherits all *Properties* of the *AuditEventType*. Their semantic is defined in Clause 6.4.3. There are no additional *Properties* defined for this *EventType*. The *SourceNode* for *Events* of this type should be assigned to the *Server Object*. The *SourceName* for *Events* of this type should be “NodeManagement/” and the *Service* that generates the *Event* (e.g. *AddNodes*, *AddReferences*, *DeleteNodes*, *DeleteReferences*).

#### 6.4.13 AuditAddNodesEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 27.

**Table 27 – AuditAddNodesEventType Definition**

Attribute	Value				
BrowseName	AuditAddNodesEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditNodeManagementEventType</i> defined in Clause 6.4.12, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	NodesToAdd	AddNodesItem[]	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditNodeManagementEventType*. Their semantic is defined in Clause 6.4.12. The *SourceName* for *Events* of this type should be “NodeManagement/AddNodes”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *NodesToAdd* reflects the *NodesToAdd* parameter of the *AddNodes Service* call.

#### 6.4.14 AuditDeleteNodesEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 28.

**Table 28 – AuditDeleteNodesEventType Definition**

Attribute	Value				
BrowseName	AuditDeleteNodesEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditNodeManagementEventType</i> defined in Clause 6.4.12, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	NodesToDelete	DeleteNodesItem[]	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditNodeManagementEventType*. Their semantic is defined in Clause 6.4.12. The *SourceName* for *Events* of this type should be “NodeManagement/DeleteNodes”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *NodesToDelete* reflects the *nodesToDelete* parameter of the *DeleteNodes Service* call.

#### 6.4.15 AuditAddReferencesEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 29.

**Table 29 – AuditAddReferencesEventType Definition**

Attribute	Value				
BrowseName	AuditAddReferencesEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditNodeManagementEventType</i> defined in Clause 6.4.12, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	ReferencesToAdd	AddReferencesItem[]	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditNodeManagementEventType*. Their semantic is defined in Clause 6.4.12. The *SourceName* for *Events* of this type should be “NodeManagement/AddReferences”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *ReferencesToAdd* reflects the *referencesToAdd* parameter of the *AddReferences Service* call.

#### 6.4.16 AuditDeleteReferencesEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 30.

**Table 30 – AuditDeleteReferenceEventType Definition**

Attribute	Value				
BrowseName	AuditDeleteReferencesEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditNodeManagementEventType</i> defined in Clause 6.4.12, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	ReferencesToDelete	DeleteReferencesItem[]	.PropertyType	New

This *EventType* inherits all *Properties* of the *AuditNodeManagementEventType*. Their semantic is defined in Clause 6.4.12. The *SourceName* for *Events* of this type should be “NodeManagement/DeleteReferences”.

The additional *Properties* defined for this *EventType* reflect parameters of the *Service* call that triggers the *Event*.

The *ReferencesToDelete* reflects the *referencesToDelete* parameter of the *DeleteReferences Service* call.

#### 6.4.17 AuditUpdateEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 31.

**Table 31 – AuditUpdateEventType Definition**

Attribute	Value				
BrowseName	AuditUpdateEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>AuditEventType</i> defined in Clause 6.4.3, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	NodeAttributeld	Int32	.PropertyType	New
HasProperty	Variable	AttributeIndexRange	NumericRange	PropertyParams	New
HasProperty	Variable	NewValue	BaseDataType	PropertyParams	New
HasProperty	Variable	OldValue	BaseDataType	PropertyParams	New

This *EventType* inherits all *Properties* of the *AuditEventType*. Their semantic is defined in Clause 6.4.3. The *SourceNode* for *Events* of this type should be assigned to the *Nodeid* that was changed. The *SourceName* for *Events* of this type should be “Attribute/” and the *Service* that generated the event (e.g. *Write*, *HistoryUpdate*). Note that one *Service* call may generate several *Events* of this type, one per changed value.

The *NodeAttributeld* identifies the *Attribute* that was written on the *SourceNode*.

The *AttributeIndexRange* identifies the index range of the written *Attribute* if the *Attribute* is an array. If the *Attribute* is not an array or the whole array was written, the *AttributeIndexRange* is set to null.

The *NewValue* identifies the value that was written to the *SourceNode*. If the *AttributeIndexRange* is provided, only the value of that range is shown.

The *OldValue* identifies the value that the *SourceNode* contained before the write. If the *AttributeIndexRange* is provided, only the value of that range is shown. It is acceptable for a server that does have this information to report a null value.

Both the *NewValue* and the *OldValue* will contain a value in the *DataType* and encoding used for writing the value.

#### 6.4.18 SystemEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 32.

**Table 32 – SystemEventType Definition**

Attribute	Value				
BrowseName	SystemEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasSubtype	ObjectType	DeviceFailureEventType	Defined in Clause 6.4.19		
Inherit the <i>Properties</i> of the <i>BaseEventType</i> defined in Clause 6.4.2, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					

This *EventType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Clause 6.4.2. There are no additional *Properties* defined for this *EventType*.

#### 6.4.19 DeviceFailureEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 33.

**Table 33 – DeviceFailureEventType Definition**

Attribute	Value				
BrowseName	DeviceFailureEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>SystemEventType</i> defined in Clause 6.4.18, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					

This *EventType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Clause 6.4.2. There are no additional *Properties* defined for this *EventType*.

#### 6.4.20 BaseModelChangeEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 34.

**Table 34 – BaseModelChangeEventType Definition**

Attribute	Value				
BrowseName	BaseModelChangeEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>BaseEventType</i> defined in Clause 6.4.2, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					

This *EventType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Clause 6.4.2. There are no additional *Properties* defined for this *EventType*. The *SourceNode* for Events of this type should be the *Node* of the *View* that gives the context of the changes. If the whole *AddressSpace* is the context, the *SourceNode* is set to null. The *SourceName* for Events of this type should be the *String* part of the *BrowseName* of the *View*; for the whole *AddressSpace* it should be "AddressSpace".

#### 6.4.21 GeneralModelChangeEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 35.

**Table 35 – GeneralModelChangeEventType Definition**

Attribute	Value				
BrowseName	GeneralModelChangeEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>BaseModelChangeEventType</i> defined in Clause 6.4.20, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	Changes	ChangeStructureDataType[]	.PropertyType	New

This *EventType* inherits all *Properties* of the *BaseModelChangeEventType*. Their semantic is defined in Clause 6.4.20.

The additional *Property* defined for this *EventType* reflects the changes that issued the *ModelChangeEvent*. Its structure is defined in Clause 11.14.

#### 6.4.22 PropertyChangeEventType

This *EventType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is formally defined in Table 36.

**Table 36 – PropertyChangeEventType Definition**

Attribute	Value				
BrowseName	PropertyChangeEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>BaseEventType</i> defined in Clause 6.4.2, i.e. it has <i>HasProperty References</i> to the same <i>Nodes</i> .					
HasProperty	Variable	Changes	PropertyChangeStructureDataType[]	PropertyParams	New

This *EventType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Clause 6.4.2. There are no additional *Properties* defined for this *EventType*. The *SourceNode* for Events of this type should be the *Node* of the *View* that gives the context of the changes. If the whole *AddressSpace* is the context, the *SourceNode* is set to null. The *SourceName* for Events of this type should be the *String* part of the *BrowseName* of the *View*, for the whole *AddressSpace* it should be “AddressSpace”.

The additional *Property* defined for this *EventType* reflects the changes that issued the *PropertyChangeEvent*. Its structure is defined in Clause 11.15.

#### 6.5 ModellingRuleType

*ModellingRules* are defined in [UA Part 3]. This *ObjectType* is used as the type for the *ModellingRules*. There are no *References* specified for this *ObjectType*. It is formally defined in Table 37.

**Table 37 – ModellingRuleType Definition**

Attribute	Value				
BrowseName	ModellingRuleType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the <i>BaseObjectType</i> defined in Clause 6.2					

## 6.6 FolderType

Instances of this *ObjectType* are used to organise the *AddressSpace* into a hierarchy of *Nodes*. They represent the *root Node* of a subtree, and have no other semantics associated with them. However, the *DisplayName* of an instance of the *FolderType*, such as “ObjectTypes”, should imply the semantics associated with the use of it. There are no *References* specified for this *ObjectType*. It is formally defined in Table 38.

**Table 38 – FolderType Definition**

Attribute	Value				
BrowseName	FolderType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2					

## 6.7 DataTypeEncodingType

*DataTypeEncodings* are defined in [UA Part 3]. This *ObjectType* is used as type for the *DataTypeEncodings*. There are no *References* specified for this *ObjectType*. It is formally defined in Table 40.

**Table 39 – DataTypeEncodingType Definition**

Attribute	Value				
BrowseName	DataTypeEncodingType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2					

## 6.8 DataTypeSystemType

*DataTypeSystems* are defined in [UA Part 3]. This *ObjectType* is used as type for the *DataTypeSystems*. There are no *References* specified for this *ObjectType*. It is formally defined in Table 40.

**Table 40 – DataTypeSystemType Definition**

Attribute	Value				
BrowseName	DataTypeSystemType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseObjectType defined in Clause 6.2					

## 7 Standard VariableTypes

### 7.1 General

Typically, the components of a complex *VariableType* are fixed and can be extended by subtyping. However, because each *Variable* of a *VariableType* can be extended with additional components, UA allows the extension of the standard *VariableTypes* defined in this document with additional components. This allows the expression of additional information in the type definition that would be contained in each *Variable* anyway. However, it is not allowed to restrict the components of the standard *VariableTypes* defined in this part. An example of extending *VariableTypes* would be putting the standard *Property NodeVersion*, defined in [UA Part 3], into the *BaseDataVariableType*, stating that each *DataVariable* of the server will provide a *NodeVersion*.

### 7.2 BaseVariableType

The *BaseVariableType* is the abstract base type for all other *VariableTypes*. However, only the *.PropertyType* and the *BaseDataVariableType* directly inherit from this type.

There are no *References*, except for *HasSubtype References*, specified for this *VariableType*. It is formally defined in Table 41.

**Table 41 – BaseVariableType Definition**

Attribute	Value				
BrowseName	BaseVariableType				
IsAbstract	True				
ArraySize	-1				
DataType	BaseDataType				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasSubtype	VariableType	.PropertyType		Defined in Clause 7.3	
HasSubtype	VariableType	BaseDataVariableType		Defined in Clause 7.4	

### 7.3 PropertyType

The *.PropertyType* is a subtype of the *BaseVariableType*. It is used as the type definition for all *Properties*. *Properties* are defined by their *BrowseName* and therefore do not need a specialised type definition. It is not allowed to subtype this *VariableType*.

There are no *References* specified for this *VariableType*. It is formally defined in Table 42.

**Table 42 – PropertyType Definition**

Attribute	Value				
BrowseName	.PropertyType				
IsAbstract	False				
ArraySize	-1				
DataType	BaseDataType				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseVariableType defined in Clause 7.2					

### 7.4 BaseDataVariableType

The *BaseDataVariableType* is a subtype of the *BaseVariableType*. It is used as the type definition whenever there is a *DataVariable* having no more concrete type definition available. This *VariableType* is the base *VariableType* for *VariableTypes* of *DataVariables*, and all other *VariableTypes* of *DataVariables* must either directly or indirectly inherit from it. However, it may not be possible for servers to provide all *HasSubtype References* from this *VariableType* to its subtypes, and therefore it is not required to provide this information.

There are no *References* except for *HasSubtype* *References* specified for this *VariableType*. It is formally defined in Table 43.

**Table 43 – BaseDataVariableType Definition**

Attribute	Value		
BrowseName	BaseDataVariableType		
IsAbstract	False		
ArraySize	-1		
DataType	BaseDataType		
References	NodeClass	BrowseName	Comment
Subtype of the BaseVariableType defined in Clause 7.2			
HasSubtype	VariableType	ServerVendorCapabilityType	Defined in Clause 7.5
HasSubtype	VariableType	DataTypeDictionaryType	Defined in Clause 7.6
HasSubtype	VariableType	ServerStatusType	Defined in Clause 7.8
HasSubtype	VariableType	BuildInfoType	Defined in Clause 7.9
HasSubtype	VariableType	ServerDiagnosticsSummaryType	Defined in Clause 7.10
HasSubtype	VariableType	SamplingRateDiagnosticsArrayType	Defined in Clause 7.11
HasSubtype	VariableType	SamplingRateDiagnosticsType	Defined in Clause 7.12
HasSubtype	VariableType	SubscriptionDiagnosticsArrayType	Defined in Clause 7.13
HasSubtype	VariableType	SubscriptionDiagnosticsType	Defined in Clause 7.14
HasSubtype	VariableType	SessionDiagnosticsArrayType	Defined in Clause 7.15
HasSubtype	VariableType	SessionDiagnosticsVariableType	Defined in Clause 7.16
HasSubtype	VariableType	SessionSecurityDiagnosticsArrayType	Defined in Clause 7.17
HasSubtype	VariableType	SessionSecurityDiagnosticsType	Defined in Clause 7.18

## 7.5 ServerVendorCapabilityType

This *VariableType* is an abstract type whose subtypes define capabilities of the server. Vendors may define subtypes of this type. This *VariableType* is formally defined in Table 44.

**Table 44 – ServerVendorCapabilityType Definition**

Attribute	Value				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					

## 7.6 DataTypeDictionaryType

*DataTypeDictionaries* are defined in [UA Part 3]. This *VariableType* is used as the type for the *DataTypeDictionaries*. There are no *References* specified for this *VariableType*. It is formally defined in Table 45.

**Table 45 – DataTypeDictionaryType Definition**

Attribute	Value				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					

## 7.7 DataTypeDescriptionType

*DataTypeDescriptions* are defined in [UA Part 3]. This *VariableType* is used as the type for the *DataTypeDescriptions*. There are no *References* specified for this *VariableType*. It is formally defined in Table 45.

**Table 46 – DataTypeDescriptionType Definition**

Attribute	Value				
BrowseName	DataTypeDictionaryType				
IsAbstract	False				
ArraySize	-1				
DataType	BaseDataType	NodeClass	BrowseName	DataType	TypeDefinition
References					ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					

## 7.8 ServerStatusType

This complex *VariableType* is used for information about the server status. Its *DataVariables* reflect its *DataType* having the same semantic defined in Clause 11.9. The *VariableType* is formally defined in Table 47.

**Table 47 – ServerStatusType Definition**

Attribute	Value				
BrowseName	ServerStatusType				
IsAbstract	False				
ArraySize	-1				
DataType	ServerStatusDataType	NodeClass	BrowseName	DataType	TypeDefinition
References					Modelling Rule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	StartTime	UtcTime	BaseDataVariableType	New
HasComponent	Variable	CurrentTime	UtcTime	BaseDataVariableType	New
HasComponent	Variable	State	ServerState	BaseDataVariableType	New
HasComponent	Variable	BuildInfo <sup>1</sup>	BuildInfo	BuildInfoType	New
Notes –					
1) Containing Objects and Variables of these Objects and Variables are defined by their BrowseName defined in the corresponding TypeDefinitionNode. The NodeId is defined by the composed symbolic name described in Clause 4.1.					

## 7.9 BuildInfoType

This complex *VariableType* is used for information about the server status. Its *DataVariables* reflect its *DataType* having the same semantic defined in [UA Part 4]. The *VariableType* is formally defined in Table 48.

**Table 48 – BuildInfoType Definition**

Attribute	Value				
BrowseName	BuildInfoType				
IsAbstract	False				
ArraySize	-1				
DataType	BuildInfo	NodeClass	BrowseName	DataType	TypeDefinition
References					ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	ApplicationUri	String	BaseDataVariableType	New
HasComponent	Variable	ManufacturerName	String	BaseDataVariableType	New
HasComponent	Variable	ApplicationName	String	BaseDataVariableType	New
HasComponent	Variable	SoftwareVersion	String	BaseDataVariableType	New
HasComponent	Variable	BuildNumber	String	BaseDataVariableType	New
HasComponent	Variable	BuildDate	UtcTime	BaseDataVariableType	New

## 7.10 ServerDiagnosticsSummaryType

This complex *VariableType* is used for diagnostic information. Its *DataVariables* reflect its *DataType* having the same semantic defined in Clause 11.8. The *VariableType* is formally defined in Table 49.

**Table 49 – ServerDiagnosticsSummaryType Definition**

Attribute	Value				
BrowseName	ServerDiagnosticsSummaryType				
IsAbstract	False				
ArraySize	-1				
DataType	ServerDiagnosticsSummaryDataType				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	ServerViewCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentSessionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CumulatedSessionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SecurityRejectedSessionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	RejectSessionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SessionTimeoutCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SessionAbortCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SamplingRateCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	PublishingRateCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentSubscriptionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CumulatedSubscriptionCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SecurityRejectedRequestsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	RejectedRequestsCount	UInt32	BaseDataVariableType	New

## 7.11 SamplingRateDiagnosticsArrayType

This complex *VariableType* is used for diagnostic information. For each entry of the array, instances of this type will provide a *Variable* of the SamplingRateDiagnosticsType *VariableType* having the sampling rate as *BrowseName*. The *VariableType* is formally defined in Table 50.

**Table 50 – SamplingRateDiagnosticsArrayType Definition**

Attribute	Value				
BrowseName	SamplingRateDiagnosticsArrayType				
IsAbstract	False				
ArraySize	0				
DataType	SamplingRateDiagnosticsDataType				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					
ExposeSItsArray	VariableType	SamplingRateDiagnosticsType	Defined in Clause 7.12		

## 7.12 SamplingRateDiagnosticsType

This complex *VariableType* is used for diagnostic information. Its *DataVariables* reflect its *DataType*, having the same semantic defined in Clause 11.7. The *VariableType* is formally defined in Table 51.

**Table 51 – SamplingRateDiagnosticsType Definition**

Attribute	Value				
BrowseName	SamplingRateDiagnosticsType				
IsAbstract	False				
ArraySize	-1				
DataType	SamplingRateDiagnosticsDataType				
References	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	SamplingRate	UInt32	BaseDataVariableType	New
HasComponent	Variable	SamplingErrorCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	SampledMonitoredItemsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	MaxSampledMonitoredItemsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	DisabledMonitoredItemsSamplingCount	UInt32	BaseDataVariableType	New

## 7.13 SubscriptionDiagnosticsArrayType

This complex *VariableType* is used for diagnostic information. For each entry of the array, instances of this type will provide a *Variable* of the *SubscriptionDiagnosticsType VariableType* having the *SubscriptionId* as *BrowseName*. The *VariableType* is formally defined in Table 52.

**Table 52 – SubscriptionDiagnosticsArrayType Definition**

Attribute	Value				
BrowseName	SubscriptionDiagnosticsArrayType				
IsAbstract	False				
ArraySize	0				
DataType	SubscriptionDiagnosticsDataType				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the BaseDataVariableType defined in Clause 7.4					
ExposeItsArray	VariableType	SubscriptionDiagnosticsType	Defined in Clause 7.14		

## 7.14 SubscriptionDiagnosticsType

This complex *VariableType* is used for diagnostic information. Its *DataVariables* reflect its *DataType*, having the same semantic defined in Clause 11.13. The *VariableType* is formally defined in Table 53.

**Table 53 – SubscriptionDiagnosticsType Definition**

Attribute	Value				
BrowseName	SubscriptionDiagnosticsType				
IsAbstract	False				
ArraySize	-1				
DataType	SubscriptionDiagnosticsDataType				
References	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	SessionId	Int32	BaseDataVariableType	New
HasComponent	Variable	SubscriptionId	Int32	BaseDataVariableType	New
HasComponent	Variable	Priority	Byte	BaseDataVariableType	New
HasComponent	Variable	PublishingRate	UInt32	BaseDataVariableType	New
HasComponent	Variable	ModifyCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	EnableCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	DisableCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	RepublishRequestCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	RepublishMsgRequestCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	RepublishMessageCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	TransferRequestCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	TransferredToAltClientCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	TransferredToSameClientCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	PublishRequestCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	DataChangeNotificationsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	EventNotificationsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	NotificationsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	LateStateCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	KeepAliveStateCount	UInt32	BaseDataVariableType	New

## 7.15 SessionDiagnosticsArrayType

This complex *VariableType* is used for diagnostic information. The *VariableType* is formally defined in Table 54.

**Table 54 – SessionDiagnosticsArrayType Definition**

Attribute	Value				
BrowseName	SessionDiagnosticsArrayType				
IsAbstract	False				
ArraySize	0				
DataType	SessionDiagnosticsDataType				
References	NodeClass	Browse Name	DataType	TypeDefinition	Modelling Rule
Subtype of the BaseDataVariableType defined in Clause 7.4					

## 7.16 SessionDiagnosticsVariableType

This complex *VariableType* is used for diagnostic information. Its *DataVariables* reflect its *DataType*, having the same semantic defined in Clause 11.10. The *VariableType* is formally defined in Table 55.

**Table 55 – SessionDiagnosticsVariableType Definition**

Attribute	Value				
References	Node Class	BrowseName	DataType	TypeDefinition	Mod. Rule
Subtype of the BaseDataVariableType defined in Clause 7.4					
HasComponent	Variable	SessionId	Int32	BaseDataVariableType	New
HasComponent	Variable	ClientName	string	BaseDataVariableType	New
HasComponent	Variable	LocaleIds	LocaleId[]	BaseDataVariableType	New
HasComponent	Variable	RequestedSessionTimeout	Int32	BaseDataVariableType	New
HasComponent	Variable	ClientConnectionTime	UtcTime	BaseDataVariableType	New
HasComponent	Variable	ClientLastContactTime	UtcTime	BaseDataVariableType	New
HasComponent	Variable	CurrentSubscriptionsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentMonitoredItemsCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentPublishRequestsInQueue	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentPublishTimerExpirations	UInt32	BaseDataVariableType	New
HasComponent	Variable	KeepAliveCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	CurrentRepublishRequestsInQueue	UInt32	BaseDataVariableType	New
HasComponent	Variable	MaxRepublishRequestsInQueue	UInt32	BaseDataVariableType	New
HasComponent	Variable	RepublishCounter	UInt32	BaseDataVariableType	New
HasComponent	Variable	PublishingCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	PublishingQueueOverflowCount	UInt32	BaseDataVariableType	New
HasComponent	Variable	ReadCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	HistoryReadCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	WriteCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	HistoryUpdateCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	MethodCallCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	CreateMonitoredItemCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	ModifyMonitoredItemCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	SetMonitoringModeCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	SetTriggeringCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	DeleteMonitoredItemsCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	CreateSubscriptionCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	ModifySubscriptionCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	SetPublishingModeCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	PublishCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	RepublishCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	TransferSubscriptionsCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	DeleteSubscriptionsCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	AddNodesCount	ServiceCounter DataType	BaseDataVariableType	New

HasComponent	Variable	AddReferencesCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	DeleteNodesCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	DeleteReferencesCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	BrowseCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	BrowseNextCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	TranslateBrowsePathsToNodeIdsCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	QueryFirstCount	ServiceCounter DataType	BaseDataVariableType	New
HasComponent	Variable	QueryNextCount	ServiceCounter DataType	BaseDataVariableType	New

### 7.17 SessionSecurityDiagnosticsArrayType

This complex *VariableType* is used for diagnostic information. For each entry of the array instances of this type will provide a *Variable* of the *SessionSecurityDiagnosticsType VariableType*, having the *SessionSecurityDiagnostics* as *BrowseName*. Those Variables will also be referenced by the *SessionDiagnostics Objects* defined by their type in Clause 6.3.5, and therefore have the *ModellingRule Shared*. The *VariableType* is formally defined in Table 56. Since this information is security related, it should not be made accessible to all users, but only to authorised users.

**Table 56 – SessionSecurityDiagnosticsArrayType Definition**

Attribute	Value				
BrowseName	SessionSecurityDiagnosticsArrayType				
IsAbstract	False				
ArraySize	0				
DataType	SessionSecurityDiagnosticsDataType				
References	NodeClass	Browse Name	DataType	TypeDefinition	Mod. Rule
Subtype of the <i>BaseDataVariableType</i> defined in Clause 7.4					
HasComponent	Variable	--	SamplingRateDiagnosticsDataType	SessionSecurityDiagnosticsType	--

### 7.18 SessionSecurityDiagnosticsType

This complex *VariableType* is used for diagnostic information. Its *DataVariables* reflect its *DataType*, having the same semantic defined in Clause 11.11. The *VariableType* is formally defined in Table 57. Since this information is security-related, it should not be made accessible to all users, but only to authorised users.

**Table 57 – SessionSecurityDiagnosticsType Definition**

Attribute	Value				
BrowseName	SessionSecurityDiagnosticsType				
IsAbstract	False				
ArraySize	-1				
DataType	SessionDiagnosticsDataType				
References	Node Class	BrowseName	DataType	Type Definition	Modelling Rule
Subtype of the <i>BaseDataVariableType</i> defined in Clause 7.4					
HasComponent	Variable	SessionId	Int32	BaseDataVariableType	New
HasComponent	Variable	ClientUserIdOfSession	String	BaseDataVariableType	New
HasComponent	Variable	ClientUserIdHistory	String[]	BaseDataVariableType	New
HasComponent	Variable	AuthenticationMechanism	String	BaseDataVariableType	New
HasComponent	Variable	Encoding	String	BaseDataVariableType	New
HasComponent	Variable	TransportProtocol	String	BaseDataVariableType	New
HasComponent	Variable	SecurityPolicy	String	BaseDataVariableType	New

## 8 Standard Objects and their Variables

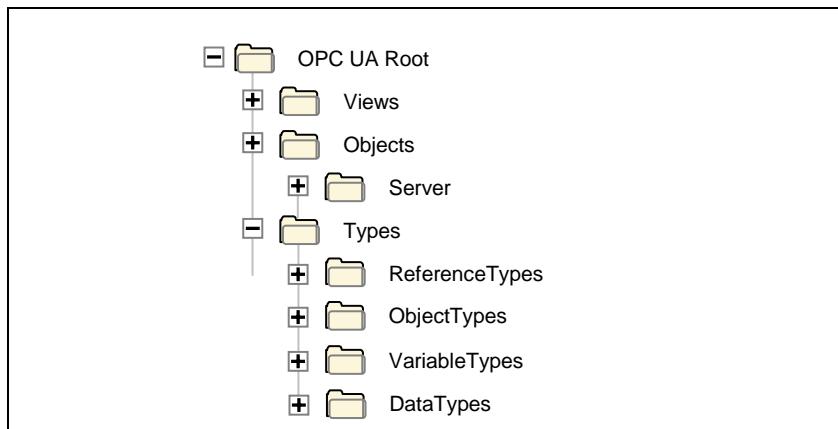
### 8.1 General

Objects and Variables described in the following subclauses can be extended by additional Properties or References to other Nodes, except where it is stated in the text that it is restricted.

### 8.2 Objects used to organise the AddressSpace structure

#### 8.2.1 Overview

To promote interoperability of clients and servers, the UA AddressSpace is structured as a hierarchy, with the top levels standardised for all servers. Figure 1 illustrates the structure of the AddressSpace. All *Objects* in this figure are organised using *Organizes References* and have the *ObjectType FolderType* as type definition.



**Figure 1 – Standard AddressSpace Structure**

The remainder of this Clause provides descriptions of these standard *Nodes* and the organization of *Nodes* beneath them. Servers typically implement a subset of these standard *Nodes*, depending on their capabilities.

#### 8.2.2 Root

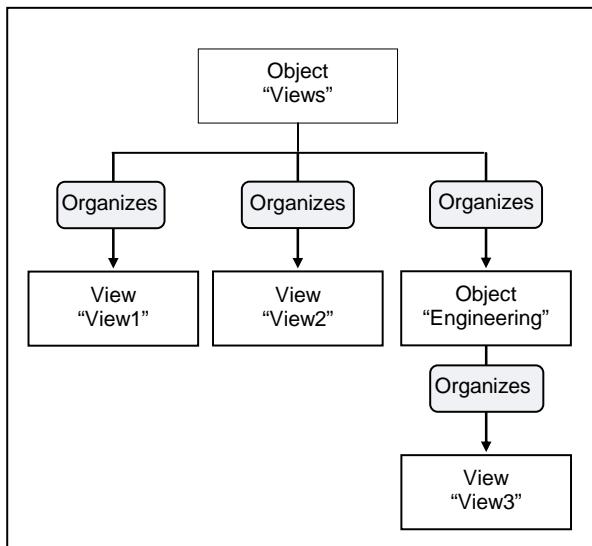
This standard *Object* is the browse entry point for the AddressSpace. It contains a set of *Organizes References* that point to the other standard *Objects*. The “Root” *Object* may not reference any other *NodeClasses*. It is formally defined in Table 58.

**Table 58 – Root Definition**

Attribute	Value		
BrowseName	Root		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	Object	Views	Defined in Clause 8.2.3
Organizes	Object	Objects	Defined in Clause 8.2.4
Organizes	Object	Types	Defined in Clause 8.2.5

### 8.2.3 Views

This standard *Object* is the browse entry point for *Views*. Only *Organizes References* are used to relate *View Nodes* to the “Views” standard *Object*. All *View Nodes* in the *AddressSpace* must be referenced by this *Node*, either directly or indirectly. I.e. the “Views” *Object* may reference other *Objects* using *Organizes References*. Those *Objects* may reference additional *Views*. Figure 2 illustrates this. The “Views” standard *Object* directly references the *Views* “View1” and “View2” and indirectly “View3” by referencing another *Object* called “Engineering”.



**Figure 2 – Views Organization**

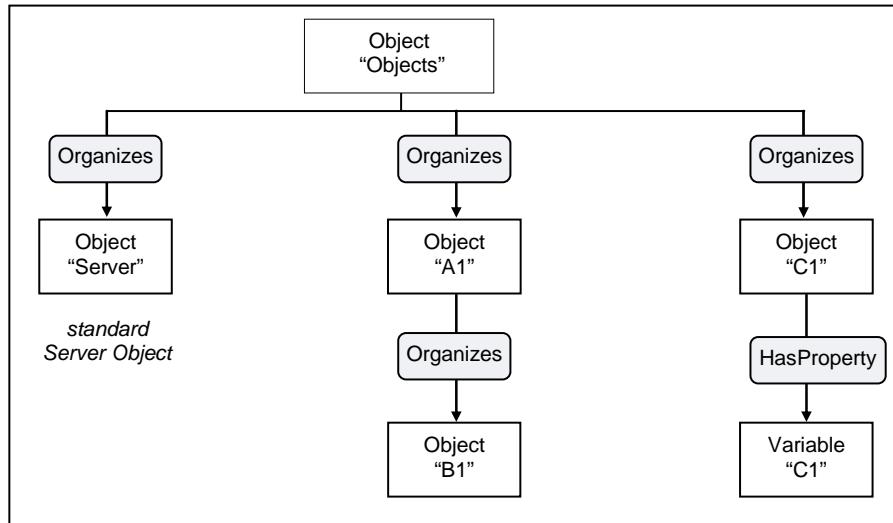
The “Views” *Object* may not reference any other *NodeClasses*. The “Views” *Object* is formally defined in Table 59.

**Table 59 – Views Definition**

Attribute	Value		
BrowseName	Views		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6

### 8.2.4 Objects

This standard *Object* is the browse entry point for *Object Nodes*. Figure 3 illustrates the structure beneath this *Node*. Only *Organizes References* are used to relate *Objects* to the “Objects” standard *Object*. The intent of the “Objects” *Object* is that all *Objects* and *Variables* that are not used for type definitions or other organizational purposes (e.g. organizing the *Views*) are accessible through *hierarchical References* starting from this *Node*. However, this is not a requirement, because not all servers may be able to support this. This *Object* references the standard *Server Object* defined in Clause 8.3.2.

**Figure 3 – Objects Organization**

The “Objects” Object may not reference any other *NodeClasses*. The “Objects” Object is formally defined in Table 60.

**Table 60 – Objects Definition**

<b>Attribute</b>	<b>Value</b>		
BrowseName	Objects		
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>	<b>Comment</b>
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	Object	Server	Defined in Clause 8.3.2

### 8.2.5 Types

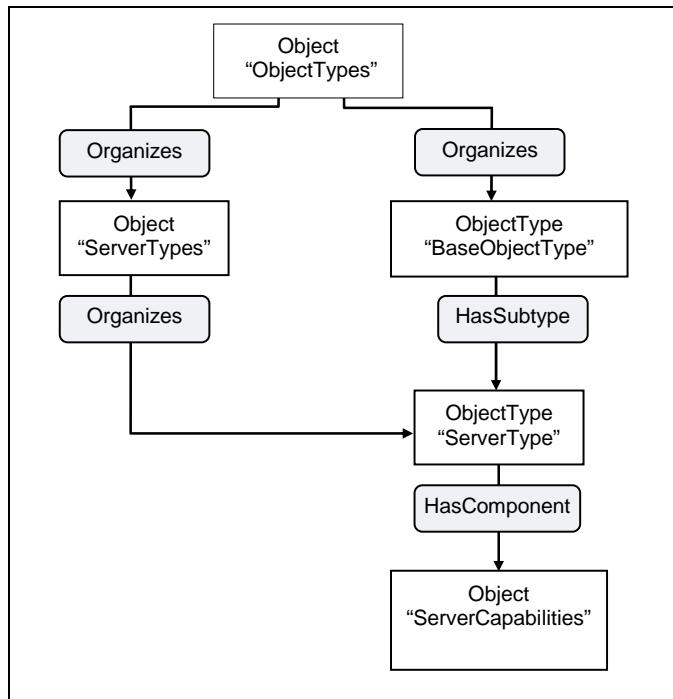
This standard *Object Node* is the browse entry point for type *Nodes*. Figure 1 illustrates the structure beneath this *Node*. Only *Organizes References* are used to relate *Objects* to the “Types” standard *Object*. The “Types” Object may not reference any other *NodeClasses*. It is formally defined in Table 61.

**Table 61 – Types Definition**

<b>Attribute</b>	<b>Value</b>		
BrowseName	Types		
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>	<b>Comment</b>
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	Object	ObjectTypes	Defined in Clause 8.2.6
Organizes	Object	VariableTypes	Defined in Clause 8.2.7
Organizes	Object	ReferenceTypes	Defined in Clause 8.2.8
Organizes	Object	DataTypes	Defined in Clause 8.2.9

### 8.2.6 ObjectTypes

This standard *Object Node* is the browse entry point for *ObjectType Nodes*. Figure 4 illustrates the structure beneath this *Node* showing some of the standard *ObjectTypes* defined in Clause 6. Only *Organizes References* are used to relate *Objects* and *ObjectTypes* to the “*ObjectTypes*” standard *Object*. The “*ObjectTypes*” *Object* may not reference any other *NodeClasses*.



**Figure 4 – ObjectTypes Organization**

The intention of the “*ObjectTypes*” *Object* is that all *ObjectTypes* of the server are either directly or indirectly accessible browsing *HierarchicalReferences* starting from this *Node*. However, this is not required and servers may not provide some of their *ObjectTypes* because they may be well-known in the industry, such as the *Server Type* defined in Clause 6.3.1.

This *Object* also indirectly references the *BaseEventType* defined in Clause 6.4.2, which is the base type of all *EventTypes*. Thereby it is the entry point for all *EventTypes* provided by the server. It is required that the server expose all its *EventTypes*, so a client can usefully subscribe to *Events*.

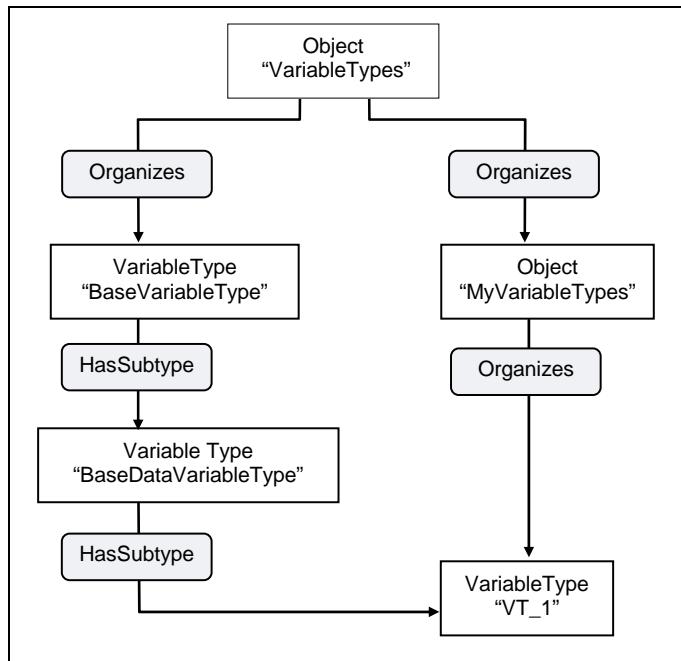
The “*ObjectTypes*” *Object* is formally defined in Table 62.

**Table 62 – ObjectTypes Definition**

Attribute	Value		
BrowseName	ObjectType		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	ObjectType	BaseObjectType	Defined in Clause 6.2

### 8.2.7 VariableTypes

This standard *Object* is the browse entry point for *VariableType Nodes*. Figure 5 illustrates the structure beneath this *Node*. Only *Organizes References* are used to relate *Objects* and *VariableTypes* to the “*VariableTypes*” standard *Object*. The “*VariableTypes*” *Object* may not reference any other *NodeClasses*.



**Figure 5 – VariableTypes Organization**

The intent of the “*VariableTypes*” *Object* is that all *VariableTypes* of the server are either directly or indirectly accessible browsing *HierarchicalReferences* starting from this *Node*. However, this is not required and servers may not provide some of their *VariableTypes*, because they may be well-known in the industry, such as the “*BaseVariableType*” defined in Clause 7.2.

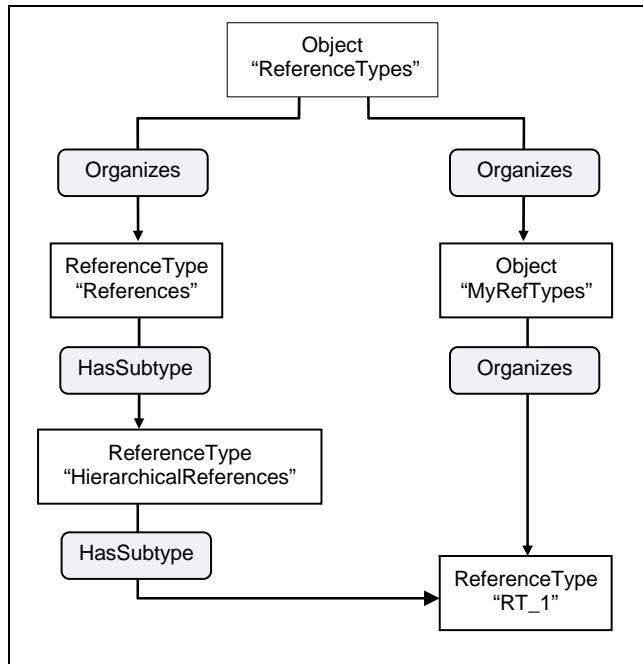
The “*VariableTypes*” *Object* is formally defined in Table 63.

**Table 63 – VariableTypes Definition**

Attribute	Value		
BrowseName	VariableTypes		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	VariableType	BaseVariableType	Defined in Clause 7.2

### 8.2.8 ReferenceTypes

This standard *Object* is the browse entry point for *ReferenceType Nodes*. Figure 6 illustrates the organization of *ReferenceTypes*. *Organizes References* are used to define *ReferenceTypes* and *Objects* referenced by the “*ReferenceTypes*” *Object*. The “*ReferenceTypes*” *Object* may not reference any other *NodeClasses*. See Clause 10 for a discussion of the standard *ReferenceTypes* that appear beneath the “*ReferenceTypes*” *Object*.



**Figure 6 – ReferenceType Definitions**

Since *ReferenceTypes* will be used as filters in the browse Service and in queries, the server must provide all its *ReferenceTypes*, directly or indirectly following *hierarchical References* starting from the “*ReferenceTypes*” *Object*. This means that, whenever the client follows a *Reference*, the server must expose the type of this *Reference* in the *ReferenceType* hierarchy. It must provide all *ReferenceTypes* so that the client would be able, following the inverse subtype of *References*, to come to the base *References ReferenceType*. It does not mean that the server must expose the *ReferenceTypes* that the client has not used any *Reference* of.

The “*ReferenceTypes*” *Object* is formally defined in Table 64.

**Table 64 – ReferenceTypes Definition**

Attribute	Value		
BrowseName	ReferenceTypes		
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>	<b>Comment</b>
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	ReferenceType	References	Defined in Clause 10.1

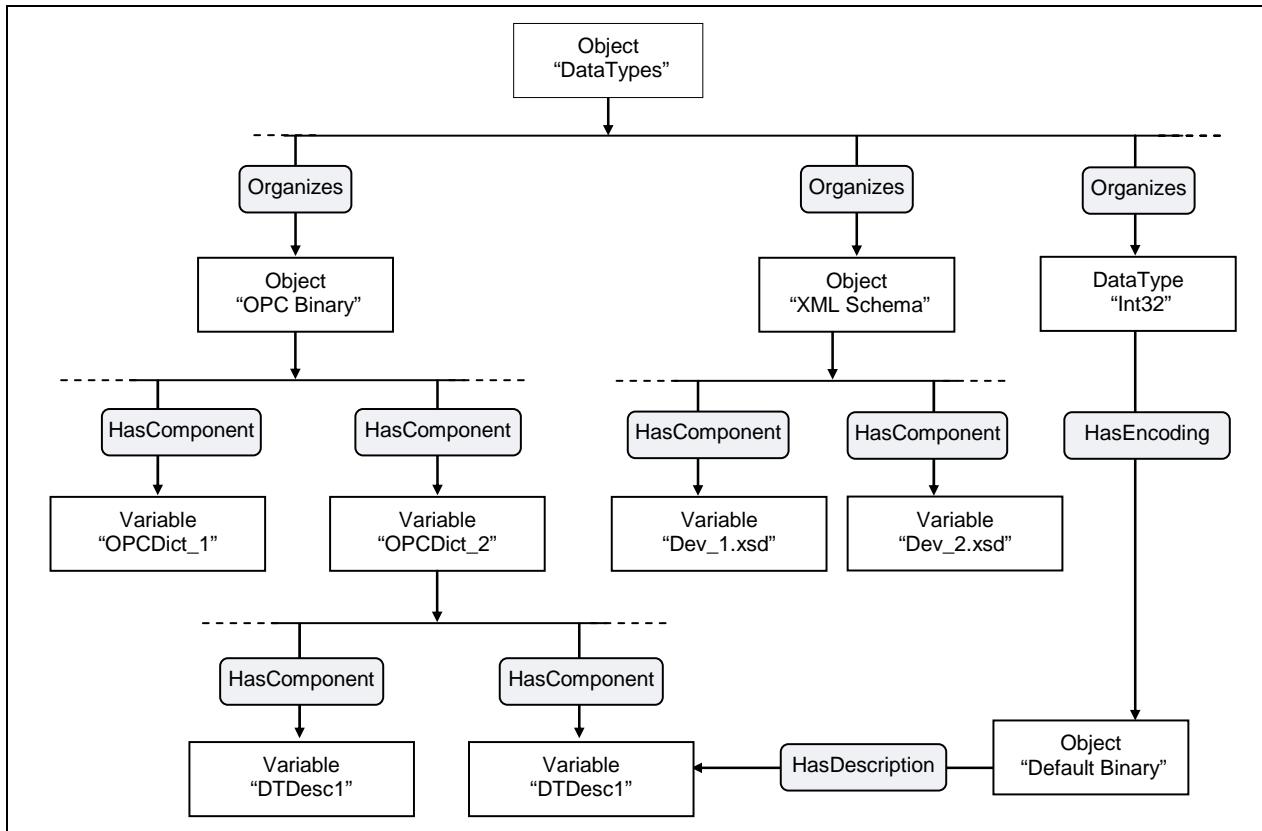
### 8.2.9 DataTypes

This standard *Object* is the browse entry point for *DataTypes* that the server wishes to expose in the *AddressSpace*. The standard *Object* uses *Organizes References* to reference *Objects* of the *DataTypeSystemType* representing *DataTypeSystems*. Referenced by those *Objects* are *DataTypeDictionaries* that refer to their *DataTypeDescriptions*. However, it is not required to provide the *DataTypeSystem Objects*, and the *DataTypeDictionary* need not to be provided.

Because *DataTypes* are not related to *DataTypeDescriptions* using *hierarchical References*, *DataType Nodes* should be made available using *Organizes References* pointing either directly from the “*DataTypes*” *Object* to the *DataType Nodes* or using additional *Folder Objects* for grouping

purposes. The intent is that all *DataTypes* of the server exposed in the *AddressSpace* are accessible following *hierarchical References* starting from the “*DataTypes*” *Object*. However, this is not required.

Figure 7 illustrates this hierarchy using the “*OPC Binary*” and “*XML Schema*” standard *DataTypeSystems* as examples. Other *DataTypeSystems* may be defined under this *Object*.



**Figure 7 – DataTypes Organization**

Each *DataTypeSystem Object* is related to its *DataTypeDictionary Nodes* using *HasComponent References*. Each *DataTypeDictionary Node* is related to its *DataTypeDescription Nodes* using *HasComponent References*. These *References* indicate that the *DataTypeDescriptions* are defined in the dictionary.

In the example, the “*DataTypes*” *Object* references the *DataType* “*Int32*” using an *Organizes Reference*. The *DataType* uses the non-hierarchical *HasEncoding Reference* to point to its default encoding, which references a *DataTypeDescription* using the non-hierarchical *HasDescription Reference*.

The “*DataTypes*” *Object* is formally defined in Table 65.

**Table 65 – DataTypes Definition**

Attribute	Value		
BrowseName	DataTypes		
References	<b>NodeClass</b>	<b>BrowseName</b>	<b>Comment</b>
HasTypeDefinition	ObjectType	FolderType	Defined in Clause 6.6
Organizes	Object	OPC Binary	Defined in Clause 8.2.10
Organizes	Object	XML Schema	Defined in Clause 8.2.11

### 8.2.10 OPC Binary

OPC Binary is a standard *DataTypeSystem* defined by OPC. It is represented in the *AddressSpace* by an *Object Node*. The OPC Binary *DataTypeSystem* is defined in [UA Part 3]. OPC Binary uses XML to describe complex binary data values. The “OPC Binary” *Object* is formally defined in Table 66.

**Table 66 – OPC Binary Definition**

Attribute	Value		
BrowseName	OPC Binary		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	DataTypeSystemType	Defined in Clause 6.8

### 8.2.11 XML Schema

XML Schema is a standard *DataTypeSystem* defined by the W3C. It is represented in the *AddressSpace* by an *Object Node*. XML Schema documents are XML documents whose *xmlns* attribute in the first line is:

schema *xmlns* =<http://www.w3.org/1999/XMLSchema>

The “XML Schema” *Object* is formally defined in Table 67.

**Table 67 – XML Schema Definition**

Attribute	Value		
BrowseName	XML Schema		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	DataTypeSystemType	Defined in Clause 6.8

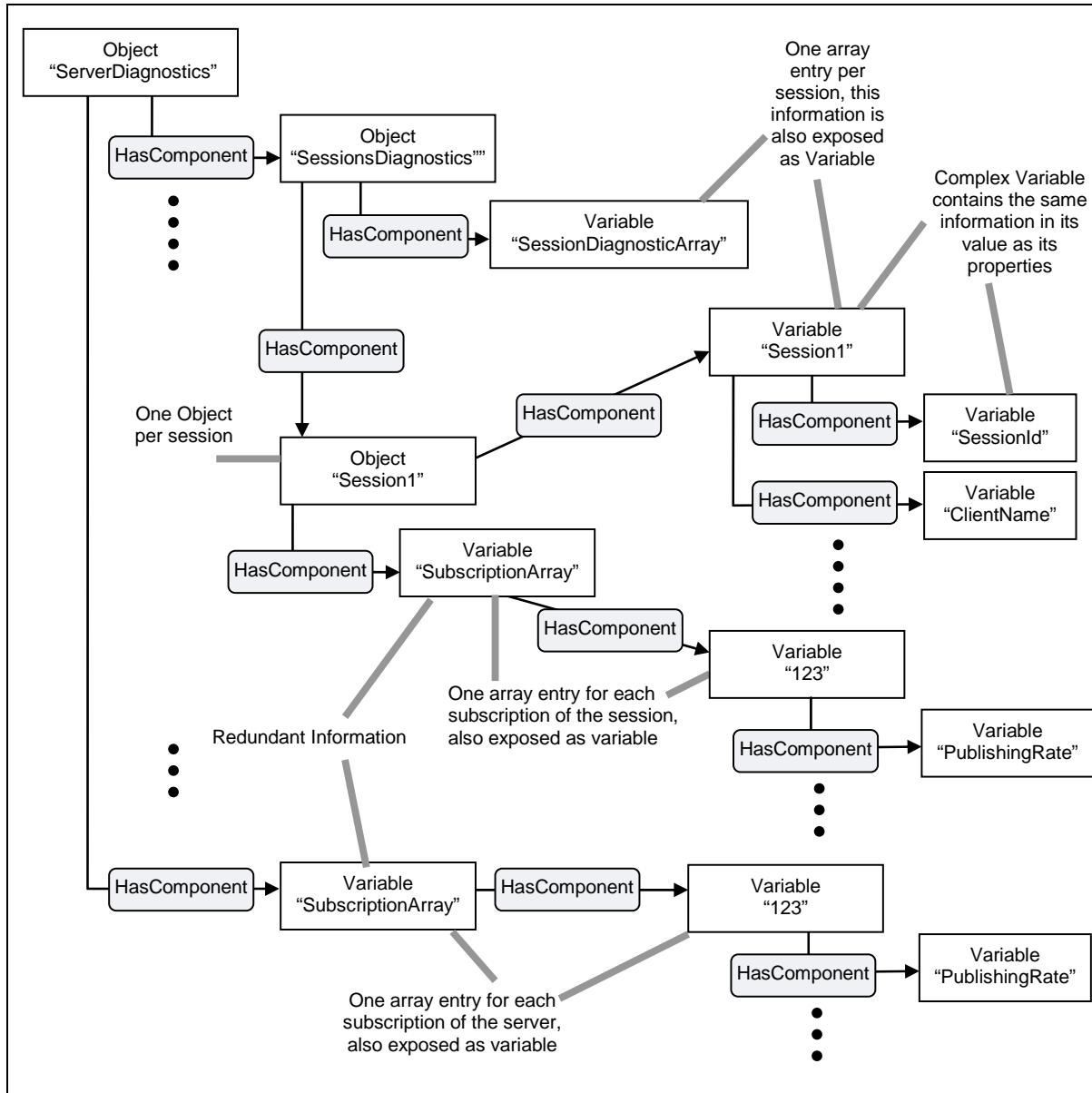
## 8.3 Server Object and its containing Objects

### 8.3.1 General

The *Server Object* and its containing *Objects* and *Variables* are built in a way that the information can be gained in several ways, suitable for different kinds of clients having different requirements. Appendix A gives an overview of the design decisions made in providing the information in that way, and discusses the pros and cons of the different approaches. Figure 8 gives an overview of the containing *Objects* and *Variables* of the diagnostic information of the *Server Object* and where the information can be found.

The *SessionsDiagnostics Object* contains one *Object* per session and a *Variable* with an array with one entry per session. This array is of a complex *DataType* holding the diagnostic information about the session. Each *Object* representing a session references a complex *Variable* containing the information about the session using the same *DataType* as the array containing information about all sessions. Such a *Variable* also exposes all its information as *Variables* with simple *DataTypes* containing the same information as in the complex *DataType*. Not shown in Figure 8 is the security-related information per session, which follows the same rules.

The server provides an array with an entry per subscription containing diagnostic information about this subscription. Each entry of this array is also exposed as a complex *Variable* with *Variables* for each individual value. Each *Object* representing a session also provides such an array, but providing the subscriptions of the session.

**Figure 8 – Excerpt of Diagnostic Information of the Server**

### 8.3.2 Server Object

This *Object* is used as the browse entry point for information about the server. The content of this *Object* is already defined by its type definition in Clause 6.3.1. It is formally defined in Table 68. The *Server Object* serves as root notifier, i.e. its *EventNotifier Attribute* must be set providing *Events*. All *Events* of the server must be accessible subscribing to the *Events* of the *Server Object*.

The *SessionDiagnostics Object*, containing diagnostic information about the session the client currently runs on, has a special symbolic name associated to it. This symbolic name does not represent the *BrowseName* of the *SessionDiagnostics Object*. The symbolic name is “*Server.ServerDiagnostics.SessionsDiagnostics.MySession*”. This *NodeID* is the same for all clients connected to the server, although it always represents the information specific to the session of the client.

**Table 68 – Server Definition**

Attribute	Value				
BrowseName	Server				
References	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule
HasTypeDefinition	ObjectType	ServerType	Defined in Clause 6.3.1		
HasProperty	Variable	ServerArray	String[]	.PropertyType	New
HasProperty	Variable	NamespaceArray	String[]	.PropertyType	New
HasComponent	Variable	ServerStatus <sup>1</sup>	ServerStatusDataType	.PropertyType	New
HasProperty	Variable	ServiceLevel	SByte	.PropertyType	New
HasComponent	Object	ServerCapabilities <sup>1</sup>	--	ServerCapabilities	New
HasComponent	Object	ServerDiagnostics <sup>1</sup>	--	ServerDiagnosticsType	New
HasComponent	Object	VendorServerInfo	--	vendor-specific <sup>2</sup>	New
HasComponent	Object	ServerRedundancy <sup>1</sup>	--	depends on supported redundancy <sup>3</sup>	New
Notes –					
1) Containing <i>Objects</i> and <i>Variables</i> of these <i>Objects</i> and <i>Variables</i> are defined by their <i>BrowseName</i> defined in the corresponding <i>TypeDefinitionNode</i> . The <i>NodeID</i> is defined by the composed symbolic name described in Clause 4.1.					
2) Must be the <i>VendorServerInfo ObjectType</i> or one of its subtypes					
3) Must be the <i>ServerRedundancyType</i> or one of its subtypes					

### 8.4 ModellingRule Objects

#### 8.4.1 None

The *ModellingRule None* is defined in [UA Part 3]. Its representation in the *AddressSpace* – the “None” *Object* – is formally defined in Table 69.

**Table 69 – None Definition**

Attribute	Value		
BrowseName	None		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	ModellingRuleType	Defined in Clause 6.5

#### 8.4.2 New

The *ModellingRule New* is defined in [UA Part 3]. Its representation in the *AddressSpace* – the “New” *Object* – is formally defined in Table 70.

**Table 70 – New Definition**

Attribute	Value		
BrowseName	New		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	ModellingRuleType	Defined in Clause 6.5

### 8.4.3 Shared

The *ModellingRule Shared* is defined in [UA Part 3]. Its representation in the *AddressSpace* – the “Shared” Object – is formally defined in Table 71.

**Table 71 – Shared Definition**

Attribute	Value		
BrowseName	Shared		
References	NodeClass	BrowseName	Comment
HasTypeDefinition	ObjectType	ModellingRuleType	Defined in Clause 6.5

## 9 Standard Methods

There are no core OPC UA *Methods* defined.

## 10 Standard ReferenceTypes

### 10.1 References

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 72.

**Table 72 – References ReferenceType**

Attributes	Value		
BrowseName	References		
InverseName	--		
Symmetric	True		
IsAbstract	True		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	HierarchicalReferences	Defined in Clause 10.2
HasSubtype	ReferenceType	NonHierarchicalReferences	Defined in Clause 10.3

### 10.2 HierarchicalReferences

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 73.

**Table 73 – HierarchicalReferences ReferenceType**

Attributes	Value		
BrowseName	HierarchicalReferences		
InverseName	--		
Symmetric	False		
IsAbstract	True		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	Aggregates	Defined in Clause 10.4
HasSubtype	ReferenceType	Organizes	Defined in Clause 10.5
HasSubtype	ReferenceType	HasEventSource	Defined in Clause 10.14

### 10.3 NonHierarchicalReferences

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 74.

**Table 74 – NonHierarchicalReferences ReferenceType**

Attributes	Value		
BrowseName	NonHierarchicalReferences		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	HasModellingRule	Defined in Clause 10.10
HasSubtype	ReferenceType	HasTypeDefinition	Defined in Clause 10.11
HasSubtype	ReferenceType	HasEncoding	Defined in Clause 10.12
HasSubtype	ReferenceType	HasDescription	Defined in Clause 10.13
HasSubtype	ReferenceType	GeneratesEvent	Defined in Clause 10.16
HasSubtype	ReferenceType	ExposesItsArray	Defined in Clause 10.17

### 10.4 Aggregates

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 75.

**Table 75 – Aggregates ReferenceType**

Attributes	Value		
BrowseName	Aggregates		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	HasComponent	Defined in Clause 10.6
HasSubtype	ReferenceType	HasProperty	Defined in Clause 10.8
HasSubtype	ReferenceType	HasSubtype	Defined in Clause 10.9

### 10.5 Organizes

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 76.

**Table 76 – Organizes ReferenceType**

Attributes	Value		
References	NodeClass	BrowseName	Comment
BrowseName	Organizes		
InverseName	OrganizedBy		
Symmetric	False		
IsAbstract	False		

## 10.6 HasComponent

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 77.

**Table 77 – HasComponent ReferenceType**

Attributes	Value		
BrowseName	HasComponent		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	HasOrderedComponent	Defined in Clause 10.7

## 10.7 HasOrderedComponent

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 78.

**Table 78 – HasOrderedComponent ReferenceType**

Attributes	Value		
BrowseName	HasOrderedComponent		
References	NodeClass	BrowseName	Comment

## 10.8 HasProperty

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 79.

**Table 79 – HasProperty ReferenceType**

Attributes	Value		
BrowseName	HasProperty		
References	NodeClass	BrowseName	Comment

## 10.9 HasSubtype

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 80.

**Table 80 – HasSubtype ReferenceType**

Attributes	Value		
BrowseName	HasSubtype		
References	NodeClass	BrowseName	Comment

## 10.10 HasModellingRule

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 81.

**Table 81 – HasModellingRule ReferenceType**

Attributes	Value		
BrowseName	HasModellingRule		
InverseName	ModellingRuleOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

## 10.11 HasTypeDefinition

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 82.

**Table 82 – HasTypeDefinition ReferenceType**

Attributes	Value		
BrowseName	HasTypeDefinition		
InverseName	TypeDefinitionOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

## 10.12 HasEncoding

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 82.

**Table 83 – HasEncoding ReferenceType**

Attributes	Value		
BrowseName	HasEncoding		
InverseName	EncodingOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

## 10.13 HasDescription

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 82.

**Table 84 – HasDescription ReferenceType**

Attributes	Value		
BrowseName	HasDescription		
InverseName	DescriptionOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

## 10.14 HasEventSource

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 85.

**Table 85 – HasEventSource ReferenceType**

Attributes	Value		
BrowseName	HasEventSource		
References	NodeClass	BrowseName	Comment
HasSubtype	ReferenceType	HasNotifier	Defined in Clause 10.15

## 10.15 HasNotifier

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 86.

**Table 86 – HasNotifier ReferenceType**

Attributes	Value		
BrowseName	HasNotifier		
References	NodeClass	BrowseName	Comment

## 10.16 GeneratesEvent

This standard *ReferenceType* is defined in [UA Part 3]. Its representation in the *AddressSpace* is specified in Table 87.

**Table 87 – GeneratesEvent ReferenceType**

Attributes	Value		
BrowseName	GeneratesEvent		
References	NodeClass	BrowseName	Comment

## 10.17 ExposesItsArray

The *ExposesItsArray ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* expresses special type information of *VariableTypes*. It can be used only by *VariableTypes* having an array. It expresses that the instances of the *VariableType* will expose the entries of the array as additional *Variables*.

The *SourceNode* of this *ReferenceType* must be a *VariableType* having its *ArraySize Attribute* set. The *TargetNode* of this *ReferenceType* must be a *VariableType*.

Each *Variable* "A" of the *VariableType* used as *SourceNode* must reference to a *Variable* of the *VariableType* used as *TargetNode* using a *HasComponent Reference* for each entry of the array of the *Variable* "A". The *ModellingRule* of the referenced *Variables* is *None* or not set.

Remark: Since the *Services* allow accessing single entries of an array, it makes sense to use this *ReferenceType* and expose the entries of an array as additional *Variables* only if those *Variables* are complex. Thus a client can access parts of an entry of the array and may need to access only simple, well-known *DataTypes*.

The representation of the *ExposeItsArray ReferenceType* in the *AddressSpace* is specified in Table 88.

**Table 88 – ExposesItsArray ReferenceType**

Attributes	Value		
BrowseName	ExposeItsArray		
References	NodeClass	BrowseName	Comment

## 11 Standard DataTypes

### 11.1 Overview

An OPC UA server need not expose its *DataTypes* in its *AddressSpace*. Independent of the exposition of *DataTypes*, it must support the *DataTypes* as described in the following subclauses. The *TypeEncodings*, the *TypeDescriptions* and the *TypeDictionaries* of the *DataTypes* and the *References* to them are specified in [UA Part 6].

## 11.2 DataTypes defined in [UA Part 3]

[UA Part 3] defines a set of *DataTypes*. Their representation in the *AddressSpace* is defined in Table 89.

**Table 89 – [UA Part 3] DataType Definitions**

BrowseName
BaseDataType
Argument
Boolean
Byte
ByteString
Date
Double
Float
Guid
IdType
SByte
Integer
Int16
Int32
Int64
LocaleId
LocalizedText
NodeId
Number
QualifiedName
String
Time
UInteger
UInt16
UInt32
UInt64
UtcTime
XmlElement

Of the *DataTypes* defined in Table 89 only the *BaseDataType*, the *Number*, the *Integer* and the *UInteger* *DataType* are the source of *References*. The *References* of the *BaseDataType* are defined in Table 90.

**Table 90 – BaseDataType Definition**

<b>Attributes</b>	<b>Value</b>	
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>
HasSubtype	DataType	Argument
HasSubtype	DataType	Boolean
HasSubtype	DataType	ByteString
HasSubtype	DataType	Date
HasSubtype	DataType	Double
HasSubtype	DataType	Float
HasSubtype	DataType	Guid
HasSubtype	DataType	IdType
HasSubtype	DataType	LocaleId
HasSubtype	DataType	LocalizedText
HasSubtype	DataType	NodeId
HasSubtype	DataType	Number
HasSubtype	DataType	QualifiedName
HasSubtype	DataType	String
HasSubtype	DataType	Time
HasSubtype	DataType	UtcTime
HasSubtype	DataType	XmlElement
HasSubtype	DataType	RedundancySupport
HasSubtype	DataType	ServerState
HasSubtype	DataType	BuildInfo
HasSubtype	DataType	DataValue
HasSubtype	DataType	RedundantServerDataType
HasSubtype	DataType	SamplingRateDiagnosticsDataType
HasSubtype	DataType	ServerDiagnosticsSummaryDataType
HasSubtype	DataType	ServerStatusDataType
HasSubtype	DataType	SessionDiagnosticsDataType
HasSubtype	DataType	SessionSecurityDiagnosticsDataType
HasSubtype	DataType	SubscriptionDiagnosticsDataType
HasSubtype	DataType	ServiceCounterDataType
HasSubtype	DataType	SignedSoftwareCertificate
HasSubtype	DataType	UserIdentityToken
HasSubtype	DataType	SecurityTokenRequestType
HasSubtype	DataType	AddNodesItem
HasSubtype	DataType	AddReferencesItem
HasSubtype	DataType	DeleteNodesItem
HasSubtype	DataType	DeleteReferencesItem
HasSubtype	DataType	NumericRange
HasSubtype	DataTypes	ChangeStructureDataType
HasSubtype	DataTypes	PropertyChangeStructureDataType

The *References* of *Number* are defined in Table 90.

**Table 91 – Number Definition**

<b>Attributes</b>	<b>Value</b>	
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>
HasSubtype	DataType	Integer
HasSubtype	DataType	UInteger
HasSubtype	DataType	Double
HasSubtype	DataType	Float

The *References* of *Integer* are defined in Table 90.

**Table 92 – Integer Definition**

Attributes	Value	
BrowseName	Integer	
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>
HasSubtype	DataType	SByte
HasSubtype	DataType	Int16
HasSubtype	DataType	Int32
HasSubtype	DataType	Int64

The *References* of *UInteger* are defined in Table 90.

**Table 93 – BaseDataType Definition**

Attributes	Value	
BrowseName	UInteger	
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>
HasSubtype	DataType	Byte
HasSubtype	DataType	UInt16
HasSubtype	DataType	UInt32
HasSubtype	DataType	UInt64

### 11.3 DataTypes defined in [UA Part 4]

[UA Part 4] defines a set of *DataTypes*. Their representation in the *AddressSpace* is defined in Table 94.

**Table 94 – [UA Part 4] DataType Definitions**

BrowseName
BuildInfo
WithValue
SignedSoftwareCertificate
UserIdentityToken
SecurityTokenRequestType
AddNodesItem
AddReferencesItem
DeleteNodesItem
DeleteReferencesItem
NumericRange

The *SecurityTokenRequestType* is an enumeration that is defined as the type of the *requestType* parameter of the *OpenSecureChannel Service* in [UA Part 4].

The *AddNodesItem* is a structure that is defined as the type of the *nodesToAdd* parameter of the *AddNodes Service* in [UA Part 4].

The *AddReferencesItem* is a structure that is defined as the type of the *referencesToAdd* parameter of the *AddReferences Service* in [UA Part 4].

The *DeleteNodesItem* is a structure that is defined as the type of the *nodesToDelete* parameter of the *DeleteNodes Service* in [UA Part 4].

The *DeleteReferencesItem* is a structure that is defined as the type of the *referencesToDelete* parameter of the *DeleteReferences Service* in [UA Part 4].

## 11.4 RedundancySupport

This *DataType* is an enumeration that defines the redundancy support of the server. Its values are defined in Table 95.

**Table 95 – RedundancySupport Values**

Numeric Value	String Value	Description
1	none	None means that there is no redundancy support.
2	cold	Cold means that the redundant servers are operational, but do not have any subscriptions defined and do not accept requests to create one.
3	warm	Warm means that the redundant servers have redundant subscriptions, but with sampling disabled.
4	hot	Hot means that the redundant servers have redundant subscriptions with sampling enabled, but not reporting.

See [UA Part 1] for a more detailed description of the different values.

Its representation in the *AddressSpace* is defined in Table 96.

**Table 96 – RedundancySupport Definition**

Attributes	Value
BrowseName	RedundancySupport

## 11.5 ServerState

This *DataType* is an enumeration that defines the execution state of the server. Its values are defined in Table 97.

**Table 97 – ServerState Values**

Numeric Value	String Value	Description
1	Running	The server is running normally. This is the usual state for a server.
2	Failed	A vendor-specific fatal error has occurred within the server. The server is no longer functioning. The recovery procedure from this situation is vendor-specific. Most Service requests should be expected to fail.
3	NoConfiguration	The server is running but has no configuration information loaded and therefore does not transfer data.
4	Suspended	The server has been temporarily suspended by some vendor-specific method and is not receiving or sending data.
5	Shutdown	The server has shut down. Depending on the implementation, this may or may not be visible to clients.
6	Test	The server is in Test Mode. The outputs are disconnected from the real hardware, but the server will otherwise behave normally. Inputs may be real or may be simulated depending on the vendor implementation. StatusCode will generally be returned normally.
7	CommunicationFault	The server is running properly, but is having difficulty accessing data from its data sources. This may be due to communication problems or some other problem preventing the underlying device, control system, etc. from returning valid data. It may be a complete failure, meaning that no data is available, or a partial failure, meaning that some data is still available. It is expected that items affected by the fault will individually return with a BAD FAILURE status code indication for the items.
8	Unknown	This state is used only to indicate that the UA server does not know the state of underlying servers.

Its representation in the *AddressSpace* is defined in Table 98.

**Table 98 – ServerState Definition**

Attributes	Value
BrowseName	ServerState

### 11.6 RedundantServerDataType

This structure contains elements that describe the status of the server. Its composition is defined in Table 99.

**Table 99 – RedundantServerDataType Structure**

Name	Type	Description
RedundantServerDataType	structure	
serverId	String	The Id of the server (not the URI).
serviceLevel	SByte	The service level of the server
serverState	ServerState	The current state of the server.

Its representation in the *AddressSpace* is defined in Table 100.

**Table 100 – RedundantServerDataType Definition**

Attributes	Value
BrowseName	RedundantServerDataType

### 11.7 SamplingRateDiagnosticsDataType

This structure contains diagnostic information about the sampling rates supported by the server. Its elements are defined in Table 101.

**Table 101 – SamplingRateDiagnosticsDataType Structure**

Name	Type	Description
SamplingRateDiagnosticsDataType	structure	
samplingRate	UInt32	The sampling rate in milliseconds
samplingErrorCount	UInt32	The number of times access to a <i>MonitoredItem</i> at this sample rate failed since the server was started (restarted).
sampledMonitoredItemsCount	UInt32	The number of <i>MonitoredItems</i> being sampled at this sample rate.
maxSampledMonitoredItemsCount	UInt32	The maximum number of <i>MonitoredItems</i> being sampled at this sample rate at the same time since the server was started (restarted).
disabledMonitoredItemsSamplingCount	UInt32	The number of <i>MonitoredItems</i> at this sample rate whose sampling currently disabled.

Its representation in the *AddressSpace* is defined in Table 102.

**Table 102 – SamplingRateDiagnosticsDataType Definition**

Attributes	Value
BrowseName	SamplingRateDiagnosticsDataType

## 11.8 ServerDiagnosticsSummaryDataType

This structure contains diagnostic summary information for the server. Its elements are defined in Table 103.

**Table 103 – ServerDiagnosticsSummaryDataType Structure**

Name	Type	Description
ServerDiagnosticsSummaryDataType	structure	
serverViewCount	UInt32	The number of server-created views in the server.
currentSessionCount	UInt32	The number of client sessions currently established in the server.
cumulatedSessionCount	UInt32	The cumulative number of client sessions that have been established in the server since the server was started (or restarted). This includes the <i>currentSessionCount</i> .
securityRejectedSessionCount	UInt32	The number of client session establishment requests that were rejected due to security constraints since the server was started (or restarted).
rejectSessionCount	UInt32	The number of client session establishment requests that were rejected since the server was started (or restarted). This number includes the <i>securityRejectedSessionCount</i> .
sessionTimeoutCount	UInt32	The number of client sessions that were closed due to timeout since the server was started (or restarted).
sessionAbortCount	UInt32	The number of client sessions that were closed due to errors since the server was started (or restarted).
samplingRateCount	UInt32	The number of sampling rates currently supported in the server.
publishingRateCount	UInt32	The number of publishing rates currently supported in the server.
currentSubscriptionCount	UInt32	The number of subscriptions currently established in the server.
cumulatedSubscriptionCount	UInt32	The cumulative number of subscriptions that have been established in the server since the server was started (or restarted). This includes the <i>currentSubscriptionCount</i> .
securityRejectedRequestsCount	UInt32	The number of requests that were rejected due to security constraints since the server was started (or restarted). The requests include all Services defined in [UA Part 4], also requests to create sessions.
rejectedRequestsCount	UInt32	The number of requests that were rejected since the server was started (or restarted). The requests include all Services defined in [UA Part 4], also requests to create sessions. This number includes the <i>securityRejectedRequestsCount</i> .

Its representation in the *AddressSpace* is defined in Table 104.

**Table 104 – ServerDiagnosticsSummaryDataType Definition**

Attributes	Value
BrowseName	ServerDiagnosticsSummaryDataType

## 11.9 ServerStatusDataType

This structure contains elements that describe the status of the server. Its composition is defined in Table 105.

**Table 105 – ServerStatusDataType Structure**

Name	Type	Description
ServerStatusDataType	structure	
startTime	UtcTime	Time (UTC) the server was started. This is constant for the server instance and is not reset when the server changes state. Each instance of a server should keep the time when the process started.
currentTime	UtcTime	The current time (UTC) as known by the server.
state	ServerState	The current state of the server. Its values are defined in Clause 11.5.
buildInfo	BuildInfo	

Its representation in the *AddressSpace* is defined in Table 106.

**Table 106 – ServerStatusDataType Definition**

Attributes	Value
BrowseName	ServerStatusDataType

### 11.10 SessionDiagnosticsDataType

This structure contains diagnostic information about client sessions. Its elements are defined in Table 107. Most of the values represented in this structure provide information about the number of calls of a *Service*, the number of currently used *MonitoredItems*, etc. Those numbers need not provide the exact value; they need only provide the approximate number, so that the server is not burdened with providing the exact numbers.

**Table 107 – SessionDiagnosticsDataType Structure**

Name	Type	Description
SessionDiagnosticsDataType	structure	
sessionId	Int32	Server-assigned identifier of the session.
clientName	string	The name of the client provided in the open session request.
localeIds	LocaleId[]	Array of LocaleIds specified by the client in the open session call.
requestedSessionTimeout	Int32	The requested session timeout specified by the client in the open session call.
clientConnectionTime	UtcTime	The server timestamp when the client opens the session.
clientLastContactTime	UtcTime	The server timestamp of the last request of the client in the context of the session.
currentSubscriptionsCount	UInt32	The number of subscriptions currently used by the session.
currentMonitoredItemsCount	UInt32	The number of <i>MonitoredItems</i> currently used by the session
currentPublishRequestsInQueue	UInt32	The number of publish requests currently in the queue for the session.
currentPublishTimerExpirations	UInt32	The number of publish timer expirations when there are data to be sent, but there are no publish requests for this session. The value must be 0 if there are no data to be sent or publish requests queued.
keepAliveCount	UInt32	Number of publish responds sent from the server without data for this session.
currentRepublishRequestsInQueue	UInt32	The number of republish requests currently in the queue for the session.
maxRepublishRequestsInQueue	UInt32	Maximum number of republish requests in the queue for the session.
republishCounter	UInt32	Number of republish requests for the session, including <i>currentRepublishRequestsInQueue</i> .
publishingCount	UInt32	The number of Notifications that have been published for the session.
publishingQueueOverflowCount	UInt32	The number of times an overflow condition occurred for the publishing queue of a <i>MonitoredItem</i> Property. Overflow behaviour is defined by the Queue Model of a <i>MonitoredItem</i> , as specified for the Subscription Service Set in [UA Part 4].
readCount	ServiceCounter DataType	Counter of the Read Service, identifying the number of received requests of this Service on the session.
historyReadCount	ServiceCounter DataType	Counter of the HistoryRead Service, identifying the number of received requests of this Service on the session.
writeCount	ServiceCounter DataType	Counter of the Write Service, identifying the number of received requests of this Service on the session.
historyUpdateCount	ServiceCounter DataType	Counter of the HistoryUpdate Service, identifying the number of received requests of this Service on the session.
methodCallCount	ServiceCounter DataType	Counter of the Call Service, identifying the number of received requests of this Service on the session.
createMonitoredItemCount	ServiceCounter DataType	Counter of the CreateMonitoredItem Service, identifying the number of received requests of this Service on the session.
modifyMonitoredItemCount	ServiceCounter DataType	Counter of the ModifyMonitoredItem Service, identifying the number of received requests of this Service on the session.
setMonitoringModeCount	ServiceCounter DataType	Counter of the SetMonitoringMode Service, identifying the number of received requests of this Service on the session.
setTriggeringCount	ServiceCounter DataType	Counter of the SetTriggering Service, identifying the number of received requests of this Service on the session.
deleteMonitoredItemsCount	ServiceCounter DataType	Counter of the DeleteMonitoredItems Service, identifying the number of received requests of this Service on the session.
createSubscriptionCount	ServiceCounter	Counter of the CreateSubscription Service, identifying the number of

	DataType	received requests of this Service on the session.
modifySubscriptionCount	ServiceCounter DataType	Counter of the ModifySubscription Service, identifying the number of received requests of this Service on the session.
setPublishingModeCount	ServiceCounter DataType	Counter of the SetPublishingMode Service, identifying the number of received requests of this Service on the session.
publishCount	ServiceCounter DataType	Counter of the Publish Service, identifying the number of received requests of this Service on the session.
republishCount	ServiceCounter DataType	Counter of the Republish Service, identifying the number of received requests of this Service on the session.
transferSubscriptionsCount	ServiceCounter DataType	Counter of the TransferSubscriptions Service, identifying the number of received requests of this Service on the session.
deleteSubscriptionsCount	ServiceCounter DataType	Counter of the DeleteSubscriptions Service, identifying the number of received requests of this Service on the session.
addNodesCount	ServiceCounter DataType	Counter of the AddNodes Service, identifying the number of received requests of this Service on the session.
addReferencesCount	ServiceCounter DataType	Counter of the AddReferences Service, identifying the number of received requests of this Service on the session.
deleteNodesCount	ServiceCounter DataType	Counter of the DeleteNodes Service, identifying the number of received requests of this Service on the session.
deleteReferencesCount	ServiceCounter DataType	Counter of the DeleteReferences Service, identifying the number of received requests of this Service on the session.
browseCount	ServiceCounter DataType	Counter of the Browse Service, identifying the number of received requests of this Service on the session.
browseNextCount	ServiceCounter DataType	Counter of the BrowseNext Service, identifying the number of received requests of this Service on the session.
translateBrowsePathsToNodeIdsCount	ServiceCounter DataType	Counter of the TranslateBrowsePathsToNodeIds Service, identifying the number of received requests of this Service on the session.
queryFirstCount	ServiceCounter DataType	Counter of the QueryFirst Service, identifying the number of received requests of this Service on the session.
queryNextCount	ServiceCounter DataType	Counter of the QueryNext Service, identifying the number of received requests of this Service on the session.

Its representation in the AddressSpace is defined in Table 108.

**Table 108 – SessionDiagnosticsDataType Definition**

Attributes	Value
BrowseName	SessionDiagnosticsDataType

### 11.11 SessionSecurityDiagnosticsDataType

This structure contains security-related diagnostic information about client sessions. Its elements are defined in Table 109. Because this information is security-related, it should not be made accessible to all users, but only to authorised users.

**Table 109 – SessionSecurityDiagnosticsDataType Structure**

Name	Type	Description
SessionSecurityDiagnosticsDataType	structure	
sessionId	Int32	Server-assigned identifier of the session.
clientUserIdOfSession	String	Name of authenticated user when creating the session
clientUserIdHistory	String[]	<p>Array containing the name of the authenticated user currently active (either from creating the session or from calling the ImpersonateUser Service) and the history of those names. Each time the active user changes, an entry must be made at the end of the array. The active user is always at the end of the array. Servers may restrict the size of this array, but must support at least a size of 2.</p> <p>How the name of the authenticated user can be obtained from the system via the information received as part of the session establishment is defined in Clause 6.4.3.</p>
authenticationMechanism	String	Type of authentication (user name and password, X.509, Kerberos).
encoding	String	Which encoding is used on the wire, e.g. XML or UA Binary.
transportProtocol	String	Which transport protocol is used, e.g. TCP or HTTP.
securityPolicy	String	The name of the security policy used for the session.

Its representation in the *AddressSpace* is defined in Table 110.

**Table 110 – SessionSecurityDiagnosticsDataType Definition**

Attributes	Value
BrowseName	SessionSecurityDiagnosticsDataType

### 11.12 ServiceCounterDataType

This structure contains diagnostic information about subscriptions. Its elements are defined in Table 111.

**Table 111 – ServiceCounterDataType Structure**

Name	Type	Description
ServiceCounterDataType	structure	
totalCount	UInt32	The number of Service requests that have been received.
unauthCount	UInt32	The number of Service requests that were rejected due to authorization failure.
errorCount	UInt32	The total number of Service requests that were rejected. This number includes the <i>unauthCount</i> .

Its representation in the *AddressSpace* is defined in Table 112.

**Table 112 – ServiceCounterDataType Definition**

Attributes	Value
BrowseName	ServiceCounterDataType

### 11.13 SubscriptionDiagnosticsDataType

This structure contains diagnostic information about subscriptions. Its elements are defined in Table 113.

**Table 113 – SubscriptionDiagnosticsDataType Structure**

Name	Type	Description
SubscriptionDiagnosticsDataType	structure	
sessionId	Int32	Server-assigned identifier of the session the subscription belongs to.
subscriptionId	Int32	Server-assigned identifier of the subscription.
priority	Byte	The priority the client assigned to the subscription.
publishingRate	UInt32	The publishing rate of the subscription in milliseconds
modifyCount	UInt32	The number of ModifySubscription requests received for the subscription.
enableCount	UInt32	The number of times the subscription has been enabled.
disableCount	UInt32	The number of times the subscription has been disabled.
republishRequestCount	UInt32	The number of Republish Service requests that have been received and processed for the subscription.
republishMsgRequestCount	UInt32	The total number of messages that have been requested to be republished for the subscription
republishMessageCount	UInt32	The number of messages that have been successfully republished for the subscription.
transferRequestCount	UInt32	The total number of TransferSubscriptions Service requests that have been received for the subscription.
transferredToAltClientCount	UInt32	The number of times the subscription has been transferred to an alternate client.
transferredToSameClientCount	UInt32	The number of times the subscription has been transferred to an alternate session for the same client.
publishRequestCount	UInt32	The number of Publish Service requests that have been received and processed for the subscription.
dataChangeNotificationsCount	UInt32	The number of data change Notifications sent by the subscription.
eventNotificationsCount	UInt32	The number of Event Notifications sent by the subscription.
notificationsCount	UInt32	The total number of Notifications sent by the subscription.
lateStateCount	UInt32	The number of times the subscription has entered the LATE State.
keepAliveStateCount	UInt32	The number of times the subscription has entered the KEEPALIVE State.

Its representation in the *AddressSpace* is defined in Table 114.

**Table 114 – SubscriptionDiagnosticsDataType Definition**

Attributes	Value
BrowseName	SubscriptionDiagnosticsDataType

### 11.14 ChangeStructureDataType

This structure contains elements that describe a change of the model. Its composition is defined in Table 115.

**Table 115 – ChangeStructureDataType Structure**

Name	Type	Description												
ChangeStructureDataType	structure													
affected	NodeId	<i>NodeId</i> of the <i>Node</i> that was changed. The client should assume that the <i>affected Node</i> has been created or deleted, had a <i>Reference</i> added or deleted, or the <i>Data Type</i> has changed as described by the <i>verb</i> .												
affectedType	NodeId	If the <i>affected Node</i> was an <i>Object</i> or <i>Variable</i> , <i>affectedType</i> contains the <i>NodeId</i> of the <i>TypeDefinitionNode</i> of the <i>affected Node</i> . Otherwise it is set to null.												
verb	enum	<p>Describes the change happening to the affected <i>Node</i>.</p> <table border="1"> <thead> <tr> <th>String Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>NodeAdded</td> <td>Indicates the <i>affected Node</i> has been added.</td> </tr> <tr> <td>NodeDeleted</td> <td>Indicates the <i>affected Node</i> has been deleted.</td> </tr> <tr> <td>ReferenceAdded</td> <td>Indicates a <i>Reference</i> has been added. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i>. Note that an added bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i>.</td> </tr> <tr> <td>ReferenceDeleted</td> <td>Indicates a <i>Reference</i> has been deleted. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i>. Note that a deleted bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i>.</td> </tr> <tr> <td>DataTypeChanged</td> <td>This verb may be used only for affected <i>Nodes</i> that are <i>Variables</i> or <i>VariableTypes</i>. It indicates that the <i>Data Type</i> Attribute has changed.</td> </tr> </tbody> </table> <p>Note that all <i>verbs</i> must always be considered in the context where the <i>ChangeStructureDataType</i> is used. A <i>NodeDeleted</i> may indicate that a <i>Node</i> was removed from a <i>view</i> but still exists in other <i>Views</i>.</p>	String Value	Description	NodeAdded	Indicates the <i>affected Node</i> has been added.	NodeDeleted	Indicates the <i>affected Node</i> has been deleted.	ReferenceAdded	Indicates a <i>Reference</i> has been added. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i> . Note that an added bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i> .	ReferenceDeleted	Indicates a <i>Reference</i> has been deleted. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i> . Note that a deleted bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i> .	DataTypeChanged	This verb may be used only for affected <i>Nodes</i> that are <i>Variables</i> or <i>VariableTypes</i> . It indicates that the <i>Data Type</i> Attribute has changed.
String Value	Description													
NodeAdded	Indicates the <i>affected Node</i> has been added.													
NodeDeleted	Indicates the <i>affected Node</i> has been deleted.													
ReferenceAdded	Indicates a <i>Reference</i> has been added. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i> . Note that an added bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i> .													
ReferenceDeleted	Indicates a <i>Reference</i> has been deleted. The <i>affected Node</i> may be either a <i>SourceNode</i> or <i>TargetNode</i> . Note that a deleted bidirectional <i>Reference</i> is reflected by two <i>ChangeStructures</i> .													
DataTypeChanged	This verb may be used only for affected <i>Nodes</i> that are <i>Variables</i> or <i>VariableTypes</i> . It indicates that the <i>Data Type</i> Attribute has changed.													

Its representation in the *AddressSpace* is defined in Table 106.

**Table 116 – ChangeStructureDataType Definition**

Attributes	Value
BrowseName	ChangeStructureDataType

### 11.15 PropertyChangeStructureDataType

This structure contains elements that describe a change of the model. Its composition is defined in Table 117.

**Table 117 – PropertyChangeStructureDataType Structure**

Name	Type	Description
PropertyChangeStructureDataType	structure	
affected	NodeId	<i>NodeId</i> of the <i>Node</i> that owns the <i>Property</i> that has changed. The client should assume that the <i>affected Node</i> has been created or deleted, had a <i>Reference</i> added or deleted, or the <i>DataType</i> has changed as described by the <i>verb</i> .
affectedType	NodeId	If the <i>affected Node</i> was an <i>Object</i> or <i>Variable</i> , <i>affectedType</i> contains the <i>NodeId</i> of the <i>TypeDefinitionNode</i> of the <i>affected Node</i> . Otherwise it is set to null.

Its representation in the *AddressSpace* is defined in Table 106.

**Table 118 – PropertyChangeStructureDataType Definition**

Attributes	Value
BrowseName	PropertyChangeStructureDataType

## Appendix A: Design decisions when modelling the server information

### A.1 Overview

This Appendix describes the design decisions of modelling the information provided by each OPC UA server, exposing its capabilities, diagnostic information, and other data needed to work with the server, such as the *NamespaceArray*.

This Appendix gives an example of what should be considered when modelling data using the Address Space Model. General considerations for using the Address Space Model can be found in Appendix A of [UA Part 3].

This Appendix is informative, that is each server vendor can model its data in the appropriate way that fits its needs.

The following subclauses describe the design decisions made while modelling the *Server Object*. General *DataTypes*, *VariableTypes* and *ObjectTypes* such as the *EventTypes* described in this Part are not taken into account.

### A.2 ServerType and Server Object

The first decision is to decide at what level types are needed. Typically, each server will provide one *Server Object* with a well known *NodeId*. The *NodeIds* of the containing *Nodes* are also well-known because their symbolic name is specified in this part and the *NodeId* is based on the symbolic name in [UA Part 6]. Nevertheless, aggregating servers may want to expose the *Server Objects* of the OPC UA servers they are aggregating in their *AddressSpace*. Therefore, it is very helpful to have a type definition for the *Server Object*. The *Server Object* is an *Object*, because it groups a set of *Variables* and *Objects* containing information about the server. The *ServerType* is a complex *ObjectType*, because the basic structure of the *Server Object* should be well-defined. However, the *Server Object* can be extended by adding *Variables* and *Objects* in an appropriate structure of the *Server Object* or its containing *Objects*.

### A.3 Typed complex Objects beneath the Server Object

*Objects* beneath the *Server Object* used to group information, such as server capabilities or diagnostics, are also typed because an aggregating server may want to provide only part of the server information, such as diagnostics information, in its *AddressSpace*. Clients are able to program against these structures if they are typed, because they have its type definition.

### A.4 Properties vs. DataVariables

Since the general description in [UA Part 3] about the semantic difference between *Properties* and *DataVariables* are not applicable for the information provided about the server the rules described in Clause A.4.2 of [UA Part 3] are used.

If simple data structures should be provided, *Properties* are used. Examples of *Properties* are the *NamespaceArray* of the *Server Object* and the *MinSupportedSampleRate* of the *ServerCapabilities Object*.

If complex data structures are used, *DataVariables* are used. Examples of *DataVariables* are the *ServerStatus* of the *Server Object* and the *ServerDiagnosticsSummary* of the *ServerDiagnostics Object*.

## A.5 Complex Variables using complex DataTypes

*DataVariables* providing complex data structures expose their information as complex *DataTypes*, as well as components in the *AddressSpace*. This allows access to simple values as well as access to the whole information at once in a transactional context.

For example, the *ServerStatus Variable* of the *Server Object* is modelled as a complex *DataVariable* having the *ServerStatusDataType* providing all information about the server status. But it also exposes the *CurrentTime* as a simple *DataVariable*, because a client may want to read only the current time of the server, and is not interested in the build information, etc.

## A.6 Complex Variables having an array

A special case of providing complex data structures is an array of complex data structures. The *SubscriptionDiagnosticsArrayType* is an example of how this is modelled. It is an array of a complex data structure, providing information of a subscription. Because a server typically has several subscriptions, it is an array. Some clients may want to read the diagnostic information about all subscriptions at once; therefore it is modelled as an array in a *Variable*. On the other hand, a client may be interested in only a single entry of the complex structure, such as the *PublishRequestCount*. Therefore, each entry of the array is also exposed individually as a complex *DataVariable*, having each entry exposed as simple data.

Note that it is never necessary to expose the individual entries of an array to access them separately. The Services already allow accessing individual entries of an array of a *Variable*. However, if the entries should also be used for other purposes in the *AddressSpace* – such as having *References* or additional *Properties* or exposing their complex structure using *DataVariables* – it is useful to expose them individually.

## A.7 Adding ReferenceTypes

In this part, one *ReferenceType* was added to the *ReferenceTypes* defined in [UA Part 3]: the *ExposesItsArray*. It is used in the type definition of *VariableTypes* to indicate that *Variables* of this type will expose each array entry individually. It was necessary to add this *ReferenceType* because all *ReferenceTypes* defined in [UA Part 3] did not provide this ability. The *ReferenceType* is used in the *SubscriptionDiagnosticsArrayType*.

Note that the *ExposesItsArray ReferenceType* is a standard *ReferenceType*, because it is defined in this Part. It was not defined in [UA Part 3], because exposing entries of an array individually is not a general concept needed to build a useful *AddressSpace*.

## A.8 Redundant information

Providing redundant information should generally be avoided. But to fulfil the needs of different clients, it may be helpful.

Using complex *DataVariables* automatically leads to providing redundant information, because the information is directly provided in the complex *DataType* of the *Value Attribute* of the complex *Variable*, and also exposed individually in the components of the complex *Variable*.

The diagnostics information about subscriptions is provided in two different locations. One location is the *SubscriptionDiagnosticsArray* of the *ServerDiagnostics Object*, providing the information for all subscriptions of the server. The second location is the *SubscriptionDiagnosticsArray* of each individual *SessionDiagnosticsObject Object*, providing only the subscriptions of the session. This is useful because some clients may be interested in only the subscriptions grouped by sessions, whereas other clients may want to access the diagnostics information of all sessions at once.

The *SessionDiagnosticsArray* and the *SessionSecurityDiagnosticsArray* of the *SessionsDiagnosticsSummary Object* do not expose their individual entries, although they represent an array of complex data structures. But the information of the entries can also be accessed individually as components of the *SessionDiagnostics Objects* provided for each session by the *SessionsDiagnosticsSummary Object*. A client can either access the arrays (or parts of the arrays) directly or browse to the *SessionDiagnostics Objects* to get the information of the individual entries. Thus, the information provided is redundant, but the *Variables* containing the arrays do not expose their individual entries.

## A.9 Usage of the *BaseDataVariableType*

All *DataVariables* used to expose complex data structures of complex *DataVariables* have the *BaseDataVariableType* as type definition if they are not complex by themselves. The reason for this approach is that the complex *DataVariables* already define the semantic of the containing *DataVariables* and this semantic is not used in another context. It is not expected that they are subtyped, because they should reflect the data structure of the *DataType* of the complex *DataVariable*.

## A.10 Subtyping

Subtyping is used for modelling information about the redundancy support of the server. Because the provided information must differ depending on the supported redundancy of the server, subtypes of the *ServerRedundancyType* will be used for this purpose.

Subtyping is also used as an extensibility mechanism (see next Clause).

## A.11 Extensibility mechanism

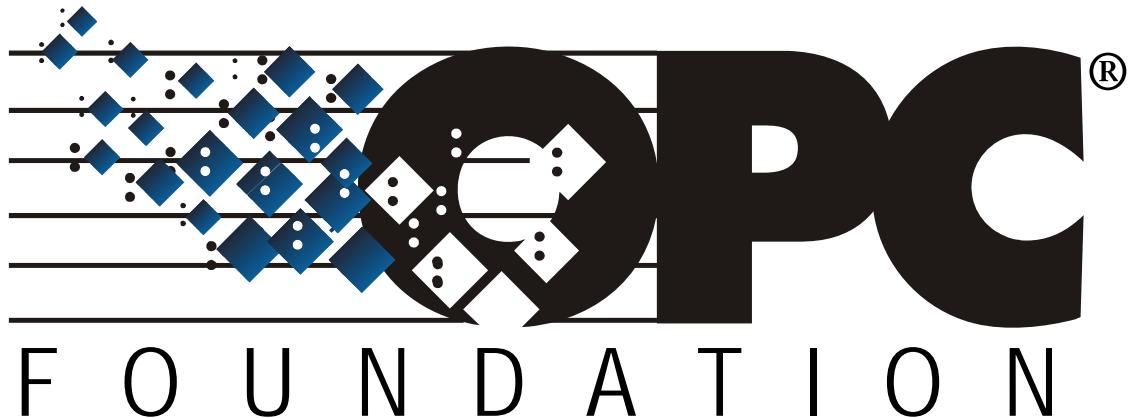
The information of the server will be extended by other parts of this multi-part specification, by companion specifications or by server vendors. There are preferred ways to provide the additional information.

Do not subtype *DataTypes* to provide additional information about the server. Clients may not be able to read those new defined *DataTypes* and are not able to get the information – including the basic information. If information is added by several sources, the *DataType* hierarchy may be difficult to maintain. Note that this rule applies to the information about the server; in other scenarios this may be a useful way to add information.

Add *Objects* containing *Variables* or add *Variables* to the *Objects* defined in this part. If, for example, additional diagnostic information per subscription is needed, add a new *Variable* containing in array with an entry per subscription in the same places that the *SubscriptionDiagnosticsArray* is used.

Use subtypes of the *ServerVendorCapabilityType* to add information about the server-specific capabilities on the *ServerCapabilities Objects*. Because this extensibility point is already defined in this part, clients will look there for additional information.

Use a subtype of the *VendorServerInfoType* to add server-specific information. Because an *Object* of this type is already defined in this part, clients will look there for server-specific information.



## **OPC Unified Architecture**

### **Release Candidate Specification**

#### **Part 6: Mappings**

**Version 0.93**

**June 1, 2006**

**Send comments to:**  
**[UACcomments@opcfoundation.org](mailto:UACcomments@opcfoundation.org)**



Specification Type	Industry Standard Specification	Comments:
Title:	OPC Unified Architecture Part 6 Mappings	Date: June 1, 2006
Version:	<u>Release Candidate 0.93</u>	Software Source: MS-Word OPC UA Part 6 - Mappings RC0.93 Specification.doc
Author:	OPC Foundation	Status: Release Candidate

## CONTENTS

	Page
1 Scope .....	1
2 Reference Documents.....	1
3 Terms, definitions, and conventions.....	2
3.1 OPC UA Part 1 terms .....	2
3.2 OPC UA Part 2 terms .....	3
3.3 OPC UA Mappings terms.....	3
3.3.1 Communication Stack.....	3
3.3.2 Mapping .....	3
3.3.3 Encoding.....	3
3.3.4 Transport Protocol.....	4
3.3.5 Programmers' Interface .....	4
3.3.6 Certificate Store .....	4
3.3.7 Certificate Trust List .....	4
3.3.8 Certificate Revocation List .....	4
3.3.9 Certification Authority .....	4
3.4 Abbreviations and symbols.....	4
4 Overview .....	4
4.1 The Mapping Process .....	4
4.2 Communication Stacks.....	5
4.3 Message Encoding.....	5
4.4 Formal Interface Definition .....	5
4.5 Certificate Management .....	6
4.6 Server Discovery.....	6
5 Mappings.....	6
5.1 General .....	6
5.2 UA Native Mapping .....	6
5.2.1 General.....	6
5.2.2 Serialization Layer.....	8
5.2.3 Secure Channel Layer .....	9
5.2.4 Transport Layer.....	14
5.3 XML Web Services Mapping.....	19
5.3.1 General.....	19
5.3.2 SOAP.....	20
5.3.3 HTTP .....	21
5.3.4 WS Addressing.....	21
5.3.5 WS Security .....	22
5.3.6 WS Trust.....	23
5.3.7 WS Secure Conversation.....	23
6 Message Encoding.....	25
6.1.1 Built-in Types .....	25
6.2 UA Binary .....	26
6.2.1 General.....	26
6.2.2 Built-in Types .....	26
6.2.3 Enumerations .....	34
6.2.4 Arrays .....	34
6.2.5 Simple Types .....	34
6.2.6 Structures .....	34

6.2.7	Messages.....	35
6.3	XML.....	35
6.3.1	Built-in Types .....	35
6.3.2	Enumerations .....	41
6.3.3	Arrays .....	41
6.3.4	Structures .....	41
6.3.5	Messages.....	42
7	Formal Interface Definition .....	42
7.1	General .....	42
7.2	Mapping File .....	42
7.3	UA Native Mapping .....	42
7.4	XML Web Service Mapping .....	42
8	Managing Certificates .....	42
8.1	Certificate Stores .....	42
8.2	Application Instance Certificates.....	43
8.2.1	Installation .....	43
8.2.2	Certificate Trust Lists .....	43
8.2.3	Certificate Revocation Lists .....	43
8.2.4	Proof of Possession.....	43
8.3	Software Certificates.....	44
8.3.1	General.....	44
8.3.2	Verification.....	44
9	Server Discovery .....	44
9.1	Overview .....	44
9.2	WS-Inspection .....	44
9.3	WS-Discovery .....	45
9.4	UDDI .....	45
Annex A	Constants.....	46
A.1	Service Type Ids .....	46
Annex B	Type Declarations for the UA Native Mapping .....	47
Annex C	WSDL for the UA XML Mapping .....	48
Annex D	Mapping File Syntax .....	49
D.1	General .....	49
D.2	Dictionary .....	49
D.3	Abstract Type .....	50
D.4	Simple Type.....	50
D.5	Complex Type.....	51
D.6	Enumerated Type.....	52
D.7	Service Type.....	53

**FIGURES**

Figure 1 – The UA Native Mapping Stack Overview.....	7
Figure 2 – UA Native Mapping Message Processing.....	8
Figure 3 – Establishing a Secure Channel.....	11
Figure 4 – Renewing Security Tokens.....	12
Figure 5 – UA TCP Normal Message Sequences.....	15
Figure 6 – UA TCP Error Recovery Sequences .....	17
Figure 7 – The XML Web Services Stack .....	19
Figure 8 – Encoding Integers in a Binary Stream.....	27
Figure 9 – Encoding Floating Points in a Binary Stream .....	27
Figure 10 – Encoding Strings in a Binary Stream .....	28

**TABLES**

Table 1 – UA Native Mapping Message Header .....	8
Table 2 – UA Native Mapping Message Footer.....	9
Table 3 – Security Header Components.....	9
Table 4 – Security Footer Components .....	10
Table 5 – Security Algorithms .....	10
Table 6 – UA TCP Message Types .....	14
Table 7 – UA TCP Hello Message.....	16
Table 8 – UA TCP Acknowledge Message .....	16
Table 9 – UA TCP Disconnect Message.....	17
Table 10 – UA TCP Data Message .....	18
Table 11 – UA Abort Message .....	18
Table 12 – UA Error Message.....	18
Table 13 – WS-Addressing Headers .....	21
Table 14 – WS-Trust User Token Policy.....	23
Table 15 – WS-* Namespace Prefixes .....	23
Table 16 – ChannelSecurityToken to SCT Mapping.....	24
Table 17 – <i>RequestSecurityToken</i> Message Components .....	24
Table 18 – Security Algorithm Mappings .....	25
Table 19 – <i>RequestSecurityTokenResponse</i> Message Components .....	25
Table 20 – Built-in Data Types.....	26
Table 21 – Supported Floating Point Types.....	27
Table 22 – NodId Components.....	29
Table 23 – NodId Encoding Values .....	29
Table 24 – Two Byte NodId Binary Encoding.....	29
Table 25 – Four Byte NodId Binary Encoding .....	29
Table 26 – Uri NodId Binary Encoding .....	30
Table 27 – Guid NodId Binary Encoding.....	30
Table 28 – Standard NodId Binary Encoding .....	30
Table 29 – DiagnosticInfo Binary Encoding .....	31
Table 30 – Extension Object Binary Encoding .....	32
Table 31 – Variant Binary Encoding .....	32
Table 32 – Variant Encoding Masks .....	33
Table 33 – Data Value Binary Encoding .....	33
Table 34 – Simple Type Mapping.....	34
Table 35 – Sample UA Binary Encoded Structure.....	35
Table 36 – XML Datatype Mappings for Integers .....	36
Table 37 – XML Datatype Mappings for Floating Points.....	36
Table 38 – String Format for Guid Components.....	37
Table 39 – Components of NodId URI.....	38
Table 40 – Components of Enumeration .....	41
Table 41 – Certificate Store Locations .....	43
Table 42 – Dictionary Element Components.....	50

Table 43 – Abstract Type Element Components .....	50
Table 44 – Simple Type Components.....	51
Table 45 – Complex Type Components.....	52
Table 46 – Field Type Components .....	52
Table 47 – Enumerated Type Components.....	53
Table 48 – Service Type Components.....	54

## OPC FOUNDATION

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

#### Trademarks

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### NON-EXCLUSIVE LICENSE AGREEMENT

The OPC Foundation, a non-profit corporation (the “OPC Foundation”), has defined a set of standard objects, interfaces and behaviours associated with the objects intended to promote interoperability between automation/control applications, field systems/devices, and business/office applications in the process control industry.

The OPC specifications, sample software that demonstrates the implementation of the specifications, standard interface components deliverables and related documentation (collectively, the “OPC Materials”), form a set of standard objects, interfaces and behaviour that are based on the technology being used in the automation marketplace, and includes the use of Microsoft Technology as well providing interoperability to non Microsoft platforms. The technology defines standard objects, methods, and properties for servers of real-time information like distributed process systems, programmable logic controllers, smart field devices and analyzers in order to communicate the information that such servers contain to standard compliant technologies enabled devices (e.g., servers, applications, etc.).

The OPC Foundation will grant to you (the “User”), whether an individual or legal entity, a license to use, and provide User with a copy of, the current version of the OPC Materials so long as User abides by the terms contained in this Non-Exclusive License Agreement (“Agreement”). If User does not agree to the terms and conditions contained in this Agreement, the OPC Materials may not be used, and all copies (in all formats) of such materials in User’s possession must either be destroyed or returned to the OPC Foundation. By using the OPC Materials, User (including any employees and agents of User) agrees to be bound by the terms of this Agreement.

#### LICENSE GRANT:

Subject to the terms and conditions of this Agreement, the OPC Foundation hereby grants to User a non-exclusive, royalty-free, limited license to use, copy, display and distribute the OPC Materials in order to make, use, sell or otherwise distribute any products and/or product literature that are compliant with the standards included in the OPC Materials.

All copies of the OPC Materials made and/or distributed by User must include all copyright and other proprietary rights notices include on or in the copy of such materials provided to User by the OPC Foundation.

The OPC Foundation shall retain all right, title and interest (including, without limitation, the copyrights) in the OPC Materials, subject to the limited license granted to User under this Agreement.

#### WARRANTY AND LIABILITY DISCLAIMERS:

User acknowledges that the OPC Foundation has provided the OPC Materials for informational purposes only in order to help User understand Microsoft’s OLE/COM technology. THE OPC MATERIALS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF PERFORMANCE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. USER BEARS ALL RISK RELATING TO QUALITY, DESIGN, USE AND PERFORMANCE OF THE OPC MATERIALS. The OPC Foundation and its members do not warrant that the OPC Materials, their design or their use will meet User’s requirements, operate without interruption or be error free.

IN NO EVENT SHALL THE OPC FOUNDATION, ITS MEMBERS, OR ANY THIRD PARTY BE LIABLE FOR ANY COSTS, EXPENSES, LOSSES, DAMAGES (INCLUDING, BUT NOT LIMITED TO, DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL, SPECIAL OR PUNITIVE DAMAGES) OR INJURIES INCURRED BY USER OR ANY THIRD PARTY AS A RESULT OF THIS AGREEMENT OR ANY USE OF THE OPC MATERIALS.

**GENERAL PROVISIONS:**

This Agreement and User's license to the OPC Materials shall be terminated (a) by User ceasing all use of the OPC Materials, (b) by User obtaining a superseding version of the OPC Materials, or (c) by the OPC Foundation, at its option, if User commits a material breach hereof. Upon any termination of this Agreement, User shall immediately cease all use of the OPC Materials, destroy all copies thereof then in its possession and take such other actions as the OPC Foundation may reasonably request to ensure that no copies of the OPC Materials licensed under this Agreement remain in its possession.

User shall not export or re-export the OPC Materials or any product produced directly by the use thereof to any person or destination that is not authorized to receive them under the export control laws and regulations of the United States.

The Software and Documentation are provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, the OPC Materials.

## 1 Scope

This document specifies mapping between the security model described in [UA Part 2], the abstract service definitions described [UA Part 4] and the data structures defined in [UA Part 5] and the physical network protocols that can be used to implement the UA specification.

## 2 Reference Documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 2] OPC UA Specification: Part 2 – Security Model

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles

<http://www.opcfoundation.org/UA/Part7/>

[XML Schema Part 1] XML Schema Part 1: Structures

<http://www.w3.org/TR/xmlschema-1/>

[XML Schema Part 2] XML Schema Part 2: Datatypes

<http://www.w3.org/TR/xmlschema-2/>

[SOAP Part 1] SOAP Version 1.2 Part 1: Messaging Framework

<http://www.w3.org/TR/soap12-part1/>

[SOAP Part 2] SOAP Version 1.2 Part 2: Adjuncts

<http://www.w3.org/TR/soap12-part2/>

[XML Encryption] XML Encryption Syntax and Processing

<http://www.w3.org/TR/xmlenc-core/>

[XML Signature] XML-Signature Syntax and Processing

<http://www.w3.org/TR/xmldsig-core/>

[WS Security] SOAP Message Security 1.1

<http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>

[X509 Security Token Profile] X509 Security Token Profile

<http://www.oasis-open.org/committees/download.php/16785/wss-v1.1-spec-os-x509TokenProfile.pdf>

[WS Addressing] Web Services Addressing (WS-Addressing)

<http://www.w3.org/Submission/ws-addressing/>

[WS Trust] Web Services Trust Language (WS-Trust)

<http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf>

[WS Secure Conversation] Web Services Secure Conversation Language (WS-SecureConversation)

<http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf>

[SSL/TLS] RFC 2246: The TLS Protocol Version 1.0

<http://www.ietf.org/rfc/rfc2246.txt>

[X509] X.509 Public Key Certificate Infrastructure

<http://www.itu.int/rec/T-REC-X.509-200003-I/e>

[WS-I Basic Profile 1.1] WS-I Basic Profile Version 1.1

<http://www.ws-i.org/Profiles/BasicProfile-1.1.html>

[WS-I Basic Security Profile 1.1] WS-I Basic Security Profile Version 1.1

<http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html>

[HTTP] RFC 2616: Hypertext Transfer Protocol - HTTP/1.1

<http://www.ietf.org/rfc/rfc2616.txt>

[HTTPS] RFC 2818: HTTP Over TLS

<http://www.ietf.org/rfc/rfc2818.txt>

[Base64] RFC 3548: The Base16, Base32, and Base64 Data Encodings

<http://www.ietf.org/rfc/rfc3548.txt>

[X690] ITU-T X.690: Basic (BER), Canonical (CER) and Distinguished (DER) Encoding Rules

<http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>

[IEEE-754] Standard for Binary Floating-Point Arithmetic

<http://grouper.ieee.org/groups/754/>

[PKCS #12] PKCS #12: Personal Information Exchange Syntax Standard

<http://www.rsasecurity.com/rsalabs/node.asp?id=2138>

[HMAC] HMAC: Keyed-Hashing for Message Authentication

<http://www.ietf.org/rfc/rfc2104.txt>

[PKCS #1] PKCS #1: RSA Cryptography Specifications Version 2.0

<http://www.ietf.org/rfc/rfc2437.txt>

[FIPS 180-2] Secure Hash Standard (SHA)

<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>

[FIPS 197] Advanced Encryption Standard (AES)

<http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

[WS Inspection] Web Services Inspection Language (WS-Inspection) 1.0

<http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-inspection.asp>

### 3 Terms, definitions, and conventions

#### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Attribute
- 3) Certificate
- 4) Message
- 5) Node
- 6) Profile

### 3.2 OPC UA Part 2 terms

The following terms defined in [UA Part 2] apply.

- 1) OPC UA application
- 2) Authentication
- 3) Integrity
- 4) Authorization
- 5) X.509 Certificate
- 6) SoftwareCertificate
- 7) SessionKeySet
- 8) SecurityToken
- 9) SecureChannel
- 10) PublicKey
- 11) PrivateKey
- 12) Nonce
- 13) Application Certificate
- 14) Software Certificate

### 3.3 OPC UA Mappings terms

#### 3.3.1 Communication Stack

A *Communication Stack* is a collection of software libraries that implement one or more communication protocols. *Communication Stacks* usually have multiple layers where application specific higher level protocols build on more generic lower level protocols.

#### 3.3.2 Mapping

A *Mapping* is way to implement UA using a specific *Communication Stack*. UA currently defines two mappingxs: the XML Web Services mapping and the UA Native mapping.

Mapping	Encoding	Transport Protocol
XML Web Services	UA Binary or XML	SOAP/HTTP
UA Native	UA Binary	SOAP/HTTP or UA TCP

#### 3.3.3 Encoding

An *Encoding* is a format that can be used to serialize the UA messages and data structures. UA currently defines two encodingxs: XML and UA Binary.

### 3.3.4 Transport Protocol

A *Transport Protocol* is a way to exchange serialized UA messages between UA applications.

### 3.3.5 Programmers' Interface

A *Programmers' Interface* is a software interface that presents an abstract view of a *Communication Stack* and allows application developers to build applications without knowing the implementation details for the stack.

### 3.3.6 Certificate Store

A *Certificate Store* is a database that can securely store X509 Certificates, *Certificate Revocation Lists* and *Certificate Trust Lists*.

### 3.3.7 Certificate Trust List

A *Certificate Trust List* is a list of X509 certificates that are trusted by a UA application.

### 3.3.8 Certificate Revocation List

A *Certificate Revocation List* is a list of X509 certificates that have been revoked by a *Certification Authority*.

### 3.3.9 Certification Authority

A *Certification Authority* is an entity responsible for issuing X509 certificates.

## 3.4 Abbreviations and symbols

API	Application Programming Interface
UA	Unified Architecture
WS-*	The XML Web Services Specifications.
BP	WS-I Basic Profile Version
BSP	WS-I Basic Security Profile
XML	Extensible Markup Language
TCP	Transmission Control Protocol
SOAP	Simple Object Access Protocol
HTTP	Hypertext Transfer Protocol
SHA1	Secure Hash Algorithm
SSL	Secure Sockets Layer
TLS	Transport Layer Security
WSS	WS Security
WS-SC	WS Secure Conversation
RST	Request Security Token
RSTR	Request Security Token Response
CRL	Certificate Revocation List
CTL	Certificate Trust List
CA	Certificate Authority

## 4 Overview

### 4.1 The Mapping Process

The other parts of the UA specification are written to be independent of the technology used for implementation. This approach means UA is a flexible specification that will continue to be applicable as technology evolves. On the other hand, this approach means that it is not possible to build a UA application with the information contained in Parts 1 through 5 because important implementation details have been left out. This document addresses this problem by specifying exactly how the UA specification should be mapped to different implementation technologies.

This document defines two mappings at this time: XML Web Services and UA Native. The XML Web Services mapping specifies how UA applications may be implemented using SOAP and the various WS-\* specifications. The UA Native mapping specifies how a UA application may be implemented using a simple binary network protocol such as TCP/IP.

Each of profiles defined in [UA Part 7] specifies one or more mappings which must be implemented by UA applications that support the profile.

## 4.2 Communication Stacks

Implementations of distributed applications like UA are built with software libraries that handle the transfer of data across the network using different network protocols. All of these libraries provide an application programmers' interface (API) that hides the details of the network protocol from the application programmer. In many cases, network protocols are layered on top of other network protocols in what is usually described as a communication stack.

UA makes a distinction between features that are implemented by the application and those which are implemented by the communication stack. Features that are implemented in the communication stack are features that have a common implementation for all UA applications. In many cases, application developers will not have to implement the features provided by the communication stack because standard 'off the shelf' implementations already exist for the development environment they plan to use.

Each UA mapping is based on a standard communication stack that provides a basic level of functionality. In some cases, additional UA specific capabilities must be added to the standard stack. This document does not specify how to implement these standard stacks but simply indicates what features are required and references the specifications that define the key components of these stacks.

## 4.3 Message Encoding

Messages are the basic unit of communication between UA applications. To send a message a UA application must serialize it as a sequence of bytes or characters so it can be sent across a network. The format of the serialized message is called an encoding.

The message encoding and decoding is done by the communication stack. Each mapping will specify the message encoding(s) that may be used as part of the mapping.

This document defines two encodings at this time: UA Binary and XML. The UA Binary encoding is a format that efficiently represents UA messages as binary data. The XML encoding is an XML representation of the same UA message.

Each of the profiles defined in [UA Part 7] specifies an encoding which must be used by UA applications that support the profile.

## 4.4 Formal Interface Definition

Each mapping must also have a formal definition that describes the format of each UA message exchanged between applications. For some mappings, the formal interface definition must also describe how the messages are exchanged.

This document describes process used to create the formal definitions from the abstract specification for each mapping. In addition, this document includes the complete interface definition for each mapping in an appendix.

## 4.5 Certificate Management

UA makes extensive use of digital certificates to provide security. Implementing UA requires a mechanism to create, distribute and control access to these certificates from within the UA applications. All UA mappings use the X509 format to create and distribute digital certificates.

This document describes how these certificates should be installed and the security issues that the application developer must consider.

## 4.6 Server Discovery

Before a UA client can connect to a UA server it must know the network address for the server. However, configuring each client with the address of each server in advance creates many administration problems. For that reason, UA requires a mechanism to publish UA services that are available on the network.

This document describes how different discovery services can be adapted for use with UA.

# 5 Mappings

## 5.1 General

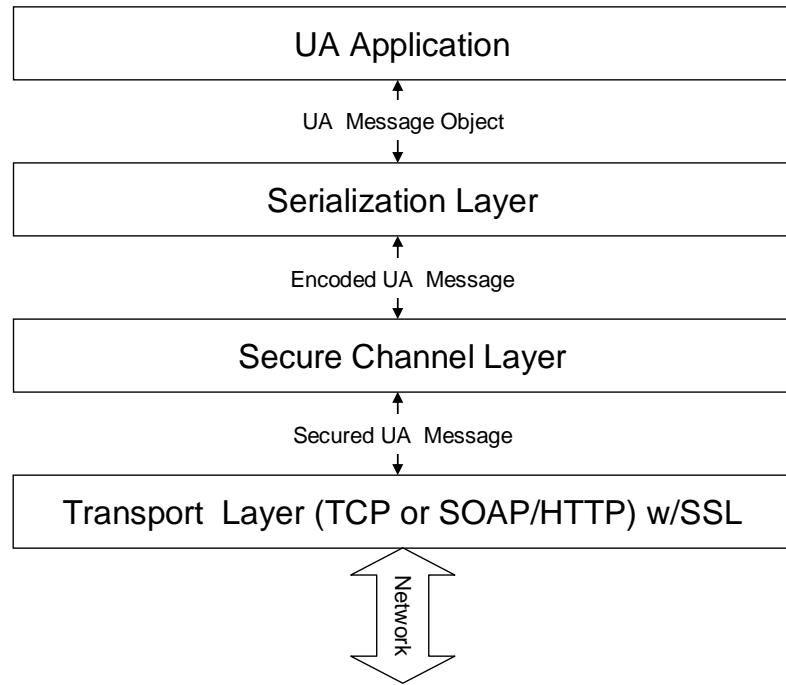
This clause defines the OPC UA mappings. Clause 6 defines the message encodings that are used by the mappings. Clause 7 defines the formal interface contracts for each mapping.

## 5.2 UA Native Mapping

### 5.2.1 General

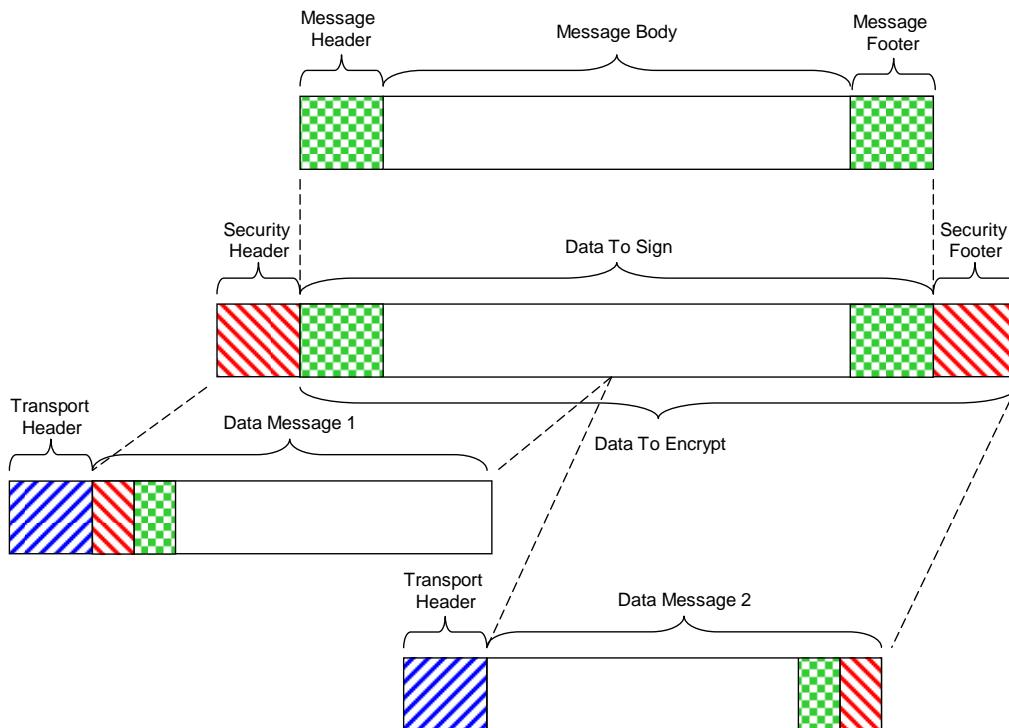
The UA native mapping defines a communication stack that is built on top of TCP/IP or similar low level network protocol. This stack consists of three key components: a transport layer that manages a communication session over the underlying protocol, a secure channel layer that ensure privacy and integrity of messages by implementing the UA secure channel services and a serialization layer which encodes/decodes messages using the UA binary encoding.

An overview of the stack for the UA native mapping is shown in Figure 1.



**Figure 1 – The UA Native Mapping Stack Overview**

Figure 1 illustrates how UA messages are processed as they pass through the different layers. The processing done by each layer is described in the sections follow.



**Figure 2 – UA Native Mapping Message Processing**

### 5.2.2 Serialization Layer

The serialization layer uses the UA binary encoding (see Clause 6.2) to serialize/deserialize UA message structures that store the service parameters. The UA message structures themselves depend on the programming environment and are not defined as part of the mapping. For that reason, the serialization layer will have different implementations for different programming environments.

Each service message defined in [UA Part 4] has either a *RequestHeader* or a *ResponseHeader*. The UA native mapping extracts the *SecurityHeader* parameter from these headers and implements it in the secure channel layer (see Clause 5.2.3).

**Table 1 – UA Native Mapping Message Header**

Name	Type	Description
ServiceId	NodeId	The data type id for the message. This is a <i>NodeId</i> encoded with a two or four-byte encoding (see Clause 6.2.2.10).
MessageLength	Int32	The length of the message body in bytes. The length does not include the length of the <i>MessageFooter</i> . If the length is not known then this value is -1.

The serialization layer also adds a *MessageFooter* that contains the location of all *ExtensionObjects* in the message which were serialized without a length prefix. The structure of the *MessageFooter* is described in Table 2.

**Table 2 – UA Native Mapping Message Footer**

Name	Type	Description
ExtensionObjectPositions	ExtensionObjectPosition[]	A list of <i>ExtensionObjects</i> encoded in the body of the message without a length prefix. This list must be ordered by <i>StartPosition</i> .
StartPosition	UInt32	The number of bytes from the beginning of the message body to the first byte of the <i>ExtensionObject</i> body.
EndPosition	UInt32	The number of bytes from the beginning of the message body to the first byte after the <i>ExtensionObject</i> body. The difference between the StartPosition and EndPosition is the length of the <i>ExtensionObject</i> body.
MessageLength	UInt32	The length of the message body. This is also position of the start of the <i>MessageFooter</i> .

The decoder must use the *MessageFooter* when it encounters an *ExtensionObject* which does not have either a data type id that it recognizes or a length prefix. When this happens, the decoder must find the *MessageLength* in the header or jump to the end of the stream and read the *MessageLength* from the footer. The *MessageLength* tells the decoder where the list of *ExtensionObjectPositions* starts. The decoder then looks for a entry with a *StartPosition* that matches the position where the unknown *ExtensionObject* was encountered.

Decoders that cannot jump to the end of the message stream must return an error.

All messages always have a *MessageFooter* that is at least 8 bytes long (4 bytes to indicate an empty list and 4 bytes for the footer start position).

The *MessageHeader* and *MessageFooter* are encoded using the UA binary encoding.

### 5.2.3 Secure Channel Layer

#### 5.2.3.1 General

The UA secure channel layer manages a virtual connection between two applications that ensures that all messages exchanged via the channel are secured (i.e. signed and/or encrypted). The secure channel connection is always tied to a single transport layer session.

The secure channel layer prepends a *SecurityHeader* to the encoded message that tells the receiver what security algorithms have been applied to the message. The components of the *SecurityHeader* are described in Table 3.

**Table 3 – Security Header Components**

Name	Type	Description
SigningToken	ByteString	The token used to sign the message.
SignatureMethod	String	The algorithm used to sign the message. Not specified if the message is not signed.
EncryptingToken	ByteString	The token used to encrypt the message.
EncryptionMethod	ByteString	The algorithm used to encrypt the message. Not specified if the message is not encrypted.

The *SecurityHeader* is encoded using the UA binary encoding. How each field is used is described in the clauses below

The secure channel layer must also append a *SecurityFooter* to the encoded message that provides additional information which the receiver needs to process and validate the message. The components of the *SecurityFooter* are shown in Table 4.

**Table 4 – Security Footer Components**

Name	Type	Description
Signature	ByteString	The signature calculated for the message. The number of bytes depends on the signature algorithm.
Padding	ByteString	Empty bytes required make the message size (including the <i>SecurityFooter</i> ) a multiple of the encryption algorithm block size.
Message Size	UInt32	The size of the unencrypted message not including the <i>SecurityHeader</i> or <i>SecurityFooter</i> .

All encryption algorithms operate on a fixed block size which means it is usually necessary to add padding to the end of a message to fill up an encryption block. The *Padding* in the *SecurityFooter* exists for this purpose. The length of the padding can be calculated with the following formula:

```
PaddingSize = (sizeof(Message)+sizeof(SecurityFooter with no Padding))
             mod EncryptionBlockSize
```

Note that the *SecurityFooter* is encoded with the UA Binary Encoding so each of the ByteString values includes a 4 byte integer that indicates how many bytes are in the array. Therefore the size of *SecurityFooter* without any padding is the length of the signature + 3 \* sizeof(Int32).

Table 5 specifies the identifiers used in the *SecurityHeader* for common security algorithms. [UA Part 7] defines mandatory and optional algorithms for each profile. [UA Part 7] may also define constants for algorithms not listed here.

**Table 5 – Security Algorithms**

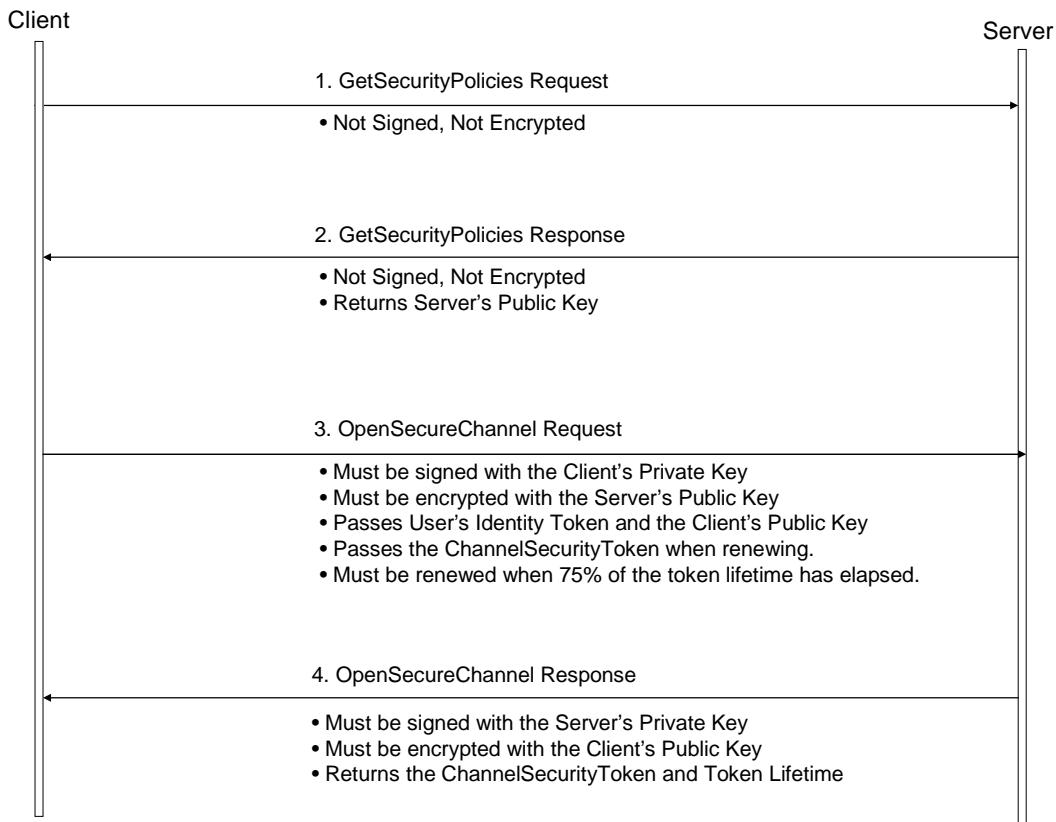
Algorithm Id	Use	Description
HMAC-SHA1	Signature	Use the [HMAC] algorithm with the SHA1 hash algorithm from [FIPS 180-2]
HMAC-SHA256	Signature	Use the [HMAC] algorithm with the SHA256 hash algorithm from [FIPS 180-2]
RSA-SHA1	Signature	Use the RSA PKCS #1 v1.5 digital signature from [PKCS #1] with a SHA1 message digest.
AES128-CBC	Encryption	Use the AES128 algorithm from [FIPS 197] in Cipher Block Chaining mode.
AES196-CBC	Encryption	Use the AES196 algorithm from [FIPS 197] in Cipher Block Chaining mode.
AES256-CBC	Encryption	Use the AES1256 algorithm from [FIPS 197] in Cipher Block Chaining mode.
RSA-PKCS1	Encryption	Use the RSA PKCS #1 v1.5 encryption from [PKCS #1]
RSA-OAEP	Encryption	Use RSA encryption with OAEP padding from [PKCS #1]

### 5.2.3.2 Establishing a Channel

A secure channel connection is created by sending a *OpenSecureChannel* request to the server as soon as the transport layer connection is available. In some cases, the client will have to send a *GetSecurityPolicies* request to the server before it can create the secure channel because it needs to discover the server's application instance certificate and supported security policies.

Once the secure connection has been established, the secure channel layer on the client is responsible for periodically renewing the session keys by sending a *OpenSecureChannel* request.

The sequence used establish and renew a secure channel is shown in Figure 3.



**Figure 3 – Establishing a Secure Channel**

Once the secure channel is established the messages are signed and encrypted with keys derived from the nonces exchanged in the OpenSecureChannel call. The algorithm for deriving keys is based on the P\_SHA1 algorithm which defined in the [SSL/TLS] specification. The function is defined as follows:

```

P_SHA1(secret, seed) = HMAC_SHA1(secret, A(1) + seed) +
                      HMAC_SHA1(secret, A(2) + seed) +
                      HMAC_SHA1(secret, A(3) + seed) + ...
  
```

Where A(i) is defined as:

```

A(0) = seed
A(i) = HMAC_SHA1(secret, A(i-1))
+ indicates that the results are appended to previous results.
  
```

The output of the P\_SHA1 function is an infinite series of bytes. For convenience, the following function is used to select a specific set of bytes from the series:

```

Byte[] P_SHA1(Byte[] secret, Byte[] seed, Int32 length, Int32 offset)
  
```

Where *length* is the number of bytes to select and *offset* is the position from the beginning of the series.

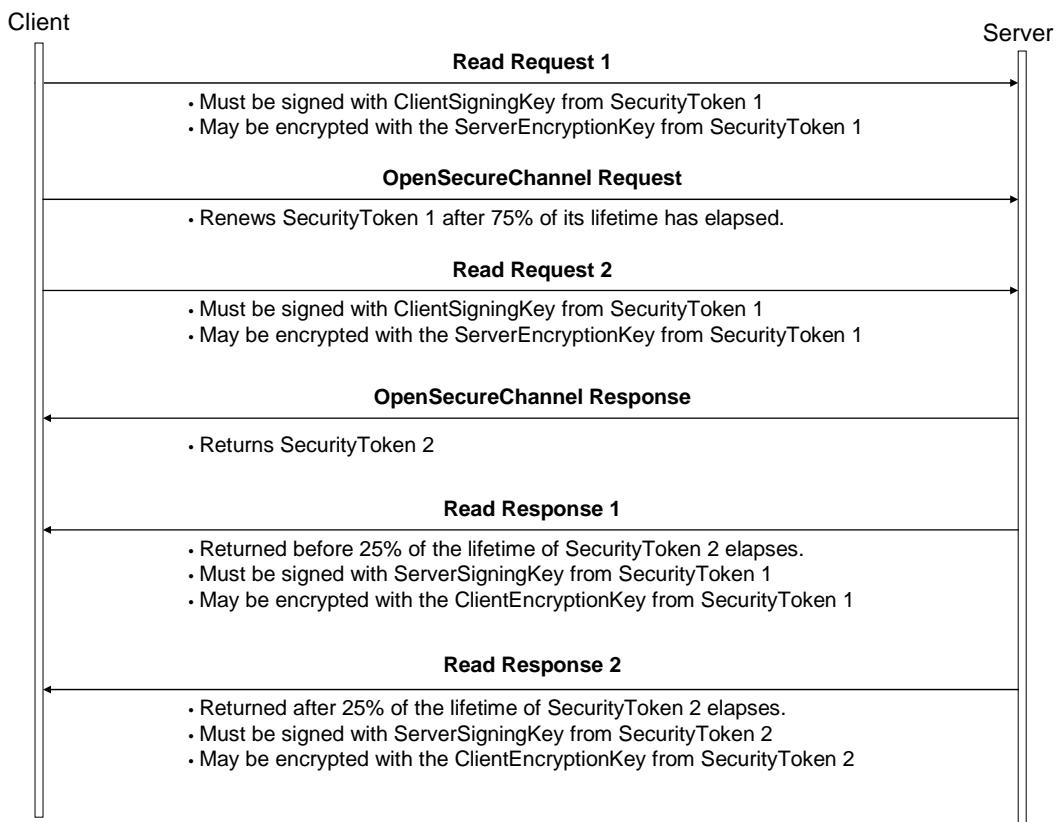
The keys used to secure messages are derived with the following formulaxs:

```

SharedSecret          = P_SHA1(ClientNonce, ServerNonce, KeySize, 0)
ClientSigningKey     = P_SHA1(SharedSecret, ClientNonce, KeySize, 0)
ClientEncryptionKey  = P_SHA1(SharedSecret, ClientNonce, KeySize, KeySize)
ClientInitializationVector = P_SHA1(SharedSecret, ClientNonce, KeySize, 2*KeySize)
ServerSigningKey     = P_SHA1(SharedSecret, ServerNonce, KeySize, 0)
ServerEncryptionKey  = P_SHA1(SharedSecret, ServerNonce, KeySize, KeySize)
ServerInitializationVector = P_SHA1(SharedSecret, ClientNonce, KeySize, 2*KeySize)

```

Figure 4 illustrates how the various keys are derived and used when securing messages exchanged between UA applications.



**Figure 4 – Renewing Security Tokens**

### 5.2.3.3 Applying Signatures

Most messages must be signed by the secure channel layer. How the signature is generated depends on the service being called and the security policies set by the server.

If the secure channel layer is sending *GetSecurityPolicies* request or response then the message is not signed.

If the secure channel layer is sending *OpenSecureChannel* request or response then the message must always be signed with the private key from application instance certificate for the sender. In this case, the *SigningToken* is the DER encoded form of the X509 certificate used to create the signature. The DER encoding is defined in [X690] specification.

The sender must choose a *SignatureMethod* that is appropriate for the X509 certificate being used. The server returns the *SignatureMethods* that it will accept as part of the *BootstrapPolicies* in the *GetSecurityPolicies* response. The server may choose to accept *SignatureMethods* which it did not

explicitly claim to support, however, clients must be prepared for an error if they do not use one of the methods specified in the server's *BootstrapPolicies*.

If the secure channel layer is sending any other message then there must be an active secure channel (i.e. *OpenSecureChannel* has been called at least once and *CloseSecureChannel* has not been called). The *SignatureMethod* is the same as specified in the *SecurityPolicy* selected in the *OpenSecureChannel* request. In this case, the *SigningToken* is a *TokenId* for the secure channel encoded as a null terminated UTF-8 string. Clause 5.4.3 of [UA Part 4] explains how to choose the correct *TokenId* for a secure channel.

Once the secure channel layer has calculated the signature it adds the signature to the *SecurityFooter* along with the size of the unencrypted message excluding the *SecurityHeader* and *SecurityFooter*.

#### 5.2.3.4 Verifying Signatures

When a secure channel layer receives and decrypts a message it must read the *SecurityHeader* and then jump to the end of the message read the unencrypted message size. The receiver can now use the information in the *SecurityHeader* to recalculate the signature and compare it to the one provided in the *SecurityFooter*.

The receiver of a message must generate an error if the signature is not valid. If the message is signed with a security token for a secure channel then the receiver must reject the message if uses an *SignatureMethod* that does not match the algorithm specified in the *SecurityPolicy* that was selected when the *OpenSecureChannel* service was called.

#### 5.2.3.5 Encryption

Most messages could be encrypted by the secure channel layer. The algorithm used to encrypt the messages depends on the message type and the active *SecurityPolicy*. The *SecurityHeader* is not encrypted, however the *SecurityFooter* is encrypted.

If the secure channel layer is sending a *GetSecurityPolicies* request or response then the message should not be encrypted.

If the secure channel layer is sending a *OpenSecureChannel* request or response then the message must be encrypted with the application instance certificate for the receiver. In this case, the *EncryptingToken* is the thumbprint of the X509 certificate. The thumbprint of a X509 certificate is the SHA-1 digest of the DER encoded form of the certificate.

The sender must choose a *EncryptionMethod* that is appropriate for the X509 certificate be used. The server returns the *EncryptionMethods* that it will accept as the *BootstrapPolicies* in the *GetSecurityPolicies* response. The server may choose to accept *EncryptionMethods* which it did not explicitly claim to support, however, clients must be prepared for an error if they do not use one of the methods specified in the server's *BootstrapPolicies*.

If the secure channel layer is sending any other message then there must be an active secure channel (i.e. *OpenSecureChannel* has been called at least once and *CloseSecureChannel* has not been called). The *EncryptionMethod* is the same as specified in the *SecurityPolicy* selected in the *OpenSecureChannel* request. In this case, the *EncryptingToken* the *SigningToken* is a *TokenId* for the secure channel encoded as a null terminated UTF-8 string. Clause 5.4.3 of [UA Part 4] explains how to choose the correct *TokenId* for a secure channel.

The signature must always be calculated before any encryption is applied and the signature bytes are encrypted as part of the security footer.

### 5.2.3.6 Decryption

A receiver reads the *SecurityHeader* to determine if encryption was applied to message and what algorithm was used. After the message is decrypted the last 4 bytes of the last block of plaintext contain the true size of the message (See description of *SecurityFooter* above). This size is used extract the *SecurityFooter* and verify signature.

## 5.2.4 Transport Layer

### 5.2.4.1 UA TCP Protocol

#### 5.2.4.1.1 General

The UA TCP transport protocol manages a virtual connection between two applications that is automatically re-established if the underlying TCP socket is interrupted for some reason. In addition, the UA TCP protocol provides a way to break UA messages in smaller chunks and re-assemble them correctly.

The client is the application that establishes the initial connection. The server is the application that listens for new connections. The client must know in advance how to send a request for a new connection to the server. This is typically specified in the URL for a server endpoint that supports the UA TCP protocol.

For this reason, UA defines the URL scheme 'opc.tcp' which can be used to identify endpoints that support UA over TCP/IP. This scheme has a syntax which is identical to an http URL. The default port is 21897; however, this can be overridden in the URL itself. Examples of these URLs are:

```
opc.tcp://mycompany.com:1245/MyServer
opc.tcp://196.168.1.200/ServerX
```

The path portion of the URL is only for visual identification since only one application can use a single port. Future versions of this specification will define a mechanism to allow multiple applications to share a single port. In this case, the path will be used to direct results to the correct server application.

The UA TCP protocol defines several messages that may be exchanged between the client and server. These messages are summarized in Table 6 and is described in detail in the clauses that follow.

**Table 6 – UA TCP Message Types**

Message	Signature	Description
Hello	UATH	Establishes a new virtual connection with the server. This message is only sent by the client.
Acknowledge	UATA	Acknowledges a new virtual connection. This message is only sent by the server.
Disconnect	UATD	Gracefully closes a virtual connection.
Data	UAMG or UAMC	Transfers part of a UA service message. The signature is 'UAMG' for the last piece of any UA message.
Abort	UAMA	Discards a partially transmitted UA message.
Error	UAME	Returns a error for a request

Platforms with limited memory may not be able to store an entire UA message in memory before sending it, as a result, they need to break the message into several chunks as send each chunk before continuing with message encoding. Receivers must be prepared to reassemble these chunks before processing the message.

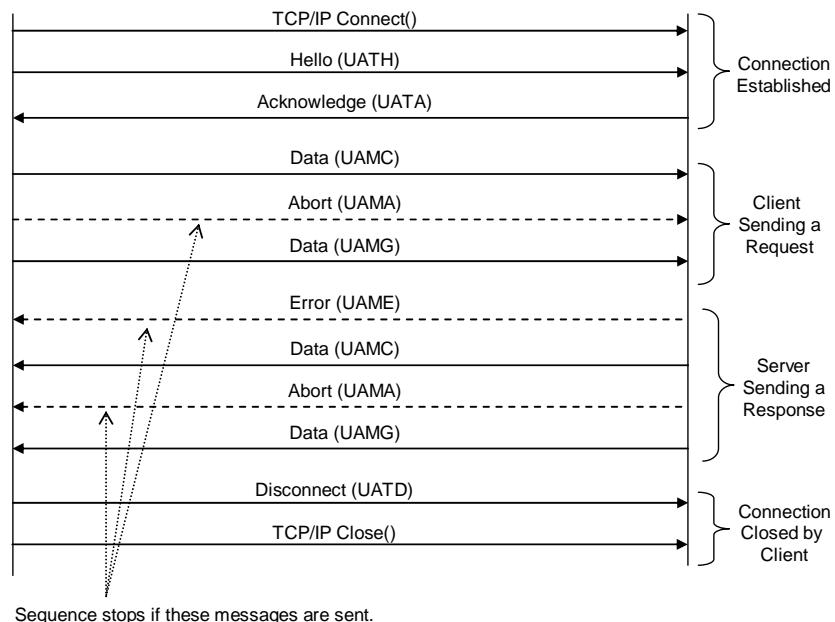
When a sender breaks a UA message into multiple pieces it must send the entire UA message (or send an abort message) before starting the transmission of a new UA message. The receiver must

abort the message immediately If the TCP/IP connect breaks. The sender must return a communication failure error for that particular message even If the connection is later re-established.

All messages headers are encoded using the UA Binary encoding.

The UA TCP transport may support communication over SSL/TLS instead of normal sockets.

Figure 5 illustrates the message sequence flow between a client and server during normal operation.



**Figure 5 – UA TCP Normal Message Sequences**

### 5.2.4.1.2 Hello

Once a TCP/IP connection has been established the client must send a hello message that includes the URL of the endpoint that it wishes to connect to plus a TCP/IP connection identifier (if re-establishing an interrupted connection). The hello message is described in Table 7.

**Table 7 – UA TCP Hello Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the hello message. Must always be the ASCII string “UATH”
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and length).
Version	UInt32	The version number of the used TCP message protocol. The server may reject the connection, if it does not support the requested protocol. If the server does not support the requested version then the server responds with an Error message.
ConnectionId	Guid	A unique identifier assigned by the server to a previous TCP/IP connection. This allows clients to re-establish connections that were interrupted for some reason. Clients should send a Null Guid if when establishing a connection for the first time.
Lifetime	UInt32	The amount of time, in seconds, that the server should keep resources allocated for the connection alive for if the client unexpectedly disconnects. If the client specifies zero then the server should choose a suitable default value.
SendBufferLength	UInt32	The size of the buffer, in bytes, that the client uses to send requests. The client will not send message fragments larger than this size. If the client specifies zero then the server should choose a suitable default value.
ReceiveBufferLength	UInt32	The size of the buffer, in bytes, that the client uses to receive responses. The server should not send message fragments larger than this size. If the client specifies zero then the server should choose a suitable default value.
Endpoint	String	The URL of the endpoint that the client wishes to connect to.

### 5.2.4.1.3 Acknowledge

After sending the hello message the client must wait for an acknowledgement from the server before sending any more messages along the connection. The acknowledgement message is described in Table 8.

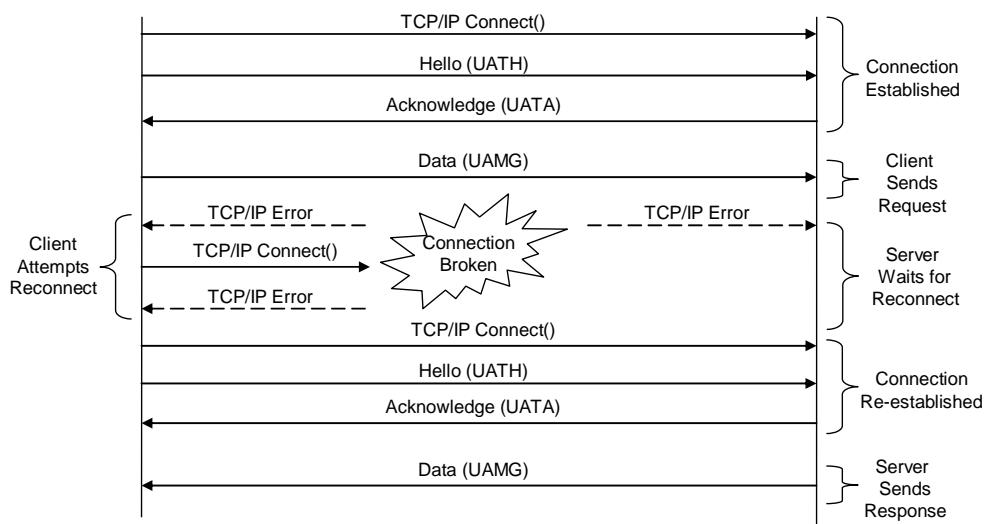
**Table 8 – UA TCP Acknowledge Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the acknowledgement message. Must always be the ASCII string “UATA”
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and length).
ConnectionId	Guid	A unique identifier assigned by the server for the connection. If this value is a Null Guid then the server has rejected the connection and closed the socket. If the client specified a connection id in the hello message then the server must return the same id if it was possible to re-establish the same connection. If the connection could not be re-established the server should return a new connection identifier.
RevisedLifetime	UInt32	The amount of time, in seconds, that the server will keep resources allocated for the connection alive for if the client unexpectedly disconnects. This value may be different from the time requested by the client.
RevisedSendBufferLength	UInt32	The size of the buffer, in bytes, that the server uses to receive requests. If the requested size is non-zero, the value should be less or equal. If the client specified zero, the server should choose a suitable default value .
RevisedReceiveBufferLength	UInt32	The maximum size of a message fragment, that will be sent by the server. This value should be equal or smaller than the requested value. If the client specified zero, the server should choose a suitable default value. Returning 0 for means messages will not be fragmented by the server.
AlternateEndpoint	String	If the server rejects a connection it can specify an alternate URL that the client may use to re-attempt a connection. This mechanism allows multiple applications to share a single port.

Once the connection has been established both the client and the server may send a UA service message at any time. In addition, the client may send multiple requests without waiting for a response and the server may respond to these requests in any order. However, once a client or server starts to send pieces of a message they must send all pieces of that message before sending any other message.

If the client closes the connection unexpectedly then the server must keep the any resources allocated for the connection active for the lifetime of the connection. Once the lifetime expires the server must assume the client has permanently gone and send an event to the higher layers.

If the server closes the connection unexpectedly then the client may periodically attempt to reconnect. If the client cannot reconnect within the connection lifetime then the client must assume the server is permanently gone and release all resources. Figure 6 illustrates the message sequence flow between a client and server when recovering from a network error. It is up to the client to decide how frequently it attempts to reconnect.



**Figure 6 – UA TCP Error Recovery Sequences**

#### **5.2.4.1.4 Disconnect**

If either the client or server wish to disconnect gracefully they must send a disconnect message before closing the socket. The disconnect message is described in Table 9.

**Table 9 – UA TCP Disconnect Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the disconnect message. Must always be the ASCII string "UATD"
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and length).
ConnectionId	Guid	A unique identifier assigned by the server for the connection.

### 5.2.4.1.5 Data

If either the client or server wish to send all or part of a UA service message the must send a data message. The data message is described in Table 10.

**Table 10 – UA TCP Data Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the data message. Must be the ASCII string “UAMG” if the message is complete. Must be the ASCII string “UAMC” if there subsequent message.
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and length).
ConnectionId	Guid	A unique identifier assigned by the server for the connection.
RequestId	UInt32	A unique identifier assigned by the client for the request that allows the client to match a response with a request. The server must return this value with the response.  A value of 0 is an invalid value and must not be used by the client.
MessageData	ByteString	The data in the message.

### 5.2.4.1.6 Abort

If the client or server wishes to discard a UA service message that has already been partially transmitted it must send an abort message. The abort message is described in Table 11.

**Table 11 – UA Abort Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the abort message. Must always be the ASCII string “UAMA”
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and size).
ConnectionId	Guid	A unique identifier assigned by the server for the connection.
Request Id	UInt32	A unique identifier assigned by the client for the request that allows the client to match a response with a request. The server must return this value with the response.

### 5.2.4.1.7 Error

If the server needs to return an error for a specific request then it must send an error message. The error message is described in Table 12.

**Table 12 – UA Error Message**

Field	Encoding	Comments
MessageType	Byte[4]	The signature of the error message. Must always be the ASCII string “UAME”
Length	UInt32	The size of the message in bytes (including the <i>MessageType</i> and size).
ConnectionId	Guid	A unique identifier assigned by the server for the connection.
Request Id	UInt32	A unique identifier assigned by the client for the request that allows the client to match a response with a request. The server must return this value with the response.  This value is 0 if the error is not related to a specific request.
StatusCode	StatusCodes	The numeric code for the error that occurred.

### 5.2.4.2 SOAP/HTTP

The UA Binary mapping also supports using SOAP/HTTP as a transport layer. In this scenario, the secured message is added to the SOAP body using the *InvokeService* XML syntax described in Clause 5.3.2.3.

In this scenario, none of other WS-\* specifications are used. SOAP is simply a mechanism to move the bytes of the messages across a network. This approach may be used by applications that need to communicate through firewalls that only allow HTTP traffic.

### 5.3 XML Web Services Mapping

#### 5.3.1 General

The XML web services mapping requires a communication stack that supports several web services specifications. The components required by UA are shown in Figure 7.

WS Secure Conversation 1.0		
WS Security 1.1		WS Trust 1.0
XML Signature 1.0	XML Encryption 1.0	WS Addressing 1.0
SOAP 1.2		
HTTP or HTTPS (SSL/TLS)		

**Figure 7 – The XML Web Services Stack**

[HTTP] is the network communication protocol used to exchange SOAP messages. A UA service request message is always sent by the client in the body of an HTTP POST request. The server returns UA response message in the body of the HTTP response.

[HTTPS] is a variant of HTTP that encrypts and/or signs HTTP messages using the [SSL/TLS] protocol. HTTPS provides a efficient way to encrypt data sent across the network when two applications can communicate directly without intermediaries. However, HTTPS cannot meet the UA security requirements on its own: [WS Security] and [XML Signature] must always be used to sign the SOAP message bodies.

[WS Security] and [XML Encryption] may be used to encrypt SOAP messages instead of HTTPS.

[WS Addressing] specifies several SOAP standard headers which are used by SOAP intermediaries and stacks to route SOAP messages.

[WS Trust] and [WS Secure Conversation] provide a mechanism to establish secure channels as described in Clause 5.4 of [UA Part 4]. The exact relationship between these specifications and the UA secure channel services is discussed in Clause 5.3.5.

In addition to the specifications shown above, web service stacks must conform to [WS-I Basic Profile 1.1] and [WS-I Basic Security Profile 1.1]. These specifications ensure better interoperability between XML web service applications by clarifying or restricting the use of the various web service specificaions.

Many XML web service stacks support web services specifications that are not listed above (e.g. MTOM, WS-Reliable Messaging or WS-Profile). These specifications are not used by UA at this time, however, they may be incorporated in future versions of this document.

### 5.3.2 SOAP

#### 5.3.2.1 General

SOAP describes an XML based syntax for exchanging messages between applications. UA messages are exchanged using SOAP by serializing the UA messages using one of the supported encodings described in Clause 6 and inserting that encoded message into the body of the SOAP message.

All UA messages are exchanged using the request-response message exchange pattern even if the UA service does not specify any output parameters. The server must return an empty response message that tells the client that no errors occurred.

UA does not define any SOAP headers, however, SOAP messages containing UA messages will include headers used by the other WS specifications in the stack.

All UA applications that support the XML web services mapping must support SOAP 1.2 as described in [SOAP Part 1].

SOAP faults are returned only for errors that occurred with in the SOAP stack. Errors that occur within the UA application are returned as UA error response messages as described in Clause 5.3 of [UA Part 4].

UA applications that support the XML Web Services mapping must support at least one of the UA message encoding formats. The mechanisms for inserting UA messages into SOAP messages are described in the clauses below. The SOAP action associated with each SOAP message indicates the encoding used in the message body.

#### 5.3.2.2 XML Encoding

The UA XML Encoding specifies a way to represent a UA message as an XML element. This element is added to the SOAP message as the only child of the SOAP body element.

If an error occurs in the server while parsing the request body, the server may return a SOAP fault or it may return a UA error response.

The SOAP Action associated with an XML encoded request message always has the form:

```
http://opcfoundation.org/UA/<service name>
```

Where <service name> is the name of the UA service being invoked.

The SOAP Action associated with an XML encoded response message always has the form:

```
http://opcfoundation.org/UA/<service name>Response
```

#### 5.3.2.3 UA Binary Encoding

The UA Binary Encoding specifies a way to represent a UA message as a sequence of bytes. These bytes sequences must be encoded in the SOAP body using the [Base64] data format.

The Base64 data format is a UTF-7 representation of binary data that is less efficient than raw binary data, however, many UA applications that exchange messages using SOAP will find that encoding UA messages in UA binary and then encoding the binary in Base64 is more efficient than encoding everything in XML.

The WSDL definition for a UA Binary encoded request message is:

```
<xs:element name="InvokeService" type="xs:base64Binary" nillable="true" />

<wsdl:message name="InvokeServiceMessage">
  <wsdl:part name="InvokeService" element="tns:InvokeService" />
</wsdl:message>
```

The SOAP Action associated with a UA binary encoded request message always has the form:

<http://opcfoundation.org/UA/InvokeService>

The WSDL definition for a UA Binary encoded response message is:

```
<xs:element name="InvokeServiceResponse" type="xs:base64Binary" nillable="true" />

<wsdl:message name="InvokeServiceResponseMessage">
  <wsdl:part name="InvokeServiceResponse" element="tns:InvokeServiceResponse" />
</wsdl:message>
```

The SOAP Action associated with a UA binary encoded response message always has the form:

<http://opcfoundation.org/UA/InvokeServiceResponse>

### 5.3.3 HTTP

HTTP defines a mechanism to exchange SOAP messages across a network. SOAP stacks used to implement the XML web service mapping must support the SOAP HTTP binding defined in [SOAP Part 2].

Some UA profiles defined in [UA Part 7] will require that transport layer security with HTTP over TLS as described in [HTTPS]

### 5.3.4 WS Addressing

[WS Addressing] defines standard headers used to route SOAP messages through intermediaries. SOAP stacks used to implement the XML Web Service mapping must support the WS-Addressing headers listed in Table 13.

**Table 13 – WS-Addressing Headers**

Header	Request	Response
To	Required	Required
From	Required	Required
ReplyTo	Required	Not Used
Action	Required	Required
MessageID	Required	Required
RelatesTo	Not Used	Required

Note that WS-Addressing defines standard URIs to use in the ReplyTo and From headers when a client does not have an externally accessible endpoint. In these cases, the SOAP response is always returned to the client using the same communication channel that sent the request.

UA servers must ignore the FaultTo header if it is specified in a request.

### 5.3.5 WS Security

#### 5.3.5.1 General

[WS Security] defines headers used to describe how the components of SOAP messages are signed and/or encrypted. The mechanisms for actually signing XML elements are described in the [XML Signature] specification. The mechanisms for encrypting XML elements are described in the [XML Encryption] specification.

[UA Part 4] defines a *RequestHeader* or a *ResponseHeader* parameter for each service message. These parameters include a field called the *SecurityHeader* which is not included in message bodies when UA applications use the XML web service mapping. Instead, it is implemented with the SOAP headers defined by the [WS Security] specification.

#### 5.3.5.2 Signing Messages

A UA message may be signed with either a private key associated with a X509 certificate or with the a secret key derived from a [WS Secure Conversation] security context token.

If the message is signed with an X509 certificate then the WS-Security header must contain a X509 security token that conforms to the [X509 Security Token Profile]. The X509 security token should not assume that the receiver has prior knowledge of the certificate so the entire X509v3 certificate should be included in the token data.

If the message is signed with a secret key then the WS Security header must contain a *DerivedKeyToken* as described in the [WS Secure Conversation] specification. The algorithm used to derive the key is specified in each *DerivedKeyToken*, however, applications should use the algorithms defined in Clause 5.2.3.2.

#### 5.3.5.3 Encrypting Messages

A UA message may be encrypted with either a public key associated with a X509 certificate or with a secret key derived from a [WS Secure Conversation] security context token. If a X509 certificate is used then the WS Security headers should include a X509 certificate token that identifies the certificate by either its thumbprint or subject. The entire X509 certificate should not be included in the token.

If the message is encrypted with a secret key then the WS Security header must contain a *DerivedKeyToken* as described in the [WS Secure Conversation] specification. The algorithm used to derive the key is specified in each *DerivedKeyToken*, however, applications should use the algorithms defined in Clause 5.2.3.2.

#### 5.3.5.4 Timestamp

The [WS Security] specification uses the timestamp header to protect against replay attacks. UA applications may use this timestamp header, however, if it is present it should be signed with the same security token that was used to sign the message body.

The timestamp header is not mandatory because UA messages include a timestamp in the *RequestHeader* and *ResponseHeader* parameters.

The timestamp header should be present in the WS-Trust messages used to implement the *OpenSecureChannel* and *CloseSecureChannel* services.

### 5.3.6 WS Trust

#### 5.3.6.1 General

[WS Trust] defines a mechanism to validate credentials and create security tokens that can be used as proof of possession of those credentials. [WS Trust] is used in the XML web services mapping in two ways. First, it provides a mechanism to translate user identity tokens owned by the UA client into something that the UA server will accept. Second, it provides the mechanism to issue and renew the security context tokens required by the [WS Secure Conversation] specification.

#### 5.3.6.2 User Identity Tokens

Each UA server uses the *GetSecurityPolicies* service described in Clause 5.4.2 of [UA Part 4] to publish the types of user identity tokens that it supports. If a UA service accepts user tokens issued by a WS-Trust server then it must return a *UserTokenPolicy* structure that identifies the WS-Trust server and the type of tokens that the WS-Trust server issues. The contents of the *UserTokenPolicy* structure are described in Table 14.

**Table 14 – WS-Trust User Token Policy**

Parameter	Description
TokenType	The WS-Security Token URI
Issuer	The URL for the WS-Trust server.
IssuerType	Must always be "WS-Trust"

The security token returned from the WS-Trust server is an XML element with a syntax defined by the appropriate WS-Security token profile specification. The client must take the XML element and pass it to the *CreateSession* service as an instance of the *WssSecurityToken* type (see Clause 7.7.4 of [UA Part 4]).

#### 5.3.6.3 Secure Channels

Most UA services can only be called after creating a secure communication channel. When UA applications implement the XML web services mapping, they must use [WS Trust] to request a security context token defined by the [WS Secure Conversation] specification to secure the message. The exact use of WS-Trust for this purpose is described in Clause 5.3.7.

### 5.3.7 WS Secure Conversation

[WS Secure Conversation] provides a mechanism to exchange shared secret keys that can be used to efficiently encrypt and sign multiple messages exchanged between web service applications over a period of time.

The tables in this section use a number of prefixes which have meanings defined in Table 15.

**Table 15 – WS-\* Namespace Prefixes**

Prefix	Specification
wsu	WS-Security Utilities
wsse	WS-Security Extensions
wst	WS-Trust
wsc	WS-Secure Conversation
wsa	WS-Addressing
xenc	XML Encryption

[WS Secure Conversation] is used to implement the secure channel service set described in [UA Part 4]. In this mapping, a secure channel is created by requesting a security context token (SCT) from a WS-Trust server (called the SCT issuer) that supports the SCT binding. This SCT maps

directly onto the *ChannelSecurityToken* parameter defined in [UA Part 4]. The exact relationship is described in Table 16.

**Table 16 – ChannelSecurityToken to SCT Mapping**

Parameter	Mapping
ChannelId	This value is the wsc:Identifier element of the SCT.
TokenId	This value is the wsc:Instance element of the SCT.
CreatedAt	This value is the wsu:Created time in the wst:Lifetime element returned by the SCT issuer. If this value is not specified then the client uses the time that it receives the SCT.
RevisedLifetime	This value is calculated by subtracting the wsu:Created time from the wsu:Expires time in the wst:Lifetime element returned by the SCT issuer. If the wst:Lifetime is not specified then this value is 0 (i.e. the SCT never expires).

It is expected that SOAP stack will automatically handle the handshaking required to establish and renew security context token. However, the UA application must have access to the identifier for the SCT associated with each message and this identifier must not change for the duration of the UA application session.

A client creates or renews a SCT by sending a *RequestSecurityToken* (RST) message to the same endpoint that is used for all other UA services. The SOAP action associated with this message must be “<http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT>”.

The RST message has the components described in Table 17

**Table 17 – RequestSecurityToken Message Components**

Name	Description
wst:RequestedSecurityToken	Specifies the new SCT or renewed SCT.
wst:RequestType	Must be “ <a href="http://schemas.xmlsoap.org/ws/2005/02/trust/Issue">http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</a> ” when creating a new SCT. Must be “ <a href="http://schemas.xmlsoap.org/ws/2005/02/trust/Renew">http://schemas.xmlsoap.org/ws/2005/02/trust/Renew</a> ” when renewing a SCT.
wst:Entropy	This contains the nonce specified by the server. The nonce is specified with the wst:BinarySecret element. The xenc:EncryptedData element is not used in UA because the message body must be encrypted with the public key from the server’s certificate.
wst:OnBehalfOf	Specifies the identity of the user associated with the security SCT. This parameter is optional. This value is a <i>SecurityToken</i> that is constructed from the <i>UserIdentityToken</i> . A <i>UserNametIdentityToken</i> is mapped onto Username Security Token A <i>X509IdentityToken</i> is mapped onto a X509 Security Token A <i>WSSIdentityToken</i> is already a WS-Security compatible Security Token.
wst:KeySize	This is the <i>SignatureKeyLength</i> from the <i>SecurityPolicy</i> selected by the client. This also specifies the length of the nonce.
wst:SignatureAlgorithm	This is the URI for <i>SignatureMethod</i> from the <i>SecurityPolicy</i> selected by the client. The mapping between the UA algorithm id and the WS-* URI is shown in Table 18.
wst:EncryptionAlgorithm	This is the URI for <i>EncryptionMethod</i> from the <i>SecurityPolicy</i> selected by the client. The mapping between the UA algorithm id and the WS-* URI is shown in Table 18.
wst:Lifetime	The requested lifetime for the SCT.

The body of the RST message must be signed with private key from the client’s application instance certificate. The body must also be encrypted with the public key from the server’s application instance certificate.

Clients are expected to choose one of the *SecurityPolicies* returned by the server’s *GetSecurityPolicies* service. The server must reject the request if cannot find a *SecurityPolicy* that matches the combination of *SignatureKeyLength*, *SignatureMethod* and *EncryptionMethod*.

Servers that support the XML web services mapping should not support multiple *SecurityPolicies* for the same combination of *SignatureKeyLength*, *SignatureMethod* and *EncryptionMethod*. The mappings between UA algorithm identifiers and the URIs specified in the RST are shown in Table 18.

**Table 18 – Security Algorithm Mappings**

UA Algorithm Id	Web Services Algorithm URI
HMAC-SHA1	<a href="http://www.w3.org/2000/09/xmldsig#hmac-sha1">http://www.w3.org/2000/09/xmldsig#hmac-sha1</a>
RSA-SHA1	<a href="http://www.w3.org/2000/09/xmldsig#rsa-sha1">http://www.w3.org/2000/09/xmldsig#rsa-sha1</a>
AES128-CBC	<a href="http://www.w3.org/2001/04/xmlenc#aes128-cbc">http://www.w3.org/2001/04/xmlenc#aes128-cbc</a>
AES196-CBC	<a href="http://www.w3.org/2001/04/xmlenc#aes196-cbc">http://www.w3.org/2001/04/xmlenc#aes196-cbc</a>
AES256-CBC	<a href="http://www.w3.org/2001/04/xmlenc#aes256-cbc">http://www.w3.org/2001/04/xmlenc#aes256-cbc</a>
RSA-PKCS1	<a href="http://www.w3.org/2001/04/xmlenc#rsa-1_5">http://www.w3.org/2001/04/xmlenc#rsa-1_5</a>
RSA-OAEP	<a href="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p</a>

A server issues a SCT by returning a *RequestSecurityTokenResponse* (RSTR) message to the client that sent the RST message. The SOAP action associated with the RSTR message must be “<http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/SCT>”.

The RSTR message has the components described in Table 19.

**Table 19 – RequestSecurityTokenResponse Message Components**

Component	Description
wst:RequestedSecurityToken	Specifies the new SCT or renewed SCT.
wst:RequestType	Must be “ <a href="http://schemas.xmlsoap.org/ws/2005/02/trust/Issue">http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</a> ” when creating a new SCT. Must be “ <a href="http://schemas.xmlsoap.org/ws/2005/02/trust/Renew">http://schemas.xmlsoap.org/ws/2005/02/trust/Renew</a> ” when renewing a SCT.
wst:Entropy	This contains the nonce specified by the server. The nonce is specified with the wst:BinarySecret element. The xenc:EncryptedData element is not used in UA because the message body must be encrypted with the public key from the client’s certificate.
wst:RequestedProofToken	This contains a wst:ComputedKey element which specifies the algorithm used to compute the shared secret key from the nonces provided by the client and the server. This value is always “ <a href="http://schemas.xmlsoap.org/ws/2005/02/trust/CK/PSHA1">http://schemas.xmlsoap.org/ws/2005/02/trust/CK/PSHA1</a> ”
wst:Lifetime	The revised lifetime for the SCT.

The body of the RSTR message must be signed with private key from the server’s application instance certificate. The body must also be encrypted with the public key from the client’s application instance certificate.

## 6 Message Encoding

### 6.1.1 Built-in Types

All UA message encodings are based on rules that are defined for a standard set of built-in types. These built-in types are used to construct structures, arrays and messages. The built-in types are described in Table 20.

**Table 20 – Built-in Data Types**

Name	Description
Null	An unspecified value.
Boolean	A two-state logical value (true or false).
SByte	An 8 bit signed integer value.
Byte	An 8 bit unsigned integer value.
Int16	A 16 bit signed integer value.
UInt16	A 16 bit unsigned integer value.
Int32	A 32 bit signed integer value.
UInt32	A 32 bit unsigned integer value.
Int64	A 64 bit signed integer value.
UInt64	A 64 bit unsigned integer value.
Float	An IEEE single precision (32 bit) floating point value.
Double	An IEEE double precision (64 bit) floating point value.
String	A sequence of Unicode characters.
DateTime	An instance in time.
Guid	A 16 byte value that can be used as a globally unique identifier.
ByteString	A sequence of bytes.
XmlElement	An XML element.
Nodeld	An identifier for a node in the address space of a UA server.
ExpandedNodeld	A Nodeld that allows the namespace URI to be specified instead of an index.
StatusCode	A numeric identifier for a error or condition that is associated with a value or an operation.
DiagnosticInfo	A structure that contains detailed error and diagnostic information associated with a StatusCode.
ExtensionObject	A structure that contains an application specific data type that may not be recognized by the receiver.
DataValue	A data value with an associated status code and timestamps.
Variant	A union of all of the types specified above.

## 6.2 UA Binary

### 6.2.1 General

The UA Binary Encoding is a data format developed to meet the performance needs of UA applications. This format is designed primarily for fast encoding and decoding, however, the size of the encoded data on the wire was also a consideration.

The UA binary encoding relies on several primitive data types with clearly defined encoding rules that can be sequentially written to or read from a binary stream. The set of primitive types and their encoding rules are described in more detail below.

A structure is encoded by sequentially writing the encoded form of the *value* of each field to the buffer. If a given field is also a structure then the values of its fields are written sequentially before writing the next field in the containing structure.

The UA binary encoding does not include any type or field name information because all UA applications are expected to have advance knowledge of the services and structures that they support. That said, the *ExtensionObject* type provides mechanism to embed data structures that the recipient may not understand in the UA binary stream.

### 6.2.2 Built-in Types

#### 6.2.2.1 Null

The *Null* type has no encoding and may only be used within an instance of *Variant*.

### 6.2.2.2 Boolean

A *Boolean* value must be encoded as a single byte where a value of 0 (zero) is false and any non-zero value is true.

It is recommended that encoders use the value of 1 to indicate a true value; however, decoders cannot assume that this will always be the case.

### 6.2.2.3 Integer

All integer types must be encoded as little endian values where the least significant byte appears first in the stream.

The Figure 8 illustrates how the value 1,000,000,000 (Hex: 3B9ACA00) should be encoded as a 32 bit integer in the stream.

00	CA	9A	3B
0	1	2	3

**Figure 8 – Encoding Integers in a Binary Stream**

### 6.2.2.4 Floating Point

All floating point values must be encoded with the appropriate [IEEE-754] binary representation which has three basic components: the sign, the exponent, and the fraction. The bit ranges assigned to each component depend on the width of the type. Table 21 lists the bit ranges for the supported floating point types.

**Table 21 – Supported Floating Point Types**

Name	Width (bits)	Fraction	Exponent	Sign
Float	32	0-22	23-30	31
Double	64	0-51	52-62	63

In addition, the order of bytes in the stream is significant. All floating point values must be encoded with the least significant byte appearing first (i.e. little endian).

The Figure 9 illustrates how the value -6.5 (Hex: C0D00000) should be encoded as a *Float*.

00	00	D0	C0
0	1	2	3

**Figure 9 – Encoding Floating Points in a Binary Stream**

### 6.2.2.5 String

All *String* values are encoded as a sequence of UTF-8 characters with a null terminator and preceded by the length in bytes.

The length is encoded as *Int32*. A value of 0 is used to indicate a ‘null’ string.

Figure 10 illustrates how the string “水 Boy” should be encoded in a byte stream.

	Length				水			B	o	y	\0
06	00	00	00	E6	B0	B4	42	6F	79	00	
0	1	2	3	4	5	6	7	8	9	10	11

**Figure 10 – Encoding Strings in a Binary Stream**

#### 6.2.2.6 DateTime

A *DateTime* value must be encoded as a 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601. This is the same as the binary representation of the WIN32 FILETIME type.

Not all platforms will be able to represent the full range of dates and times that can be represented with this encoding. For example, the UNIX time\_t structure only has a 1 second resolution and cannot represent dates prior to 1970. For this reason, a number of rules must be applied when dealing with date/time values that exceed the dynamic range of a platform's. These rules are:

- 1) A date/time value is encoded as 0 if:
  - a. The value equal to or earlier than 1601-01-01 12:00AM
  - b. The value is the earliest date that can be represented with the platform's encoding.
- 2) A date/time is encoded as *Int64.MaxValue* if:
  - a. The value is equal to or greater than 9999-01-01 11:59:59PM
  - b. The value is the latest date that can be represented with the platform's encoding
- 3) A date/time is decoded as the earliest time that can be represented on the platform if:
  - a. The encoded value is 0
  - b. The encoded value represents a time earlier than the earliest time that can be represented with the platform's encoding.
- 4) A date/time is decoded as the latest time that can be represented on the platform if:
  - a. The encoded value is *Int64.MaxValue*
  - b. The encoded value represents a time later than the latest time that can be represented with the platform's encoding.

These rules imply that the earliest and latest times that can be represented on a given platform are invalid date/time values and should be treated that way by applications.

#### 6.2.2.7 Guid

A *Guid* is encoded as sequence of unsigned integers (1 UInt32 value, 2 UInt16 values, 8 Byte values). The integers must be encoded individually in order to ensure the correct byte order is used for each integer.

#### 6.2.2.8 ByteString

A *ByteString* is encoded as sequence of bytes preceded by its length in bytes. The length is encoded as a 32-bit signed integer as described above.

If the length of the byte string is -1 then the byte string is 'null'.

#### 6.2.2.9 XmlElement

An *XmlElement* is an XML fragment serialized as UTF-8 string and then encoded as *ByteString*.

### 6.2.2.10 NodId

The components of a *NodId* are described the Table 22.

**Table 22 – NodId Components**

Name	Type	Description
Namespace	UInt32	<p>The index for a namespace URI. An index of 0 is used for UA defined <i>NodIds</i>. An index of 1 is the default namespace for the server and indicates that the identifier is unique within the server's address space. This value must be 0 for Uri and Guid identifier types because these types do not allow a namespace to be specified.</p>
Identifier Type	Enum	<p>The format and data type of the identifier The value may be one of the following:            NUMERIC - the value is an <i>Int32</i>;            STRING - the value is <i>String</i>;            URI - the value is <i>String</i>;            GUID - the value is a <i>Guid</i>;            OPAQUE - the value is a <i>ByteString</i>;            The namespace index must be 0 if identifier type is GUID or URI.</p>
Value	*	The identifier for a node in the address space of a UA server.

The encoding of a *NodId* varies according to the contents of the instance. For that reason the first byte of the encoded form indicates the format of the rest of the encoded *NodId*. The possible encoding formats are show in Table 23. The tables that follow describe the structure of the each possible format (they exclude the byte which indicates the format).

**Table 23 – NodId Encoding Values**

Name	Value	Description
Two Byte	0x00	A numeric value that fits into the two byte representation.
Four Byte	0x01	A numeric value that fits into the four byte representation.
Numeric	0x02	A numeric value that does not fit into the two or four byte representations.
String	0x03	A String value.
Uri	0x04	A URI value.
Guid	0x05	A Guid value.
ByteString	0x06	An opaque (ByteString) value.
NamespaceURI Flag	0x80	See discussion of <i>ExpandedNodId</i> in Clause 6.2.2.11.

The Two Byte Node Id encoding has the structure shown in Table 24.

**Table 24 – Two Byte NodId Binary Encoding**

Name	Data Type	Description
Identifier	Byte	<p>The Namespace is the default UA namespace (i.e. 0). The Identifier Type is 'Numeric'. The Identifier must be in the range 0 to 255.</p>

The Four Byte Node Id encoding has the structure shown in Table 25.

**Table 25 – Four Byte NodId Binary Encoding**

Name	Data Type	Description
Namespace	Byte	The Namespace must be in the range 0-255.
Identifier	UInt16	<p>The Identifier Type is 'Numeric'. The Identifier must be an integer in the range 0-65535.</p>

The Uri *NodId* encoding has the structure shown in Table 26.

**Table 26 – Uri NodId Binary Encoding**

Name	Data Type	Description
Identifier	String	The Identifier Type is 'Uri'. The Identifier is a String.

The Guid *NodId* encoding has the structure shown in Table 27.

**Table 27 – Guid NodId Binary Encoding**

Name	Data Type	Description
Identifier	Guid	The Identifier Type is 'Guid'. The Identifier is a Guid.

The standard *NodId* encoding has the structure shown in Table 28. The standard encoding is used for all formats that do not have an explicit format defined.

**Table 28 – Standard NodId Binary Encoding**

Name	Data Type	Description						
Namespace	UInt32	The Namespace index.						
Identifier	*	The identifier which is encoded according to the following rulexs: <table style="margin-left: 20px;"> <tr><td>NUMERIC</td><td>Int32</td></tr> <tr><td>STRING</td><td>String</td></tr> <tr><td>OPAQUE</td><td>ByteString</td></tr> </table>	NUMERIC	Int32	STRING	String	OPAQUE	ByteString
NUMERIC	Int32							
STRING	String							
OPAQUE	ByteString							

### 6.2.2.11 ExpandedNodId

An *ExpandedNodId* extends the *NodId* structure by explicitly specifying the namespace URI instead of using the namespace index. Any given instance of a *ExpandedNodId* may specify either the namespace index or the namespace URI. If both are specified by mistake then the encoder/decoder should ignore the namespace index.

The *ExpandedNodId* is encoded by first encoding a *NodId* as described in Clause 6.2.2.10 and then encoding namespace URI as a *String*.

An instance of an *ExpandedNodId* may still use the namespace index instead of the namespace URI. In this case, the namespace URI is not encoded in the stream. The presence of the namespace URI in the stream is indicated by setting the most significant bit of the encoding format byte for the *NodId*.

If the namespace URI is present then encoder must encode the namespace index as 0 in the stream when the *NodId* portion is encoded. The unused namespace index is included in the stream because it allows the encoder to reuse the functions used to encode/decode *NodIds*.

During decoding, the decoder should ignore any non-zero value for the namespace index if the namespace URI is present.

### 6.2.2.12 StatusCode

A *StatusCode* is encoded as a *UInt32*.

### 6.2.2.13 DiagnosticInfo

A *DiagnosticInfo* structure contains a number of fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

Some of the fields are indexes in a string table which is returned in the response header. Only the index of the string in this table is encoded. An index of -1 indicates that there is no value for the string.

**Table 29 – DiagnosticInfo Binary Encoding**

Name	Data Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bitxs: 0x01 Symbolic Id 0x02 Namespace 0x04 LocalizedText 0x08 Additional Info 0x10 Inner StatusCode 0x20 Inner DiagnosticInfo
Symbolic Id	Int32	A symbolic name for the status code.
Namespace	Int32	A namespace that qualifies the symbolic id.
LocalizedText	Int32	A human readable summary of the status code.
Additional Info	String	Detailed application specific diagnostic information.
Inner StatusCode	StatusCodes	A status code provided by an underlying system.
Inner DiagnosticInfo	DiagnosticInfo	Diagnostic info associated with the inner status code.

#### 6.2.2.14 ExtensionObject

An *ExtensionObject* is encoded as sequence of bytes prefixed by a identifier for its data type encoding and the number of bytes. The data type encoding is the *NodeID* of a data type encoding node in the address space. Clause 5.8 of [UA Part 3] completely describes how data type encoding node are used. That said, the data type encoding is simply a unique identifier that tells the decoder how to interpret the data in the *ExtensionObject*.

An *ExtensionObject* may be encoded by the application which means it is passed as a *ByteString* or an *XmlElement* to the encoder. In this case, encoder will be able to write the number of bytes in the object before it encodes the bytes. However, an *ExtensionObject* may know how to encode/decode itself which means the encoder cannot encode the number of bytes before it encodes the object. In this situation, the encoder calls the function to encode the *ExtensionObject* in the output stream and calculates the length by reading the position in the stream before and after the object body. If the stream supports seeking, the encoder should update the length in the stream. If seeking is not supported by the stream then the encoder should save the start and end position of the object in the *MessageFooter* appended to the end of the message when encoding is complete (see Clause 5.2.2).

When a decoder encounters an *ExtensionObject* it must check if it recognizes the data type encoding identifier. If it does then it can call the appropriate function to decode the object body. If the decoder does not recognize the type it must decode the object body as a stream of bytes. The decoder can either read the length of the body from the stream if it is available or skip to the end of the data and find the object in the table of contents. If the decoder cannot skip to the end of the data then it must generate a decoding error.

The serialized form of a *ExtensionObject* is shown in Table 30.

**Table 30 – Extension Object Binary Encoding**

Name	Data Type	Description
Encoding Mask		A bit mask that indicates which fields are present in the stream. The mask has the following bitxs: 0x01      Type Id 0x02      Binary Body 0x04      Xml Body
Type Id	ExpandedNodeID	The id for the data type encoding node in the server's address space. <i>ExtensionObjects</i> defined by the UA specification have a numeric node identifier assigned to them with a <i>NamespaceIndex</i> of 0. This value is required whenever an <i>ExtensionObject</i> is used as a message parameter. It may be omitted if the <i>ExtensionObject</i> is contained in a Variant structure.
Length	Int32	The length of the object body. If the length is not known then the it set to -1.
Body	*	The object body. A sequence of bytes for a Binary Body An XML element encoded as a UTF-8 string for an Xml Body

*ExtensionObjects* are used in two contexts: as values contained in *Variant* structures or as parameters in UA messages. If the *ExtensionObject* is a value in a *Variant* then the application will likely need to manually encode/decode the *ExtensionObject*. In this case, the application may omit the Type Id because it knows the receiver knows what data type to expect (e.g. the receiver is reading the value of a Variable which specifies the Data Type Id in an attribute). When an *ExtensionObject* is a message parameter it must have the Type Id specified and is usually serialized automatically by the encoder.

#### 6.2.2.15 Variant

An *Variant* is a union of the built in types.

The structure of an *Variant* is shown in Table 31.

**Table 31 – Variant Binary Encoding**

Name	Data Type	Description
Encoding Mask	Byte	The type of data encoded in the stream. The most significant bit indicates that the Variant contains an array. The lower 7 bits are the type identifier for a built in type.
Array Length	Int32	The number of elements in the array. This field is only present if the array bit is set in the encoding mask.
Value	*	The value encoded according to its data type. If the array bit is set in the encoding mask then each element in the array is encoded sequentially. Since many types have variable length encoding each element must be decoded in order. The value must not be an <i>Variant</i> but it could be an array of <i>Variants</i> . Arrays of <i>Byte</i> or <i>XmlElement</i> values are not permitted.

The possible types that can be encoded in an Variant are shown in Table 32.

**Table 32 – Variant Encoding Masks**

Mask	Type
0	Null
1	Boolean
2	SByte
3	Byte
4	Int16
5	UInt16
6	Int32
7	UInt32
8	Int64
9	UInt64
10	Float
11	Double
12	String
13	DateTime
14	Guid
15	ByteString
16	XmlElement
17	NodeId
18	ExpandedNodeId
20	StatusCodes
21	DiagnosticInfo
22	ExtensionObject
23	Variant
24	DataValue

#### 6.2.2.16 DataValue

A *DataValue* is always preceded by a mask that indicates which fields are present in the stream.

The fields of a *DataValue* are described in Table 33.

**Table 33 – Data Value Binary Encoding**

Name	Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bitxs: 0x01      Value 0x02      StatusCode 0x04      Server Timestamp 0x08      Source Timestamp
Value	Variant	The value. Not present if the value is <i>Null</i> .
Status	StatusCodes	The status associated with the value. Not present if the status 'Good'
Server Timestamp	DateTime	The server timestamp associated with the value. Not present if the timestamp is <i>DateTime.MinValue</i> .
Source Timestamp	DateTime	The source timestamp associated with the value. Not present if the timestamp is <i>DateTime.MinValue</i> .

### 6.2.3 Enumerations

Enumerations are encoded as Int32 values.

### 6.2.4 Arrays

Arrays are encoded as a sequence of elements preceded by the number of elements encoded as an Int32 value. If an array is 'null' then its length is encoded as -1. An array of zero length is different from an array that is 'null' so encoders and decoders must preserve this distinction.

### 6.2.5 Simple Types

There are a number of simple types defined in [UA Part 3], [UA Part 4] and [UA Part 5]. These types are mapped to built-in types as shown in Table 34.

**Table 34 – Simple Type Mapping**

Simple Type	Mapping
Date	DateTime
Time	DateTime
UtcTime	DateTime
Counter	UInt32
Index	UInt32
Duration	Int32
IntegerId	Int32
NumericRange	String

### 6.2.6 Structures

Structures are encoded as a sequence of fields in the order that they appear in the definition. The encoding for each field is determined by the data type for the field.

All fields specified in the complex type must be encoded.

Structures do not have a 'null' value. If an encoder written in programming language that allows structures to have null values then the encoder must create new an instance with default values for all fields and serialize that. Encoders must not generate an encoding error in this situation.

An example the following structure defined in with C# syntax:

```
class Type2
{
    int A;
    int B;
}

class Type1
{
    int      X;
    Type2[] Y;
    int      Z;
}
```

An instance of *Type1* which contains an array of two *Type2* instances would be encoded as seven 32-bit integers. The encoded form of an instance of *Type1* is shown in Table 35.

**Table 35 – Sample UA Binary Encoded Structure**

Field	Bytes	Value
Type ID	4	The identifier for Type1
Length	4	28
X	4	The value of field 'X'
Y Length	4	2
Y.A	4	The value of field 'Y[0].A'
Y.B	4	The value of field 'Y[0].B'
Y.A	4	The value of field 'Y[1].A'
Y.B	4	The value of field 'Y[1].B'
Z	4	The value of field 'Z'

### 6.2.7 Messages

Messages are encoded as *ExtensionObjects*. The parameters in each message are serialized in the same way the fields of a structure are serialized.

Each UA service described in [UA Part 4] has a request and response message. The data type id for the request message is always an even number. The data type id for the response message is an odd number one more than the request. The data type ids assigned to each service are in Annex A.1.

Messages may contain *ExtensionObjects* that do not have a pre-calculated size when messages are serialized into a stream that does not support seeking. The *MessageFooter* with the positions of these *ExtensionObjects* must be appended to the message. See Clause 5.2.2 for more information.

## 6.3 XML

### 6.3.1 Built-in Types

#### 6.3.1.1 General

Most built-in types are encoded in XML using the formats defined in [XML Schema Part 2] specification. Any special restrictions or usages are discussed below.

The prefix `xs:` is used to denote a symbol defined by the XML Schema specification.

#### 6.3.1.2 Null

A `Null` is represented by an empty element or with the `xs:nil` attribute set to "true".

#### 6.3.1.3 Boolean

A Boolean value is encoded as an `xs:boolean` value.

#### 6.3.1.4 Integer

Integer values are encoded using one of the sub types of the `xs:decimal` type. The mappings between the UA integer types and XML schema datatypes are shown in Table 36.

**Table 36 – XML Datatype Mappings for Integers**

Name	XML Type
SByte	xs:byte
Byte	xs:unsignedByte
Int16	xs:short
UInt16	xs:unsignedShort
Int32	xs:int
UInt32	xs:unsignedInt
Int64	xs:long
UInt64	xs:unsignedLong

**6.3.1.5 Floating Point**

Floating point values are encoded using one of the XML floating point types. The mappings between the UA floating point types and XML schema datatypes are shown in Table 37.

**Table 37 – XML Datatype Mappings for Floating Points**

Name	XML Type
Float	xs:float
Double	xs:double

**6.3.1.6 String**

A *String* value is encoded as an *xs:string* value.

**6.3.1.7 DateTime**

A *DateTime* value is encoded as an *xs:dateTime* value.

All *DateTime* values must be encoded as UTC times or with the time zone explicitly specified.

Correct:

2002-10-10T00:00:00+05:00

2002-10-09T19:00:00Z

Incorrect:

2002-10-09T19:00:00

It is recommended that all *xs:dateTime* values be represented in UTC format.

The earliest and latest date/time values that can be represented on a platform have special meaning and must be not be literally encoded in XML.

The earliest date/time value on a platform must be encoded in XML as '0001-01-01T00:00:00Z'.

The latest date/time value on a platform must be encoded in XML as '9999-12-31T11:59:59Z'

If a decoder encounters a *xs:dateTime* value that cannot be represented on the platform should convert the value to either the earliest or latest date/time that can be represented on the platform. The XML decoder should not generate an error if it encounters an out of range date value.

The earliest date/time value on a platform is equivalent to a 'null' date/time value.

### 6.3.1.8 Guid

A *Guid* is encoded as an xs:string with the format:

```
123456789-1234-1234-0102-030405060708
```

The string is created by formatting each of the *Guid* components as hexadecimal numbers.

**Table 38 – String Format for Guid Components**

Component	Data Type
Data1	UInt32
Data2	UInt16
Data3	UInt16
Data4	Byte[8]

The following string illustrates where the *Guid* components appear in the string:

```
<Data1>-<Data2>-<Data3>-<Data4[0:1]>-<Data4[2:8]>
```

All leading zeros must be included in the strings.

The XML schema for a *Guid* is:

```
<xs:complexType name="Guid">
  <xs:sequence>
    <xs:element name="String" type="xs:string" minOccurs="0" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.1.9 ByteString

A *ByteString* value is encoded as an xs:base64Binary value.

### 6.3.1.10 XmlElement

An *XmlElement* value is encoded as a xs:complexType with the following XML schema:

```
<xs:complexType name="XmlElement">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="1" processContents="lax" />
  </xs:sequence>
</xs:complexType>
```

*Xmlelements* may only be used inside Variant or ExtensionObject values.

### 6.3.1.11 NodId

An NodId value is encoded as a xs:string with URI syntax:

```
<schema>:n=<index>:<type>=<value>
```

The elements of the syntax are described in Table 39.

**Table 39 – Components of NodId URI**

Field	Type	Description
<schema>	String	The URI schema. Must always be 'opcua'. The scheme may be omitted in when a URI is used within an XML document.
<index>	UInt32	The namespace index formatted as a base 10 number. If the index is 0 then the entire 'n=0:' clause must be omitted.
<type>	Enum	A flag that specifies the identifier type. The flag has the following values: i        NUMERIC (Int32) s        STRING (String) u        URI (String) g        GUID (Guid) b        OPAQUE (ByteString)
<value>	*	The identifier encoded as string. The identifier is formatted using the XML data type mapping for the identifier type. Any character is permitted for the value. When the URI is encoded in an XML document the XML parser must escape characters that are not permitted in XML. These escape characters but be removed by the decoder.

Examples of *NodeIdxs*:

```
i=13
n=10:i=-1
n=10:s=Hello:World
u=http://opcfoundation.org/UA/1.0
g=09087e75-8e5e-499b-954f-f2a9603db28a
n=1:b=M/RbKBsRVkePCePcx24oRA==
```

The XML schema for a *NodId* is:

```
<xs:complexType name="NodeId">
  <xs:sequence>
    <xs:element name="Identifier" type="xs:string"
      minOccurs="0" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.1.12 ExpandedNodId

An *ExpandedNodId* value is encoded as a xs:complexType with the following XML schema:

```
<xs:complexType name="ExpandedNodeId">
  <xs:complexContent mixed="false">
    <xs:extension base="tns:NodeId">
      <xs:sequence>
        <xs:element name="NamespaceURI" type="xs:string"
          minOccurs="0" maxOccurs="1" nillable="true" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

If the *NamespaceUri* is missing then the *NodeId* should not include the namespace index (i.e. the namespace index should be set to 0).

### 6.3.1.13 StatusCode

A *StatusCode* is formatted in an *xs:string* as an 8 digit hexadecimal number.

The XML schema for a *StatusCode* is:

```
<xs:complexType name="StatusCode">
  <xs:sequence>
    <xs:element name="Code" type="xs:string" minOccurs="0" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.1.14 DiagnosticInfo

An *DiagnosticInfo* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="DiagnosticInfo">
  <xs:sequence>
    <xs:element name="SymbolicId" type="xs:int" minOccurs="0" />
    <xs:element name="NamespaceURI" type="xs:int" minOccurs="0" />
    <xs:element name="LocalizedText" type="xs:int" minOccurs="0"/>
    <xs:element name="AdditionalInfo" type="xs:string" minOccurs="0" />
    <xs:element name="InnerStatusCode" type="tns:StatusCode" minOccurs="0" />
    <xs:element name="InnerDiagnosticInfo" type="tns:DiagnosticInfo"
      minOccurs="0" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.1.15 ExtensionObject

An *ExtensionObject* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="ExtensionObject">
  <xs:sequence>
    <xs:element name="TypeId" type="tns:ExpandedNodeId" nillable="true" />
    <xs:element name="Body" minOccurs="0" nillable="true">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

The body of the *ExtensionObject* contains a single element which indicates how the body is encoded. If the element name is "Xml" then the body is encoded directly in the XML. If the element name is "ByteString" then the body is encoded as *xs:base64Binary* text. No other body encodings are permitted.

The *TypeId* is the identifier for the *DataTypeEncoding* node.

### 6.3.1.16 Variant

An *Variant* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="Variant">
  <xs:sequence>
    <xs:element name="Value" minOccurs="0" nillable="true">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

If the *Variant* represents a scalar value then it must contain a single child element with the name of the built-in type. For example, the single precision floating point value 3.1415 would be encoded as:

```
<Float>3.1415</Float>
```

If the *Variant* represents an array then it must contain a single child element with the prefix 'ListOf' and the name built-in type. For example an array of strings would be encoded as:

```
<ListOfString>
  <String>Hello</String>
  <String>World</String>
</ListOfString>
```

The complete set of built-in type names is found in Table 20.

*Variants* may not contain an instance of an *Variant*, however, *Variants* may contain an array of *Variants*.

### 6.3.1.17 DataValue

A *DataValue* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="DataValue">
  <xs:sequence>
    <xs:element name="Value" type="tns:Variant" minOccurs="0" nillable="true" />
    <xs:element name="StatusCode" type="tns:StatusCode" minOccurs="0" />
    <xs:element name="SourceTimestamp" type="xs:dateTime" minOccurs="0" />
    <xs:element name="ServerTimestamp" type="xs:dateTime" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.2 Enumerations

Enumerations are encoded as `xs:string` with the following syntax:

```
<symbol>_<value>
```

The elements of the syntax are described in Table 40.

**Table 40 – Components of Enumeration**

Field	Type	Description
<code>&lt;symbol&gt;</code>	String	The symbolic name for the enumerated value.
<code>&lt;value&gt;</code>	UInt32	The numeric value associated with enumerated value.

For example, the XML schema for the `SecurityRequirement` enumeration is:

```
<xs:simpleType name="SecurityRequirement">
  <xs:restriction base="xs:string">
    <xs:enumeration value="None_0" />
    <xs:enumeration value="Header_1" />
    <xs:enumeration value="Update_2" />
    <xs:enumeration value="All_3" />
  </xs:restriction>
</xs:simpleType>
```

### 6.3.3 Arrays

Arrays parameters are always encoded by wrapping the elements in a container element and inserting the container into the structure. The name of the container element should be the name of the parameter. The name of the element in the array must be the type name.

For example, the Read service takes an array of `ReadValueIds`. The XML schema would look like:

```
<xs:complexType name="ListOfReadValueId">
  <xs:sequence>
    <xs:element name="ReadValueId" type="tns:ReadValueId"
      minOccurs="0" maxOccurs="unbounded" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

The `nillable` attribute must be specified because XML encoders will drop elements in arrays if those elements are empty.

### 6.3.4 Structures

Structures are encoded as a `xs:complexType` with all of the fields appearing in a sequence. All fields are encoded as an `xs:element` and have `xs:minOccurs` and `xs:maxOccurs` set to 1.

For example, the Read service has a `ReadValueId` structure in the request. The XML schema would look like:

```
<xs:complexType name="ReadValueId">
  <xs:sequence>
    <xs:element name="NodeId" type="tns:NodeId" />
    <xs:element name="AttributeId" type="xs:int" />
    <xs:element name="IndexRange" type="xs:string" />
    <xs:element name="DataEncoding" type="tns:DataEncoding" />
  </xs:sequence>
</xs:complexType>
```

### 6.3.5 Messages

Messages are encoded as an `xs:complexType`. The parameters in each message are serialized in the same way the fields of a structure are serialized.

## 7 Formal Interface Definition

### 7.1 General

The UA specification is abstract specification that needs to be formally specified in a form that can be used by application developers.

### 7.2 Mapping File

The interface definition are created from the abstract specification by following a mechanical process. The same process is also used to create the software that implements the UA message encoding formats as well as programmers' APIs for different development environments. This mapping is done with a code generator developed by the OPC Foundation.

The input to the code generator is a mapping file which is an XML document that formally describes all of the services and structures defined in [UA Part 4] and [UA Part 5]. The syntax of the mapping file is described in Annex D.

The mapping file is not a normative document and is not published with the specification. It exists to automate the process of producing the APIs in different programming environments. The APIs are also not normative. The formal interface definitions are the normative document which implementers must use.

### 7.3 UA Native Mapping

The formal interface definition for the UA Native mapping is an OPC Binary Type Dictionary. This file defines the structure of all types and messages. The syntax for an OPC Binary Type Dictionary is described in Appendix B of [UA Part 3]. This dictionary captures normative names for types and their fields as well the order the fields appear when encoded. The data type of each field is also captured.

The UA native mapping type dictionary is attached in Annex B.

### 7.4 XML Web Service Mapping

The formal interface definition for the XML web service mapping is a WSDL. This WSDL defines all types and messages required to implement the UA services. It also defines the a standard port type for all UA services.

The WSDL include bindings and service elements to ensure standard WSDL tools can process the WSDL, however, these elements should be replaced by values appropriate for each server implementation.

The UA XML web service mapping WSDL file is attached in Annex C.

## 8 Managing Certificates

### 8.1 Certificate Stores

UA applications make use of X509 certificates to implement the various security features. All systems with UA applications must have a certificate store that can be used to manage these certificates. A single machine may have multiple stores that are identified by a path (i.e. the `StoreLocation`) and name (i.e. the `StoreName`). Common store locations are described in Table 41.

**Table 41 – Certificate Store Locations**

Logical Name	Description
Current User	A store location for certificates visible only to the currently logged in user.
Local Machine	A store location for certificates visible to all users on a machine.

Each certificate in a store may have additional access privileges attached to it so even if a store is visible to a user, the user may not be able to use every certificate in the store.

All application instance certificates for UA applications must be placed in a store with the name 'UA Applications'. This store may be in any store location, however, it is recommended that all UA applications use the 'Local Machine' location for UA application certificates.

Private keys associated with software certificates must not be placed in the certificate stores because certificate stores allow any administrator to access the keys. This means that vendors would lose control of their software certificates and run the risk that others will write applications that pretend to be theirs. See Clause 8.2.4 for more information on protecting software certificates.

## **8.2 Application Instance Certificates**

### **8.2.1 Installation**

All UA applications must have an application instance certificate. This certificate is an X509 certificate that is used by a single application installed on a single machine. Multiple instances of the same application running on the same machine would use the application instance certificate.

When an application is installed the install program must automatically generate a new certificate or prompt the user to provide one. If the installer automatically generates the certificate it must sign the certificate with the private key for the certificate.

### **8.2.2 Certificate Trust Lists**

Each UA application can only connect to other UA applications that it trusts. This is done by building a certificate trust list (CTL) for each application. The CTL for an application must be protected in the same way the private key for the application instance certificate is protected.

The CTL for an application should also be stored in certificate store with its certificate.

### **8.2.3 Certificate Revocation Lists**

An administrator for a machine may periodically download certificate revocation lists (CRLs) for one or more certification authorities (CA). These lists are stored in the certificate store with the CA certificate. UA applications must not connect to UA applications with revoked certificates.

CRLs may be downloaded automatically by a UA application. UA does not define a mechanism to do this.

### **8.2.4 Proof of Possession**

When a UA application connects to another UA application it must prove possession of the application certificate. It does this by taking random data provided by the other UA application and appending it to the DER form of the application instance certificate. It then applies the SHA-1 algorithm to calculate a digest and then signs that digest with the private key associated with the certificate.

The resulting signature is passed as a ByteString to the other UA application.

## 8.3 Software Certificates

### 8.3.1 General

UA applications may have one or more software certificates which contain a variety of information about the application including what certification tests it has passed. These content of the certificates is described in [UA Part 4], however, that description does not specify the physical format of the certificates.

UA supports two formats for the *SoftwareCertificate* structure: XML and UA Binary. Certificates using the XML format must be converted to the UA Binary format before they can be verified. The syntax for the UA Binary format follows the rules defined in Clause 6.2. The syntax for the XML format follows the rules defined in Clause 6.3.

A software certificate is signed by taking the UA binary encoded version and calculating the digest using the SHA-1 algorithm. This digest is then signed with the private key of a certificate owned by the organization responsible for testing the software for compliance. In many cases, the OPC Foundation will be the organization responsible for signing the software certificates.

UA applications that need to verify software certificates need to ensure the public keys for the testing authorities are loaded into the certificate store and are accessible to the application.

The bytes of the *SoftwareCertificate* and the signature are stored together in a structure called a *SignedSoftwareCertificate*.

### 8.3.2 Verification

When a UA application receives a *SignedSoftwareCertificate* from another UA application it must verify that the certificate is valid.

There are two steps:

- 1) Verify the issuer's signature in the *SignedSoftwareCertificate*.
- 2) Verify the X509 certificate contained in the *SoftwareCertificate*.

If any of these steps fail the receiver should reject the software certificate.

## 9 Server Discovery

### 9.1 Overview

UA applications will be deployed in many different environments which will have different requirements for a discovery. For that reason, UA defines conventions for locating UA services with different discovery protocols.

### 9.2 WS-Inspection

[WS Inspection] is a simple XML file based discovery mechanism. Each machine that has UA servers installed should provide a WS-Inspection file at the URL:

`http://<machine>/UA/Services.wsil`

**9.3 WS-Discovery****9.4 UDDI**

## Annex A Constants

### A.1 Service Type Ids

**Annex B Type Declarations for the UA Native Mapping**

**Annex C WSDL for the UA XML Mapping**

## Annex D Mapping File Syntax

### D.1 General

The other parts of this specification describe the UA services and information model in abstract terms without any reference to a specific implementation technology. This approach reflects the belief that specific implementation technologies will come and go, however, the overall UA architecture does not need to change.

In order to support this implementation independence, the UA specification defines a way to formally describe its services and information model in a form that can be interpreted by a computer program. The UA Type Dictionary is an XML based syntax that is intended to be used for this purpose. The choice of XML syntax is due to the availability of tools and parsers on a wide range of platforms.

A UA Type Dictionary is not the same as an XML Schema because UA does not require all of the features of XML Schema. For example, an XML Schema makes a distinction between attributes and child elements which is a complexity that the UA Type Dictionary does not require. In addition, the UA Type Dictionary uses terms and built-in data types that are consistent with the UA specification. That said, the syntax of the UA Type Dictionary can be described by an XML Schema. More importantly, any UA Type Dictionary document can be unambiguously converted to an XML Schema document by using the XML encoding rules described later in this document.

All data types that are defined by UA services or the UA information model will have UA Type Dictionary description. This description can be then used in a variety of ways to automate tasks related to developing an UA application including:

- Creating classes or structures in multiple programming languages (C/C++, .NET, Java);
- Serializing instances of the data types with different encodings (Binary, XML);
- Creating interface contracts for services for different platforms (WSDL, Indigo);

The UA Type Dictionary can also be used to describe any vendor defined data type; however, the UA specification does not require this. Vendor defined data types may use UA Type Dictionaries, XML Schema or other type description system that may be appropriate for the data type.

### D.2 Dictionary

A Dictionary is the root element of UA Type Dictionary document. It contains one or more simple types, complex types, enumerated types or services. A dictionary is used to group several related types that are managed together. All types defined in a dictionary must belong to the same namespace.

A dictionary is represented in the UA namespace by a data type dictionary node where the Node ID is constructed from the dictionary's namespace and name attributes. The value of the document property of the dictionary node is an XML document containing the dictionary.

The XML Schema definition of a dictionary element is:

```

<xss:element name="Dictionary">
  <xss:complexType>
    <xss:choice maxOccurs="unbounded">
      <xss:element name="SimpleType" type=" SimpleType"></xss:element>
      <xss:element name="ComplexType" type="ComplexType"></xss:element>
      <xss:element name="EnumeratedType" type="EnumeratedType"></xss:element>
      <xss:element name="Service" type="Service"></xss:element>
    </xss:choice>
    <xss:attribute name="Namespace" type="xss:string"
      use="required"></xss:attribute>
  </xss:complexType>
</xss:element>

```

The components of a dictionary are described in Table 42.

**Table 42 – Dictionary Element Components**

Name	Use	Description
Namespace	1..1	The URI of the namespace which provides the context for all types defined in the dictionary.
Simple Type	0..n	The description of a simple data type.
Complex Type	0..n	The description of a complex data type.
Enumerated Type	0..n	The description of an enumerated type.
Service	0..n	The description of a UA server type.

### D.3 Abstract Type

An Abstract Type is the base class for all other types. It defines attributes that are common to all types.

The XML Schema definition of a abstract type element is:

```

<xss:complexType name="AbstractType">
  <xss:attribute name="Name" type="xss:string"
    use="required"></xss:attribute>
</xss:complexType>

```

The components of a dictionary are described in Table 43.

**Table 43 – Abstract Type Element Components**

Name	Use	Description
Name	1..1	The name of the type. All types in a dictionary must have unique names.

### D.4 Simple Type

A Simple Type allows other types (including built-in types) to be re-used with a different identifier. This allows special semantics to be associated with a type without changing underlying structure of the type. For example, a 64-bit integer that represents the number of 100 nanosecond intervals since 1601-01-01 could be defined to a Simple Type called 'Gregorian Time'.

The XML Schema definition of a Simple Type element is:

```
<xss:complexType name="SimpleType">
  <xss:complexContent>
    <xss:extension base="AbstractType">
      <xss:attribute name="BaseType" type="xss:QName"
        use="required"></xss:attribute>
      <xss:attribute name="ArrayLength"
        type="xss:unsignedInt"></xss:attribute>
    </xss:extension>
  </xss:complexContent>
</xss:complexType>
```

The components of a simple type are described in Table 44.

**Table 44 – Simple Type Components**

Name	Use	Description
Base Type	1..1	The qualified name of the type that the simple type is derived from. This can be a built-in type, another type in the same dictionary or a type in another dictionary.
Array Length	0..1	The number of elements in an array. If specified then the simple type is an array of the base type. If this value is zero then the size of the array is set at runtime. Otherwise, the simple type is a fixed length array.

## D.5 Complex Type

A Complex Type represents structured data that contains one or more named fields. A complex type can exist independently as part of a dictionary or can be part of another complex type. A complex type that is defined within another complex type is called a nested complex type.

A complex type may be derived from another complex type. In this case, an instance of the type consists of the fields in the base type appearing in sequence followed by the fields in the derived type. The names of the fields in the derived type may not be the same as names in the base type.

The XML Schema definition of a Complex Type element is:

```
<xss:complexType name="ComplexType">
  <xss:complexContent>
    <xss:extension base="AbstractType">
      <xss:sequence>
        <xss:element name="Field" type="FieldType"
          maxOccurs="unbounded"></xss:element>
      </xss:sequence>
      <xss:attribute name="BaseType" type="xss:QName"></xss:attribute>
    </xss:extension>
  </xss:complexContent>
</xss:complexType>
```

The components of a complex type are described in Table 45.

**Table 45 – Complex Type Components**

Name	Use	Description
Base Type	1..1	The qualified name of the type that the complex type is derived from. This can be another complex type in the same dictionary or a complex type in another dictionary.

The XML Schema definition of a Field Type element is:

```
<xss:complexType name="FieldType">
  <xss:sequence>
    <xss:element name="ComplexType" type="ComplexType"
      minOccurs="0"></xss:element>
  </xss:sequence>
  <xss:attribute name="Name" type="xss:string"
    use="required"></xss:attribute>
  <xss:attribute name="ArrayLength" type="xss:unsignedInt"></xss:attribute>
  <xss:attribute name="SingularName" type="xss:string"></xss:attribute>
  <xss:attribute name="IsAttribute" type="xss:boolean"
    default="false"></xss:attribute>
  <xss:attribute name="DefaultValue" type="xss:string"></xss:attribute>
  <xss:attribute name="DataType" type="xss:QName"></xss:attribute>
</xss:complexType>
```

The components of a field type element are described in Table 46.

**Table 46 – Field Type Components**

Name	Use	Description
Name	0..1	A name for the field that is unique within the scope of the dictionary.
Data Type	0..1	The qualified name of the data type contained in the field. This can be a built-in type, another type in the same dictionary or a type in another dictionary. This attribute must not be specified if a nested complex type is defined.
Complex Type	0..1	This is a complex type that describes the data contained in the field. This element must not be specified if the data type attribute is specified.
Singular Name	0..1	The singular spelling of the field name that can be used in the generated code.
Is Attribute	0..1	Whether the field should be mapped to an attribute in an XML schema.
Default Value	0..1	The default value for the field.
Array Length	0..1	The number of elements in an array. If specified then the simple type is an array of the base type. If this value is zero then the size of the array is set at runtime. Otherwise, the simple type is a fixed length array.

## D.6 Enumerated Type

An enumerated type represents a simple type which has a finite set of named values. Each value for an enumerated type has a string and a numeric identifier.

The XML Schema definition of an enumerated type element is:

```

<xxs:complexType name="EnumeratedType">
  <xxs:complexContent>
    <xxs:extension base="AbstractType">
      <xxs:sequence>
        <xxs:element name="Value" maxOccurs="unbounded">
          <xxs:complexType>
            <xxs:simpleContent>
              <xxs:extension base="xxs:string">
                <xxs:attribute name="Index" type="xxs:int"></xxs:attribute>
              </xxs:extension>
            </xxs:simpleContent>
          </xxs:complexType>
        </xxs:element>
      </xxs:sequence>
    </xxs:extension>
  </xxs:complexContent>
</xxs:complexType>

```

The components of an enumerated type are described in Table 47.

**Table 47 – Enumerated Type Components**

Name	Use	Description
Value	1..n	A valid value for an instance of the type. Each value has numeric index associated with it. If the index is not explicitly specified in the type definition then a default index is one more than the previous value. If the first value has no index specified then it has a default index of zero.
Index	0..1	An integer value associated with the string. If not specified the enumerated value will have an integer value 1 more than the previous enumerated value. The first enumerated have has an index of 0 if it is not explicitly specified.

## D.7 Service Type

A Service Type describes an operation which a client may ask a server to perform. Each service consists of a request message, a response message and a set of faults.

The XML Schema definition of a service type element is:

```

<xxs:complexType name="Service">
  <xxs:complexType name="ServiceType">
    <xxs:complexContent>
      <xxs:extension base="AbstractType">
        <xxs:sequence>
          <xxs:element name="Request" minOccurs="0" maxOccurs="1">
            <xxs:complexType>
              <xxs:sequence minOccurs="0">
                <xxs:element name="Parameter" type="FieldType"
                  maxOccurs="unbounded"></xxs:element>
              </xxs:sequence>
            </xxs:complexType>
          </xxs:element>
          <xxs:element name="Response" minOccurs="0" maxOccurs="1">
            <xxs:complexType>
              <xxs:sequence minOccurs="0">
                <xxs:element name="Parameter" type="FieldType"
                  maxOccurs="unbounded"></xxs:element>
              </xxs:sequence>
            </xxs:complexType>
          </xxs:element>
        </xxs:sequence>
      </xxs:extension>
    </xxs:complexContent>
  </xxs:complexType>
</xxs:complexType>

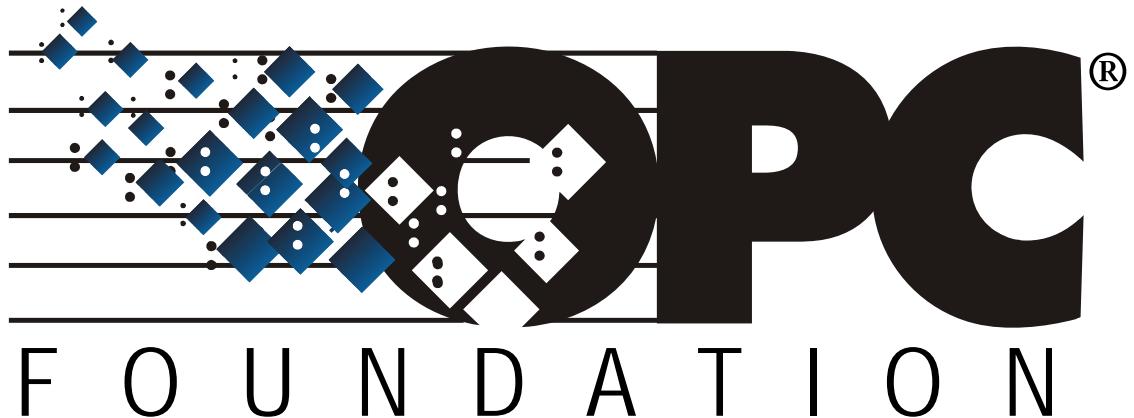
```

The components of a service type are described in Table 48.

**Table 48 – Service Type Components**

Name	Use	Description
Request	0..1	A sequence of named parameters that are passed by the client to the server. The request is used to initiate the service operation.
Response	0..1	A sequence of named parameters that are passed by the server back to the client after the service operation completes
Parameter	0..1	A named value that is part of either the request or the response for a service. The structure of the parameter is described by the field type element in Section D.5.





## **OPC Unified Architecture**

### **Draft Specification**

#### **Part 7: Profiles**

**Version 0.23**

**July 28th, 2006**

**Send comments to:**

**[UAcomments@opcfoundation.org](mailto:UAcomments@opcfoundation.org)**



Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture Part 7 Profiles	Date:	July 28th, 2006
Version:	Draft 0.23	Software Source:	MS-Word OPC UA Part 7 - Profiles Draft0.23 Specification.doc
Author:	OPC Foundation	Status:	Draft

## CONTENTS

	Page
1 Scope .....	2
2 Reference documents .....	2
3 Terms, definitions, and conventions.....	3
3.1 OPC UA part 1 terms .....	3
3.2 OPC UA part 2 terms .....	3
3.3 OPC UA part 3 terms .....	3
3.4 OPC UA profile terms.....	4
3.4.1 Conformance Unit.....	4
3.5 Abbreviations and symbols.....	4
3.6 Conventions for profile definitions.....	4
4 Overview .....	4
4.1 General .....	4
4.2 Conformance Unit .....	4
4.3 Profiles .....	4
4.4 Services .....	4
4.5 Information Functionality .....	4
5 Profiles .....	5
5.1 Services .....	5
5.2 Other Features.....	9
5.3 Named Profiles .....	13

## FIGURES

**Error! No table of figures entries found.**

**TABLES**

Table 1 - Secure Channel Services.....	5
Table 2 – Session Services .....	5
Table 3 – Node Management Services.....	6
Table 4 – View Services .....	6
Table 5 – Query Services .....	7
Table 6 – Attribute Services .....	7
Table 7 – Method Services .....	7
Table 8 – MonitoredItem Services.....	8
Table 9 – Subscription Services.....	8
Table 10 – Base Interaction.....	9
Table 11 – Security Information .....	9
Table 12 – Address Model.....	10
Table 13 – Data Access Model .....	11
Table 14 – Alarm & Conditions Model .....	11
Table 15 – Historical Access Model .....	11
Table 16 – Program Model .....	12
Table 17 – Miscellaneous items.....	12
Table 18 – Named Profiles .....	13
Table 19 –Profile vs. Conformance Unit.....	15

# OPC FOUNDATION

## UNIFIED ARCHITECTURE –

### FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

### AGREEMENT OF USE

#### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

#### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

#### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

#### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

#### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not

meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

## TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

## GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

## 1 Scope

This specification specifies the OPC Unified Architecture services.

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part2/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 6] OPC UA Specification: Part 6 – Mappings, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part6/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part8/>

[UA Part 9] OPC UA Specification: Part 9 – Alarms and Conditions, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part9/>

[UA Part 10] OPC UA Specification: Part 10 – Programs, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part10/>

[UA Part 11] OPC UA Specification: Part 11 – Historical Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part11/>

### 3 Terms, definitions, and conventions

#### 3.1 OPC UA part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Alarm
- 3) Attribute
- 4) Certificate
- 5) Client
- 6) Event
- 7) EventNotifier
- 8) Message
- 9) MonitoredItem
- 10) Node
- 11) Notification
- 12) NotificationMessage
- 13) Object
- 14) Object Type
- 15) Profile
- 16) Program
- 17) Reference
- 18) ReferenceType
- 19) RootNode
- 20) Subscription
- 21) View

#### 3.2 OPC UA part 2 terms

There are no [UA Part 2] terms used in this part.

#### 3.3 OPC UA part 3 terms

The following terms defined in [UA Part 3] apply.

- 1) branch node
- 2) hierarchical reference
- 3) ownership reference
- 4) source node
- 5) target node

### 3.4 OPC UA profile terms

#### 3.4.1 Conformance Unit

*Conformance Unit* defines a group of services or features that can be tested as a single unit. Conformance units are used to build a profile

### 3.5 Abbreviations and symbols

API	Application Programming Interface
DA	Data Access
HA	Historical Access
UA	Unified Architecture

### 3.6 Conventions for profile definitions

OPC UA Profiles contains tables that create titles and provided references to the various items that can be profiled. Subsequent tables provided named Conformance units that reference the titles for the items that are to be included in the listed Conformance Unit.

## 4 Overview

### 4.1 General

The UA multipart specification describes a large number of services and a variety of information models. These services and information models can be referred to as features of a server or client. Servers and clients need to be able to describe which of the features they support. Grouping of these features simplifies the display of this information; it also simplifies testing of the features. . The individual features are initially grouped in Conformance Units which are further grouped into Profiles.

### 4.2 Conformance Unit

A Conformance Unit is a small group of items that can be tested together. These grouping are used by compliance testing to verify functionality. Each Conformance Unit would have individual test cases. Features would not appear in more then one Conformance Unit.

### 4.3 Profiles

Profiles are named groupings of Conformance Units. The servers and clients in a UA system will provide the names of Profiles that they support. The grouping of features into Profiles is a dynamic activity. New Profiles can be defined in the future. A server or client can support multiple profiles. Multiple profiles may include the same Conformance Unit. Testing of Profile will entail testing the individual Conformance Units that comprise the Profile. In addition a profile may add additional test cases that apply to the interaction between Conformance Units or limitations on a conformance unit.

### 4.4 Services

Services are the calls that are provided by the server to expose functionality. A single call may expose different functionality depending on the information model that is being exposed.

### 4.5 Information Functionality

The UA specification contains multiple information models. This includes information models for Alarms and Conditions, Programs, Data Access, Historical Access and Security. Each of these information models may include multiple Conformance Units. These Conformance Units may be separated into multiple Profiles. Profiles may also include the entire information model.

## 5 Profiles

### 5.1 Services

The services listed in the following tables can be included in a Conformance Units. All of these services are described in detail in the [Part 4]. They are separated into tables based on the service set in which they are defined.

The Secure Channel Service Set is composed of a single conformance unit (see Table 1). These services must be support as a single unit. All servers must provide this functionality

Table 1 - Secure Channel Services

Conformance Unit	Service	Brief Description
Channel 1	GetSecurityPolicies	This base service is used to request information about a server's capabilities.
Channel 1	OpenSecureChannel	This base service is used by an OPC UA client to open a session on behalf of a specific system user.
Channel 1	CloseSecureChannel	This service is used to terminate a session

The Session Service Set is composed of a two conformance units (see Table 2). The CreateSession, ActivateSession, and CloseSession services must be support as a single unit. All servers must provide this functionality.

Table 2 – Session Services

Conformance Unit	Service	Brief Description
Session 1	CreateSession	This base service is used by an OPC UA client to open a session on behalf of a specific system user.
Session 1	ActivateSession	This service is used by an OPC UA client to submit its certificates to the server for validation.
Session 2	ImpersonateUser	This service is used to change the user ID associated with the given session
Session 1	CloseSession	This service is used to terminate a session

The Node Management Service Set is composed of multiple conformance units (see Table 3). The functionality in this service set is profile dependant.

Table 3 – Node Management Services

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Node 1	AddNodes	This service is used to add one or more nodes into the address space hierarchy.
Node 2	AddReferences	This service is used to add one or more references to one or more nodes.
Node 3	DeleteNodes	This service is used to delete one or more nodes from the address space.
Node 4	DeleteReferences	This service is used to delete one or more references of a node.

The View Service Set is composed of a single conformance unit (see Table 4). The View service set is core functionality of the OPC UA specification and as such must be supported as a single conformance unit.

Table 4 – View Services

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
View 1	BrowseProperties	This service is used to discover the properties supported by one or more specified nodes.
View 1	Browse	This service is used to discover the references of a specified node of a view. A primitive filtering capability is provided.
View 1	BrowseNext	This service is used to request the next set of Browse or BrowseNext response information that is too large to be sent in a single response.
View 1	TranslateBrowsePathsToNodeIds	This service is used to request the server to translate one or more browse paths to node ids.

The Query Service Set is composed of multiple conformance units (see Table 5). The Query functionality in this service set is profile dependant.

Table 5 – Query Services

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Query 1	QueryFirst	This Service is used to issue a <i>Query</i> request to the server.
Query 1	QueryNext	This service is used to obtain the next group of query results (when a QueryFirst call returned a continuation point)
Query 1	Basic Query Model	This includes support for basic content filter, but is limited to a single hop for references and does not support named instances in the RelatedTo Operator
Query 2	Advanced Query Model 1	This includes supporting a user specified number of hops for references in the content filters in the RelatedTo Operator
Query 3	Advanced Query Model 2	This includes support for “alias” in content filters

The Attribute Service Set is composed of multiple conformance units (see Table 6). The majority of the View service set is a core functionality of the OPC UA specification and as such must be supported as a single unit. The Query functionality in this service set is profile dependant.

Table 6 – Attribute Services

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Attrib 1	Read	This service is used to read one or more attributes of one or more nodes.
Attrib 2	HistoryRead	This service is used to read one or more historical attribute values of variables or read historical events
Attrib 3	Write	This service is used to write values to one or more attributes of one or more nodes.
Attrib 4	HistoryUpdate	This service is used to update historical values of one or more attributes of variables or update historical events.

The Method Service Set is composed of a conformance unit (see Table 7). The call functionality in this service set may be further broken down by individual methods.

Table 7 – Method Services

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Call 1	Call	This service is used to call (invoke) a method. Methods are invoked as instances of method types.

The MonitoredItem Service Set is composed of multiple conformance units (see Table 8).

**Table 8 – MonitoredItem Services**

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Monitor 1	CreateMonitoredItems	This service is used to create and add one or more monitored items to a subscription.
Monitor 1	ModifyMonitoredItems	This service is used to modify monitored items of a subscription.
Monitor 1	SetMonitoringMode	This service is used to set the monitoring mode for one or more monitored items of a subscription.
Monitor 2	SetTriggering	This service is used to create and/or delete triggering links for a triggering item.
Monitor 1	DeleteMonitoredItems	This service is used to remove one or more monitored items of a subscription.

The Subscription Service Set is composed of a conformance unit (see Table 9). The call functionality in this service set may be further broken down by individual methods.

**Table 9 – Subscription Services**

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Subscript 1	CreateSubscription	This service is used to create a subscription. Subscriptions monitor a set of monitored items for notifications and return them to the client in response to Publish requests. The lifetime of the subscription is three times the keep-alive interval negotiated by the server.
Subscript 1	ModifySubscription	This service is used to modify a subscription.
Subscript 1	SetPublishingMode	This service is used to enable sending of notifications on one or more subscriptions.
Subscript 1	Publish	This service is used for two purposes. First, it is used to acknowledge the receipt of notification messages for one or more subscriptions. Second, it is used to request the server to return a notification message or a keep-alive message.
Subscript 1	Republish	This service requests the subscription to republish a notification message from its retransmission queue.

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
Subscript 2	TransferSubscription	This service is used to transfer a subscription from one session to another.
Subscript 1	DeleteSubscription	This service is invoked by the client to delete one or more subscriptions that it has created and that have not been transferred to another client, or that have been transferred to it.

## 5.2 Other Features

Table 10 describes Base features related items that can be profiled. These items are defined in detail in [UA Part 6].

Table 10 – Base Interaction

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
TCP	UA TCP Transport	This include all items related to using the TCP transport instead of a soap transport
Soap	Soap 1.2 / HTTP transport	This includes all items related to using soap 1.2 / HTTP transport
WSS	WS-SecureConversation	Support for standard WS-SecureConversation
UASec	UA Security	Support for UA Security
Binary	Binary encoding	This is all items related to using a binary payload for the OPC UA messages
XML	XML encoding	This is all items related to using the XML payload for the OPC UA Messages

Table 11 describes Security related items that can be profiled. These items are defined in detail in [UA Part 6]. It is recommended that a server and client support as many of these options as possible for greatest interoperability.

Table 11 – Security Information

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
Encrypt 1*	AES – 256 – CBC	the shared-key (symmetric) encryption algorithm
Encrypt 2	AES – 128 – CBC	the shared-key (symmetric) encryption algorithm

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
Encrypt 3	RSA-PKCS-#1-V1.5	The RSA Public key Algorithms
Encrypt 4*	RSA-OAEP	The RSA Public key Algorithms
User 1	X.509 certificates	a public/private key pair
User 2	Kerberos tickets	Kerberos server
User 3*	User name	User name password combination
Key 1	RSA-1024	The RSA Crypto Algorithm with at least a 1024 bit length
Key 2*	RSA-2048	The RSA Crypto Algorithm with at least a 2048 bit length
Signature 1	RSA-v1.5-SHA1	
Signature 2*	RSA-v1.5-SHA256	
Signature 3	HMAC-SHA1	
Signature 4*	HMAC-SHA256	
Key Gen 1	P-SHA1	pseudo random number generator based on SHA1

Table 12 describes Address Model information related items that can be profiled. The details of these model items is defined in [UA Part 3] & [UA Part 5]

Table 12 – Address Model

<b>Conformance Unit</b>	<b>Address</b>	<b>Brief Description</b>
Diag	Diagnostic	
Event		
MCE	ModelChangeEvent & NodeVersion	are Model change events supported along with versioning
GenObjRef	Generate Objects reference	
DataDic	Expose data dictionaries	

<b>Conformance Unit</b>	<b>Address</b>	<b>Brief Description</b>
	Property Change events	

Table 13 describes Data Access data model related items that can be profiled. The details of this model are defined in [UA Part 8].

Table 13 – Data Access Model

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
DA Model	Data Access model	This include all items defined in the Data access model
DA Model2	Deadband	Support EU Range Deadband

Table 14 describes Alarm and Conditions data model related items that can be profiled. The details of this model are defined in [UA Part 9]

Table 14 – Alarm & Conditions Model

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
AC Model ????	Alarm and Condition Model	This includes all items defined in the Alarm and Condition model.
	Condition instances	

Table 15 describes Historical Data Access data model related items that can be profiled. The details of this model are defined in Part 11 Historical Data Access of this multipart specification

Table 15 – Historical Access Model

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
HA 1	HA support	General support for basic historical data access (this include read raw data)
HA 2	HA read processed	Support for reading processed data
HA 3	HA ATime	Support for reading Historical data at a time

<b>Conformance Unit</b>	<b>Feature</b>	<b>Brief Description</b>
HA Up 1	HA Insert	Support for historical data inserts
HA Up 2	HA Delete	Support for historical data delete
HA Up 3	HA Replace	Support for historical data Replace
HAE 1	HAE support	Support for Historical Alarm and event support.

Table 16 describes Command data model related items can be profiled.

Table 16 – Program Model

<b>Conformance Unit</b>	<b>Service</b>	<b>Brief Description</b>
PGM 1	Program Base	This includes Basic support for OPC Program Data Model.
PGM 2		Simple Program Objects are supported (limited control, can't be suspended).
PGM 3		Advanced Program Objects are supported (full control).

Table 17 describes miscellaneous items that need to be profiled.

Table 17 – Miscellaneous items

<b>Conformance Unit</b>	<b>Service / Object</b>	<b>Brief Description</b>
ServerRed1	Redundancy	Server Redundancy – transparent
ServerRed2	Redundancy	Server Redundancy – server based
ClientRed	Redundancy	Client Redundancy
Audit1	Auditing	Support for auditing
Audit2	Auditing	


### 5.3 Named Profiles

The profiles listed in Table 18 could be supported by server or clients. A server must implement all of the items in the profile, to be compliant with the profile. A client has to use the items listed in the profile (note it may not use all of the features in a profile).

Table 18 – Named Profiles

1A	Base Server Profile1	This profile provides the basic functionality that all servers would have to support to be considered a UA Server. This profile is directed towards the device that is capable TCP and UA Binary	Server
1B	Base Server Profile2	This profile provides the basic functionality that all servers would have to support to be considered a UA Server. This profile is direct towards the enterprise operation that is capable of support Soap and WS-SecureConversation	Server
1C	Base Client Profile1	This profile provides the basic functionality that all client would have to support to be considered UA Client	Client
2A	Advanced Security	This profile provides additional security capabilities	Server / Client
2B		A Server that would be used for XML based communication	Server
2C		A Server that would be used for TCPIP based communication	Server
2D		A client for communicating using TCP/IP using OPC Binary	Client
2E		A client for communicating using Soap using OPC Binary	Client
2F		A client for communicating using TCP/IP using XML	Client
2G		A client for communicating using Soap using XML	Client
3B	Advance UA Server	This profile provides additional server capabilities	Server / client
4A	DA 1	This profile provides the basic DA model	

5A	A&C 1	This profile provide basic support for the Alarm and Conditions	
5B	A&C 2		
6A	HA 1	This profile supports basic History Services	
6B	HA 2	This profile supports access processed History Historical Data	
6C	HA 3	This profile supports writes to History Historical Data	
6D	HA 4	This profile supports Access to Historical Alarm, condition and Event Data	
6E	HA 5	This profile provides support for writes of Historical Alarm, Condition ad Event data	
7a	Programs 1	This profile provides support for basic Programs	
7b	Programs 2	This profile provides support for a more advanced version of Programs (it is additive to Programs 1, which must also be supported)	
8a	No Time Clock	This Profile will be developed in the future for the environment in which no time clock exist and the limitation this creates	

The following a break down of what functionality is supported by what profile. Note some functionality may be in more then one profile and other profiles will require that a base profile exist. It is assumed that all server and client will support at least a base profile (1a/1b).

NOTES: Write should be part of base (since some base attributes are writeable)

Add table for methods and which are supported

Monitor and Subscribe – could combine some of these table items (same conformance item)

Regroup monitor and subscribe (some client would not need all of them thus they would require more conformance units – profile for server would require some as combined) (create/create/delete as minimum for client)

Bin/TCP – Base 1 device

Soap/WSS/XML – Base 2 - enterprise

Communication1 – communication 2 profiles for additional combinations

\* mark defaults

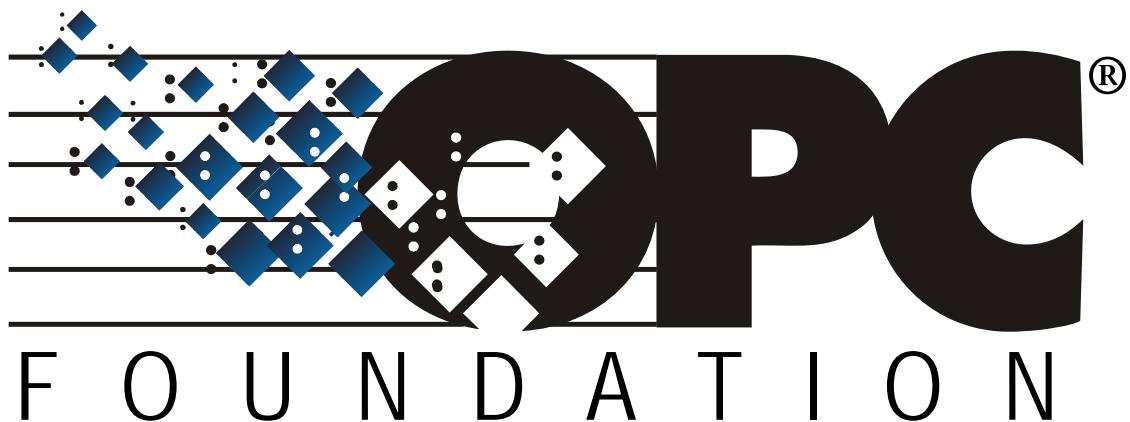
- Diagnostic Profile?

- Gateway Profile?

- create two table and create cross ref of client to servers

Table 19 –Profile vs. Conformance Unit





# **OPC Unified Architecture**

## **Specification**

### **Part 8: Data Access**

**Version 1.00**

**September 25, 2006**

Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture	Date:	September 25, 2006
<u>Data Access</u>			
Version:	Release 1.00	Software Source:	MS-Word OPC UA Part 8 - Data Access 1.00 Specification.doc
Author:	OPC Foundation	Status:	Release

## CONTENTS

	Page
<u>FOREWORD</u> .....	v
<u>AGREEMENT OF USE</u> .....	v
1 Scope.....	1
2 Reference documents .....	1
3 Terms, definitions, and abbreviations .....	1
3.1 OPC UA Part 1 terms.....	1
3.2 OPC UA Part 3 terms.....	1
3.3 OPC UA Part 4 terms.....	1
3.4 OPC UA Data Access terms.....	2
3.4.1 DataItem.....	2
3.4.2 AnalogItem .....	2
3.4.3 DiscreteItem .....	2
3.4.4 EngineeringUnits.....	2
3.5 Abbreviations and symbols .....	2
4 Concepts.....	3
5 Model.....	4
5.1 General .....	4
5.2 Variable Types .....	4
5.2.1 “Optional New” ModellingRule for DataAccess Properties.....	4
5.2.2 DataItemType .....	4
5.2.3 AnalogItemType .....	5
5.2.4 DiscreteItemType .....	6
5.3 Address Space Model .....	8
5.4 Attributes of DataItems .....	8
5.5 Property DataTypes .....	9
5.5.1 Overview .....	9
5.5.2 Range.....	9
5.5.3 EUInformation .....	10
6 Data Access specific usage of Services .....	11
6.1 PercentDeadband .....	11
6.2 Data Access Status Codes.....	11
6.2.1 Overview .....	11
6.2.2 Operation level result codes .....	11
6.2.3 LimitBits.....	12
6.2.4 SemanticsChanged .....	13

**FIGURES**

Figure 1 – OPC <i>DataItems</i> are linked to automation data .....	3
Figure 2 – <i>DataItem VariableType</i> Hierarchy.....	4
Figure 3 – Representation of <i>DataItems</i> in the <i>AddressSpace</i> .....	8

**TABLES**

Table 1 – <i>DataItemType</i> Definition .....	5
Table 2 – <i>AnalogItemType</i> Definition .....	5
Table 3 – <i>DiscreteItemType</i> Definition .....	6
Table 4 – <i>TwoStateDiscreteType</i> Definition .....	6
Table 5 – <i>MultiStateDiscreteType</i> Definition .....	7
Table 6 – <i>Range</i> Data Type Structure.....	9
Table 7 – <i>Range</i> Definition .....	9
Table 8 – <i>EUIInformation</i> Data Type Structure.....	10
Table 9 – <i>EUIInformation</i> Definition .....	10
Table 10 – <u>Bad</u> operation level result codes .....	12
Table 11 – <u>Uncertain</u> operation level result codes .....	12
Table 12 – <u>Good</u> operation level result codes .....	12

## OPC FOUNDATION

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance

with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here:

<http://www.opcfoundation.org/errata>

## 1 Scope

This specification is part of the overall OPC Unified Architecture specification series and defines the information model associated with Data Access (DA). It particularly includes additional *VariableTypes* and complementary descriptions of the *NodeClasses* and *Attributes* needed for Data Access, additional standard *Properties* and other information and behavior.

The complete address space model, including all *NodeClasses* and *Attributes*, is specified in [UA Part 3]. The services to detect and access data are specified in [UA Part 4].

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part5/>

## 3 Terms, definitions, and abbreviations

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- AddressSpace
- Node
- NodeClass
- Reference
- Subscription
- View

### 3.2 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

- DataVariable
- Object
- Property
- Variable
- VariableType

### 3.3 OPC UA Part 4 terms

The following terms defined in [UA Part 4] apply.

- Deadband

### 3.4 OPC UA Data Access terms

#### 3.4.1 DataItem

A *DataItem* represents a link to arbitrary, live automation data, i.e. data that represents currently valid information. Examples of such data are

- device data (such as temperature sensors)
- calculated data
- status information (open/closed, moving)
- dynamically-changing system data (such as stock quotes)
- diagnostic data

#### 3.4.2 AnalogItem

*AnalogItems* are *DataItems* that represent continuously-variable physical quantities. Typical examples are the values provided by temperature sensors or pressure sensors. OPC UA defines a specific *VariableType* to identify an *AnalogItem*. *Properties* describe the possible ranges of *AnalogItems*.

#### 3.4.3 DiscreteItem

*DiscreteItems* are *DataItems* that represent data that may take on only a certain number of possible values. Specific *VariableTypes* are used to identify *DiscreteItems* with two states or with multiple states. *Properties* specify the string values for these states.

#### 3.4.4 EngineeringUnits

*AnalogItems* represent continuously-variable physical quantities (e.g., length, mass, time, temperature) as integer or floating point values. *EngineeringUnits* specify the units of measurement for these quantities. This specification defines *Properties* to inform about the unit used for the *DataItem* value and about the highest and lowest value likely to be obtained in normal operation.

### 3.5 Abbreviations and symbols

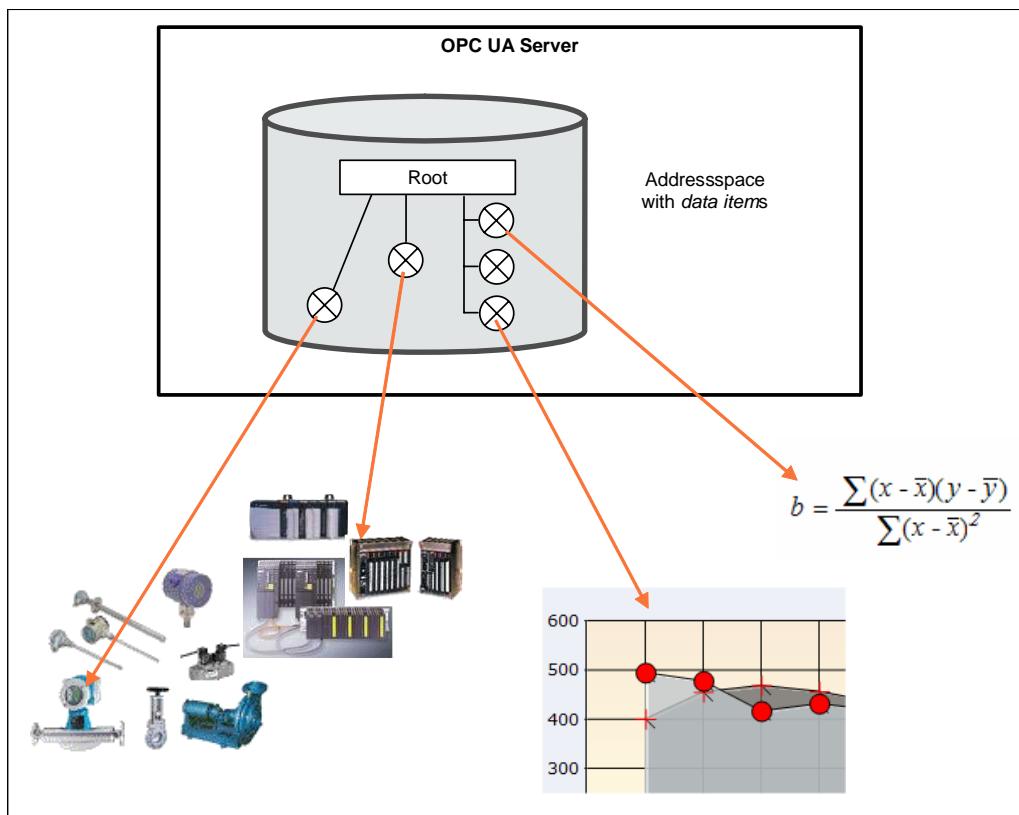
DA	Data Access
EU	Engineering Unit
UA	Unified Architecture

## 4 Concepts

Data Access deals with the representation and use of automation data in OPC UA Servers.

Automation data can be located inside the UA Server or on I/O cards directly connected to the UA Server. It can also be located in sub-servers or on other devices such as controllers and input/output modules, connected by serial links via field buses or other communication links. UA Data Access Servers provide one or more UA Data Access Clients with transparent access to their automation data.

The links to automation data instances are called *DataItems*. Which categories of automation data are provided is completely vendor-specific. Figure 1 illustrates how the *AddressSpace* of a UA server might consist of a broad range of different *DataItems*.



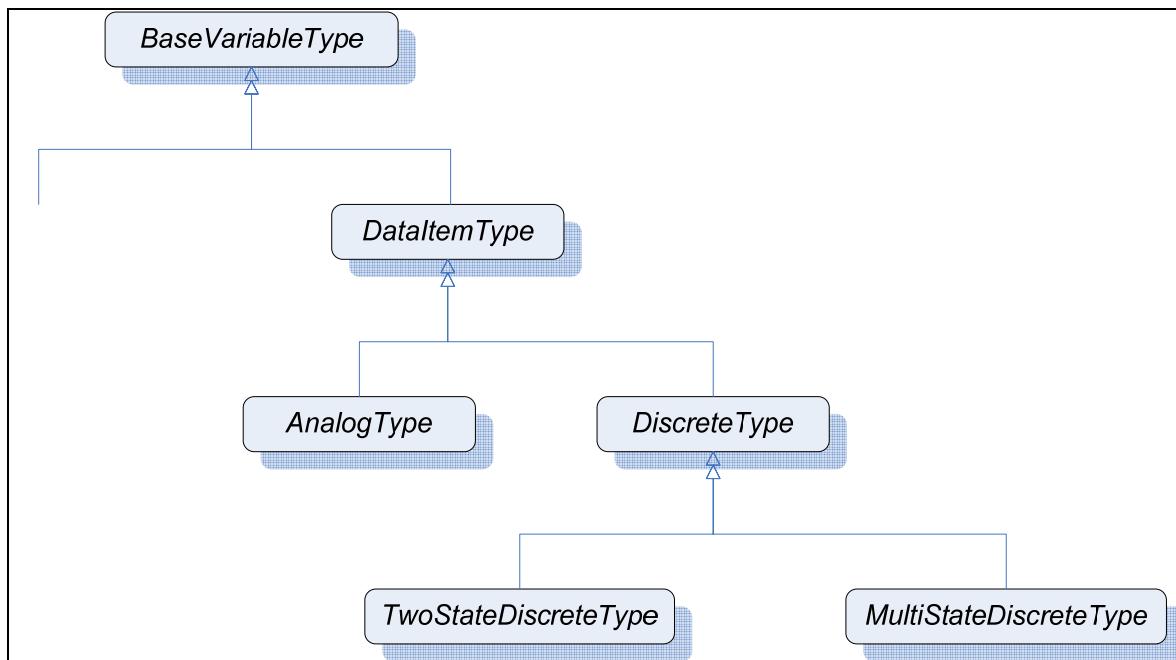
**Figure 1 – OPC *DataItems* are linked to automation data**

Clients may read or write *DataItems*, or monitor them for value changes. The services needed for these operations are specified in [UA Part 4]. Changes are defined as a change in status (quality) or a change in value that exceeds a client-defined range called a *Deadband*. To detect the value change, the difference between the current value and the last reported value is compared to the *Deadband*.

## 5 Model

### 5.1 General

The DataAccess model extends the variable model by defining *VariableTypes*. The *DataItem* type is the base type. *AnalogType* and *DiscreteType* (and its *TwoState* and *MultiState* subtypes) are specializations. Each of these *VariableTypes* can be further extended to form domain or server specific *DataItems*.



**Figure 2 – DataItem VariableType Hierarchy**

### 5.2 Variable Types

#### 5.2.1 “Optional New” ModellingRule for DataAccess Properties

*ModellingRules* are an extendable concept in OPC UA; [UA Part 3] defines the rules “None”, “Shared” and “New”. Some *DataAccess* properties, however, are optional and this part therefore defines the *OptionalNew ModellingRule*.

*OptionalNew* indicates that the *Node* referenced by a *TypeDefinitionNode* is optional, but when needed, the *ModellingRule New* is applied (see [UA Part 3]). This *ModellingRule* applies only to *Variables* referenced with a *HasProperty Reference*.

#### 5.2.2 DataItem Type

This *VariableType* defines the general characteristics of a *DataItem*. All other *DataItem* Types derive from it. The *DataItem* type derives from the *BaseVariableType* and therefore shares the variable model as described in [UA Part 3] and [UA Part 5]. It is formally defined in Table 1.

**Table 1 – DataItemDefinition**

Attribute	Value				
BrowseName	DataItemDefinition				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>BaseVariableType</i> defined in [UA Part 5]					
HasSubtype	VariableType	AnalogItemDefinition	Defined in Clause 5.2.3		
HasSubtype	VariableType	DiscreteItemDefinition	Defined in Clause 5.2.4		
HasProperty	Variable	Definition	String	.PropertyType	Optional New
HasProperty	Variable	ValuePrecision	Double	.PropertyType	Optional New

**Definition** is a vendor-specific, human readable string that specifies how the value of this *DataItem* is calculated. *Definition* is non-localized and will often contain an equation that can be parsed by certain clients.

Example:

*Definition* ::= "(TempA - 25) + TempB"

**ValuePrecision** specifies the maximum precision that the server can maintain for the item based on restrictions in the target environment.

*ValuePrecision* can be used for the following  *DataTypes*:

- For Float and Double values it specifies the number of digits after the decimal place.
- For DateTime values it indicates the minimum time difference in nanoseconds. E.g., a ValuePrecision of 20.000.000 defines a precision of 20 milliseconds.

The *ValuePrecision* property is an approximation that is intended to provide guidance to a client. A server is expected to silently round any value with more precision than it supports. This implies that a client may encounter cases where the value read back from a server differs from the value that it wrote to the server. This difference must be no more than the difference suggested by this property.

### 5.2.3 AnalogItemDefinition

This VariableType defines the general characteristics of an *AnalogItem*. All other *AnalogItem* Types derive from it. The *AnalogItemDefinition* derives from the *DataItemDefinition*. It is formally defined in Table 2.

**Table 2 – AnalogItemDefinition**

Attribute	Value				
BrowseName	AnalogItemDefinition				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>DataItemDefinition</i> defined in Clause 5.2.2					
HasProperty	Variable	InstrumentRange	Range	.PropertyType	OptionalNew
HasProperty	Variable	EURange	Range	PropertyParams	New
HasProperty	Variable	EngineeringUnits	EUIInformation	PropertyParams	OptionalNew

**InstrumentRange** defines the value range that can be returned by the instrument.

Example: *InstrumentRange* ::= {-9999.9, 9999.9}

The *Range* type is specified in clause 5.5.2.

**EURange** defines the value range likely to be obtained in normal operation. It is intended for such use as automatically scaling a bar graph display.

Sensor or instrument failure or deactivation can result in a returned item value which is actually outside this range. Client software must be prepared to deal with this. Similarly a client may attempt to write a

value that is outside this range back to the server. The exact behavior (accept, reject, clamp, etc.) in this case is server-dependent. However in general servers must be prepared to handle this.

Example: `EURange ::= {-200.0,1400.0}`

See also clause 6.1 for a special monitoring filter (*PercentDeadband*) which is based on the engineering unit range.

**EUInformation** specifies the units for the *DataItem*'s value (e.g., DEGC, hertz, seconds).

The *EUInformation* type is specified in clause 5.5.3.

#### 5.2.4 DiscreteItemType

This *VariableType* is an abstract type. I.e., no instances of this type can exist. However, it might be used in a filter when browsing or querying. The *DiscreteItemType* derives from the *DataItemType* and therefore shares all of its characteristics. It is formally defined in Table 3.

**Table 3 – DiscreteItemType Definition**

Attribute	Value				
BrowseName	DiscreteItemType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>DataItemType</i> defined in Clause 5.2.2					
HasSubtype	VariableType	TwoStateDiscreteType	Defined in Clause 5.2.4.1		
HasSubtype	VariableType	MultiStateDiscreteType	Defined in Clause 5.2.4.2		

##### 5.2.4.1 TwoStateDiscreteType

This *VariableType* defines the general characteristics of a *DiscreteItem* that can have two states. The *TwoStateDiscreteType* derives from the *DiscreteItemType*. It is formally defined in Table 4.

**Table 4 – TwoStateDiscreteType Definition**

Attribute	Value				
BrowseName	TwoStateDiscreteType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherit the <i>Properties</i> of the <i>DiscreteItemType</i> defined in Clause 5.2.4					
HasProperty	Variable	FalseState	LocalizedText	.PropertyType	New
HasProperty	Variable	TrueState	LocalizedText	PropertyParams	New

**TrueState** contains a string to be associated with this *DataItem* when it is TRUE. This is typically used for a contact when it is in the closed (non-zero) state.

e.g. "RUN", "CLOSE", "ENABLE", "SAFE", etc.

**FalseState** contains a string to be associated with this *DataItem* when it is FALSE. This is typically used for a contact when it is in the open (zero) state.

e.g. "STOP", "OPEN", "DISABLE", "UNSAFE", etc.

##### 5.2.4.2 MultiStateDiscreteType

This *VariableType* defines the general characteristics of a *DiscreteItem* that can have more than two states. The *MultiStateDiscreteType* derives from the *DiscreteItemType*. It is formally defined in Table 5.

**Table 5 – MultiStateDiscreteType Definition**

<b>Attribute</b>	<b>Value</b>				
BrowseName	MultiStateDiscreteType				
IsAbstract	False				
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>	<b>DataType</b>	<b>TypeDefinition</b>	<b>ModellingRule</b>
Inherit the <i>Properties</i> of the <i>DiscreteItemType</i> defined in Clause 5.2.4					
HasProperty	Variable	EnumStrings		LocalizedText[]	New

**EnumStrings** is a string lookup table corresponding to sequential numeric values (0, 1, 2, etc.)

Example:

- "OPEN"
- "CLOSE"
- "IN TRANSIT" etc.

Here the string "OPEN" corresponds to 0, "Close" to 1 and "IN TRANSIT" to 2.

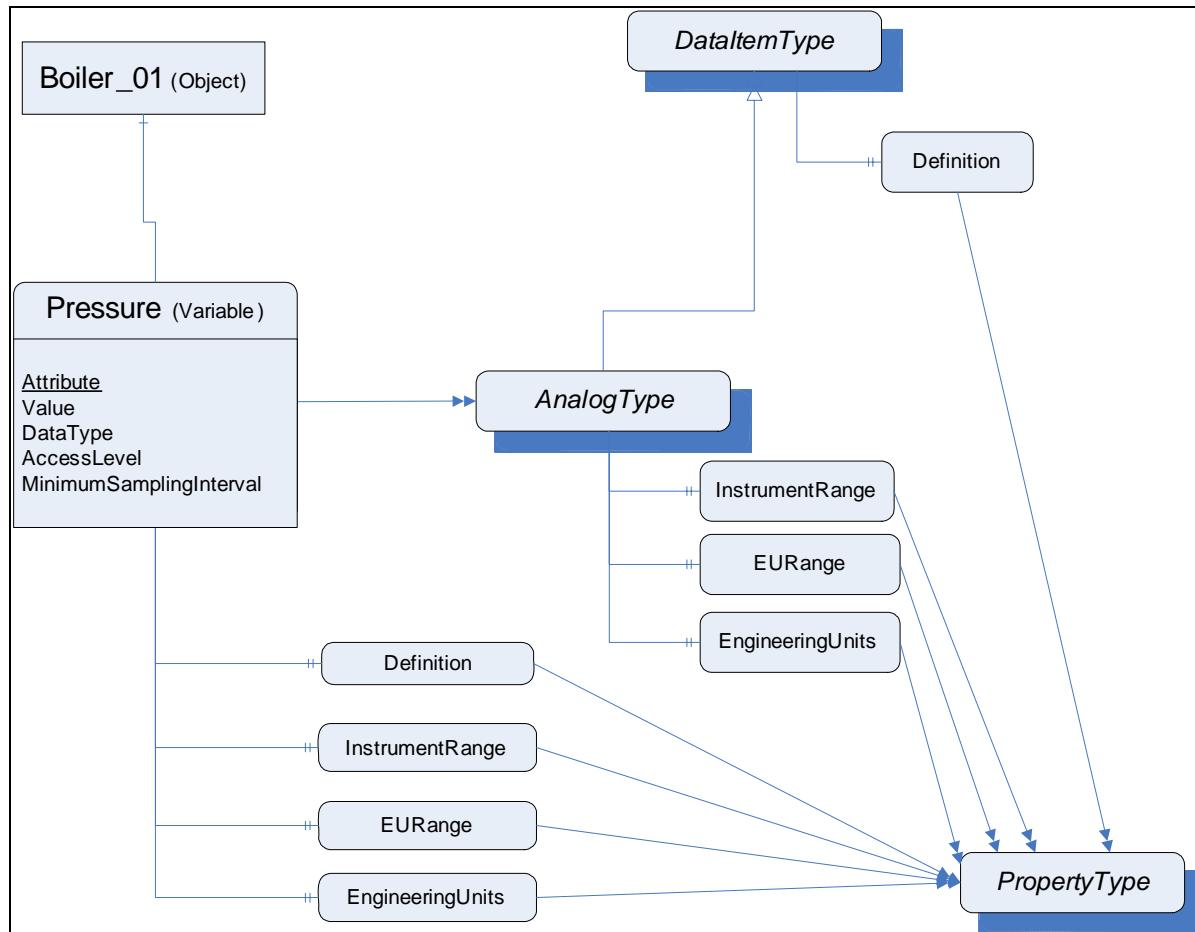
Clients should be prepared to handle item values outside the range of the list and robust servers should be prepared to handle writes of illegal values.

If the item contains an array of integer values this lookup table will apply to all elements in the array.

### 5.3 Address Space Model

*DataItems* are always defined as data components of other *Nodes* in the *AddressSpace*. They are never defined by themselves. A simple example of a container for *DataItems* would be a “Folder Object” but it can be an *Object* of any other type.

Figure 3 illustrates the basic *AddressSpace* model of a *DataItem* – in this case an *AnalogItem*.



**Figure 3 – Representation of *DataItems* in the *AddressSpace***

Each *DataItem* is represented by a *DataVariable* with a specific set of *Attributes*. The *TypeDefinition* reference indicates the type of the *DataItem* (in this case the *AnalogType*). Additional characteristics of *DataItems* are defined using *Properties*. The *VariableTypes* in Clause 5.2 specify which properties may exist. These *Properties* have been found useful for a wide range of Data Access clients. Servers that want to disclose similar information should use the OPC-defined *Property* rather than a vendor-specific one.

The above figure shows only a subset of *Attributes* and *Properties*. Other *Attributes* as defined for *Variables* in [UA Part 3] (e.g., *Description*) may also be available.

### 5.4 Attributes of DataItems

This section lists the *Attributes* of *Variables* that have particular importance for Data Access. They are specified in detail in [UA Part 3]. The following *Attributes* are particularly important for Data Access:

- Value

- `DataType`
- `AccessLevel`
- `MinimumSamplingInterval`

*Value* is the most recent value of the *Variable* that the server has. Its data type is defined by the *DataType Attribute*. The *AccessLevel Attribute* defines the server's basic ability to access current data and *MinimumSamplingInterval* defines how current the data are.

When a client requests the *Value Attribute* for reading or monitoring, the server always returns a *StatusCode* (the quality and the server's ability to access/provide the value) and, optionally, a *ServerTimestamp* and a *SourceTimestamp*. See [UA Part 4] for details on *StatusCodes* and the meaning of the two timestamps. Specific status codes for Data Access are defined in clause 6.2.

## 5.5 Property DataTypes

### 5.5.1 Overview

Following is a description of the data types used for Data Access properties defined in this part.

Standard *DataTypes* like *String*, *Boolean*, *Double* or *LocalizedText* are defined in [UA Part 3]. Their representation is specified in [UA Part 5].

### 5.5.2 Range

This structure defines the *Range* for a value. Its elements are defined in Table 6.

**Table 6 – Range Data Type Structure**

Name	Type	Description
Range	structure	
low	Double	Lowest value in the range.
high	Double	Highest value in the range.

Its representation in the *AddressSpace* is defined in Table 7

**Table 7 – Range Definition**

Attributes	Value
BrowseName	Range

### 5.5.3 EUInformation

This structure contains information about the *EngineeringUnits*. Its elements are defined in Table 8.

The structure has been defined such that standard bodies can incorporate their engineering unit definitions into OPC UA. Servers will use the *namespaceUri* in this structure to identify the proper organization.

**Table 8 – EUInformation Data Type Structure**

Name	Type	Description
EUInformation	structure	
namespaceUri	String	Identifies the organization (company, standard organization) that defines the <i>EUInformation</i> .
unitId	Int32	Identifier for programmatic evaluation. - 1 is used if a <i>unitId</i> is not available.
displayName	LocalizedText	The <i>displayName</i> of the engineering unit is typically the abbreviation of the engineering unit, e.g. "h" for hour or "m/s" for meter per second.
description	LocalizedText	Contains the full name of the engineering unit such as "hour" or "meter per second".

Its representation in the *AddressSpace* is defined in Table 9

**Table 9 – EUInformation Definition**

Attributes	Value
BrowseName	EUInformation

## 6 Data Access specific usage of Services

[UA Part 4] specifies the complete set of services. Those needed for the purpose of DataAccess are in particular:

- The *View* service set and *Query* service set to detect *DataItems*, and their *Properties*.
- The *Attribute* service set to read or write *Attributes* and in particular the value *Attribute*.
- The *MonitoredItem* and *Subscription* service set to set up monitoring of *DataItems* and to receive data change notifications.

### 6.1 PercentDeadband

The *DataChangeFilter* in [UA Part 4] defines the conditions under which a data change notification must be reported. This filter contains a deadband which can be of type *AbsoluteDeadband* or *PercentDeadband*. [UA Part 4] already specifies the behavior of the *AbsoluteDeadband*. This clause specifies the behavior of the *PercentDeadband* type.

#### **DeadbandType = PercentDeadband:**

For this type of deadband the *deadbandValue* is defined as the percentage of the *EURange*. That is, it applies only to *AnalogItems* with an *EURange* *Property* that defines the typical value range for the item. This range will be multiplied with the *deadbandValue* to generate an exception limit. An exception is determined as follows:

```
Exception if (absolute value of (last cached value - current value) >
(deadbandValue/100.0) * ((high-low) of EURange))
```

If the item is an array of values and any array element exceeds the *deadbandValue*, the entire monitored array is returned.

### 6.2 Data Access Status Codes

#### 6.2.1 Overview

This section defines additional codes and rules that apply to the *StatusCodes* when used for Data Access values.

The general structure of the *StatusCodes* is specified in [UA Part 4]. It includes a set of common operational result codes that also apply to Data Access.

#### 6.2.2 Operation level result codes

Certain conditions under which a *Variable* value was generated are only valid for automation data and in particular for device data. They are similar, but slightly more generic than the description of data quality in the various fieldbus specifications.

In the following, Table 10 contains codes with BAD severity, indicating a failure;

Table 11 contains codes with UNCERTAIN severity, indicating that the value has been generated under sub-normal conditions.

Table 12 contains GOOD (success) codes.

Note again, that these are the codes that are specific for Data Access and supplement the codes that apply to all types of data and are therefore defined in [UA Part 4].

**Table 10 – Bad operation level result codes**

Symbolic Id	Description
Bad_ConfigurationError	There is a problem with the configuration that affects the usefulness of the value.
Bad_NotConnected	The variable should receive its value from another variable, but has never been configured to do so.
Bad_DeviceFailure	There has been a failure in the device/data source that generates the value that has affected the value.
Bad_SensorFailure	There has been a failure in the sensor from which the value is derived by the device/data source. The limits bits are used to define if the limits of the value have been reached.
Bad_NoCommunication	Communications to the data source is defined, but not established, and there is no last known value available. This status/substatus is used for cached values before the first value is received.
Bad_OutOfService	The source of the data is not operational.
Bad_DeadbandFilterInvalid	The specified <i>PercentDeadband</i> is not supported, since an <i>EURange</i> is not configured.

**Table 11 – Uncertain operation level result codes**

Symbolic Id	Description
Uncertain_NoCommunicationLastUsable	Communication to the data source has failed. The variable value is the last value that had a good quality and it is uncertain whether this value is still current. The server timestamp in this case is the last time that the communication status was checked. The time at which the value was last verified to be true is no longer available.
Uncertain_LastUsableValue	Whatever was updating this value has stopped doing so. This happens when an input variable is configured to receive its value from another variable and this configuration is cleared after one or more values have been received. This status/substatus is not used to indicate that a value is stale. Stale data can be detected by the client looking at the timestamps.
Uncertain_SubstituteValue	The value is an operational value that was manually overwritten.
Uncertain_InitialValue	The value is an initial value for a variable that normally receives its value from another variable. This status/substatus is set only during configuration while the variable is not operational (while it is out-of-service).
Uncertain_SensorNotAccurate	The value is at one of the sensor limits. The Limits bits define which limit has been reached. Also set if the device can determine that the sensor has reduced accuracy (e.g. degraded analyzer), in which case the Limits bits indicate that the value is not limited.
Uncertain_EngineeringUnitsExceeded	The value is outside of the range of values defined for this parameter. The Limits bits indicate which limit has been reached or exceeded.
Uncertain_SubNormal	The value is derived from multiple sources and has less than the required number of <u>Good</u> sources.

**Table 12 – Good operation level result codes**

Symbolic Id	Description
Good_LocalOverride	The value has been Overridden. Typically this means the input has been disconnected and a manually-entered value has been "forced".

### 6.2.3 LimitBits

The bottom 16 bits of the *StatusCode* are bit flags that contain additional information, but do not affect the meaning of the *StatusCode*. Of particular interest for *DataItems* is the *LimitBits* field. In some cases, such as sensor failure it can provide useful diagnostic information.

Servers that do not support Limit have to set this field to 0.

#### 6.2.4 SemanticsChanged

The *StatusCode* also contains an informational bit called *SemanticsChanged*.

UA Servers that implement Data Access must set this Bit in notifications if one or several of the following *Properties* changes:

- *EngineeringUnits* (could create problems if the client uses the value to perform calculations)
- *EURange* (could change the behavior of a *Subscription* if a *PercentDeadband* filter is used)

It should not be changed for any of the other Data Access *Properties*.

Clients should not process the data value until they re-read the mentioned *Properties* associated with the *Variable*.



# **OPC Unified Architecture**

## **Draft Specification**

### **Part 9: Alarms**

**Draft Version 0.62**

**Mar 26, 2006**

**Send comments to:**  
**[UAcomments@opcfoundation.org](mailto:UAcomments@opcfoundation.org)**

Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture	Date:	Mar 26, 2006
<u>Alarms</u>			
Version:	Draft Specification 0.62	Software Source:	MS-Word OPC UA Part 9 Alarms Draft v0.61.doc
Author:	OPC Foundation	Status:	Draft Specification

## CONTENTS

	Page
FOREWORD.....	6
INTRODUCTION.....	7
1 Scope.....	8
2 Reference documents .....	8
3 Terms, definitions, and abbreviations .....	8
3.1 OPC UA Part 1 terms .....	8
3.2 OPC UA Part 3 terms.....	8
3.3 Abbreviations and symbols .....	9
4 Concepts.....	9
4.1 General .....	9
5 Information Model.....	10
5.1 General .....	10
5.2 Sub Condition model.....	10
5.2.1 Overview .....	10
5.2.2 Standard Sub Condition types .....	11
5.3 Condition Event Type.....	11
5.3.1 Overview .....	11
5.3.2 Enabled .....	12
5.3.3 Active .....	12
5.3.4 ActiveTime .....	13
5.3.5 InactiveTime .....	13
5.3.6 Unacknowledged.....	13
5.3.7 Sub condition .....	13
5.3.8 Condition State .....	13
5.4 Process Conditions .....	15
5.4.1 Process Condition .....	15
5.4.2 Level Alarm.....	16
5.4.3 Deviation Alarm .....	16
5.4.4 Rate Alarm.....	16
5.4.5 Change of State Alarm .....	16
5.4.6 Trip Alarm.....	16
5.5 Maintenance Conditions.....	16
5.5.1 Equipment Service Warning .....	16
5.5.2 Equipment Failure Alarm .....	16
5.6 System Conditions.....	17
5.6.1 System Warning.....	17
5.6.2 System Fault Alarm .....	17
5.6.3 Safety Fault Alarm .....	17
6 Services .....	17
6.1 Read .....	17
6.2 Historical Read .....	17
6.3 Condition Subscriptions .....	17
6.3.1 Refresh.....	17
6.4 Acknowledge .....	17

**FIGURES**

Figure 1 – Condition Types .....	10
Figure 2 – Condition State Diagram.....	14

**TABLES**

Table 1 – Sub Condition Type Definition .....	11
Table 2 – Standard Sub conditions .....	11
Table 3 – Condition Event Type Definition .....	12
Table 4 – Process Condition Type Definition.....	15
Table 5 – Level Alarm Type Definition .....	16
Table 6 – Acknowledge Request .....	18
Table 7 – Acknowledge Response.....	错误！未定义书签。

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is the specification for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### NON-EXCLUSIVE LICENSE AGREEMENT

The OPC Foundation, a non-profit corporation (the “OPC Foundation”), has defined a set of standard objects, interfaces and behaviors associated with the objects intended to promote interoperability between automation/control applications, field systems/devices, and business/office applications in the process control industry.

The OPC specifications, sample software (that demonstrates the implementation of the specifications, standard interface components deliverables and related documentation (collectively, the “OPC Materials”), form a set of standard objects, interfaces and behavior that are based on the technology being used in the automation marketplace, and includes the use of Microsoft Technology as well providing interoperability to non Microsoft platforms. The technology defines standard objects, methods, and properties for servers of real-time information like distributed process systems, programmable logic controllers, smart field devices and analyzers in order to communicate the information that such servers contain to standard compliant technologies enabled devices (e.g., servers, applications, etc.).

The OPC Foundation will grant to you (the “User”), whether an individual or legal entity, a license to use, and provide User with a copy of, the current version of the OPC Materials so long as User abides by the terms contained in this Non-Exclusive License Agreement (“Agreement”). If User does not agree to the terms and conditions contained in this Agreement, the OPC Materials may not be used, and all copies (in all formats) of such materials in User’s possession must either be destroyed or returned to the OPC Foundation. By using the OPC Materials, User (including any employees and agents of User) agrees to be bound by the terms of this Agreement.

#### LICENSE GRANT:

Subject to the terms and conditions of this Agreement, the OPC Foundation hereby grants to User a non-exclusive, royalty-free, limited license to use, copy, display and distribute the OPC Materials in order to make, use, sell or otherwise distribute any products and/or product literature that are compliant with the standards included in the OPC Materials.

All copies of the OPC Materials made and/or distributed by User must include all copyright and other proprietary rights notices include on or in the copy of such materials provided to User by the OPC Foundation.

The OPC Foundation shall retain all right, title and interest (including, without limitation, the copyrights) in the OPC Materials, subject to the limited license granted to User under this Agreement.

#### WARRANTY AND LIABILITY DISCLAIMERS:

User acknowledges that the OPC Foundation has provided the OPC Materials for informational purposes only in order to help User understand Microsoft’s OLE/COM technology. THE OPC MATERIALS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF PERFORMANCE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. USER BEARS ALL RISK RELATING TO QUALITY, DESIGN, USE AND PERFORMANCE OF THE OPC MATERIALS. The OPC Foundation and its members do not warrant that the OPC Materials, their design or their use will meet User’s requirements, operate without interruption or be error free.

IN NO EVENT SHALL THE OPC FOUNDATION, ITS MEMBERS, OR ANY THIRD PARTY BE LIABLE FOR ANY COSTS, EXPENSES, LOSSES, DAMAGES (INCLUDING, BUT NOT LIMITED TO, DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL, SPECIAL OR PUNITIVE DAMAGES) OR INJURIES INCURRED BY USER OR ANY THIRD PARTY AS A RESULT OF THIS AGREEMENT OR ANY USE OF THE OPC MATERIALS.

GENERAL PROVISIONS:

This Agreement and User's license to the OPC Materials shall be terminated (a) by User ceasing all use of the OPC Materials, (b) by User obtaining a superseding version of the OPC Materials, or (c) by the OPC Foundation, at its option, if User commits a material breach hereof. Upon any termination of this Agreement, User shall immediately cease all use of the OPC Materials, destroy all copies thereof then in its possession and take such other actions as the OPC Foundation may reasonably request to ensure that no copies of the OPC Materials licensed under this Agreement remain in its possession.

User shall not export or re-export the OPC Materials or any product produced directly by the use thereof to any person or destination that is not authorized to receive them under the export control laws and regulations of the United States.

The Software and Documentation are provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, the OPC Materials.

## INTRODUCTION

This specification defines the alarming model that extends the base event model for OPC Unified Architecture (UA) servers.

## Scope

This specification specifies the representation of alarms in the OPC Unified Architecture. Included is the specification of the representation of alarms in the OPC UA address space, the definition of alarm notification formats, and the definition of filters used in alarm subscriptions.

## 1 Reference documents

It is assumed that the reader is familiar with Web Services, and XML/SOAP. Information on these technologies can be found at the W3C web sites.

## 2 Terms, definitions, and abbreviations

### 2.1 OPC UA Part 1 terms

The following terms defined in Part 1 of this multi-part specification apply.

- 1) address space
- 2) complex data
- 3) information model
- 4) message
- 5) node
- 6) node hierarchy
- 7) node type
- 8) notification
- 9) notification message
- 10) object
- 11) object type
- 12) property
- 13) property type
- 14) reference
- 15) reference type
- 16) root node
- 17) server type
- 18) source node
- 19) subscription
- 20) view

### 2.2 OPC UA Part 3 terms

The following terms defined in Part 3 of this multi-part specification apply.

- 1) branch node
- 2) cardinality
- 3) leaf node
- 4) node attribute
- 5) target node

### 2.3 Abbreviations and symbols

A&E	Alarms and Events
API	Application Programming Interface
COM	Component Object Model
DA	Data Access
DCOM	Distributed COM
DX	Data Exchange
HDA	Historical Data Access
PLC	Programmable Logic Controller
SOAP	Simple Object Access Protocol
UA	Unified Architecture
UML	Unified Modelling Language
WSDL	Web Services Definition Language
XML	Extensible Mark-up Language

## 3 Concepts

### 3.1 General

This specification extends general eventing by defining an alarm model. The Alarm model provides the core behavior expected to support control system alarm management functionality. It is however not limited to control systems and can be extended to support the needs of other domains.

Alarms are defined as extensions of the basic event model. Alarms represent *conditions* of a system or one of its components. For example, a device or block may have a “LevelAlarm” and a “DeviationAlarm” condition associated with it.

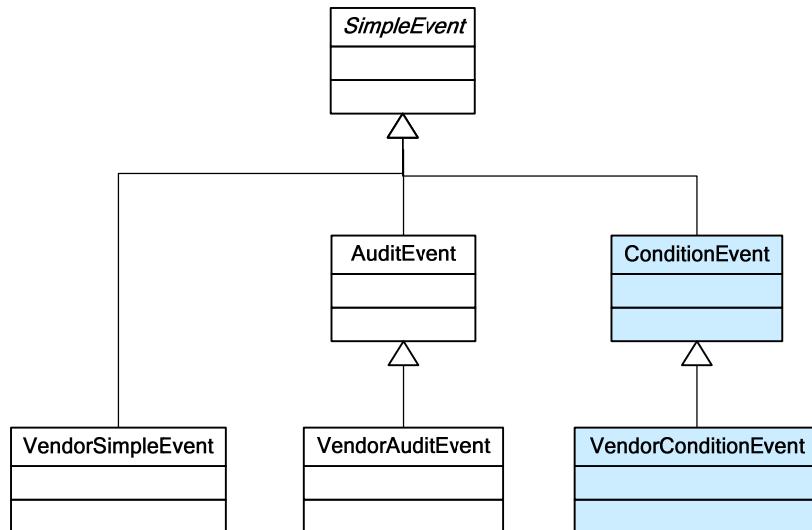
Furthermore, a condition may be defined to include multiple *sub-conditions*. For example, a LevelAlarm condition may include the “HighAlarm”, “HighHighAlarm”, “LowAlarm”, and “LowLowAlarm” sub-conditions.

## 4 Information Model

### 4.1 General

The alarm model extends the event model by defining condition events. This event type can be further extended to form domain or server specific condition types. The condition type introduces the concept of state to form the alarming model differentiating it from the basic event model. Unlike the other event types conditions are not transient.

Alarms are represented by Conditions in UA. Conditions may be optionally exposed in the address space in order to allow direct access to the state of conditions.



**Figure 1 – Condition Types**

### 4.2 Sub Condition model

#### 4.2.1 Overview

Sub conditions make up part of the overall state of a condition. For example a level alarm condition can be in a normal condition, a high alarm condition, etc. These states of a condition are referred to as sub conditions of the condition.

Sub conditions are exposed in the server's address space as an array of Sub Condition types. Sub condition types are always abstract and are only used by a condition to describe its present sub condition state.

**Table 1 – Sub Condition Type Definition**

Attribute	Value						
browse name	Sub Condition Type						
node identifier	9000						
Is abstract	True						
References	Node Class	Browse Name	Data Type	Node Id	Instance Definition	Type Definition	Instantiation Rule
has property	variable	Definition	String		-		new

The Definition variable provides the detailed definition of what is used to establish the sub condition. For example a high limit sub condition could be defined as being “measured value exceeds high limit value”.

#### 4.2.2 Standard Sub Condition types

Standard sub conditions are defined in Table 2 below. Servers may use this standard set and may define server specific sub condition types.

**Table 2 – Standard Sub conditions**

### 4.3 Condition Event Type

### 4.3.1 Overview

Conditions are specializations of events and therefore share the event model described in [part 3], [part 4] and [part 5].

Conditions are not just transient they exist in an active or inactive state. They can be enabled and disabled and acknowledged. In addition to generating event notifications like other event types instances of conditions may exist in the address space. An instance of a condition type allows clients to subscribe directly to them.

**Table 3 – Condition Event Type Definition**

Attribute	Value						
Reference	Node Class	Browse Name	Data Type	Node Id	Instance Definition	Type Definition	Instantiation Rule
has property	variable	Event Id	GUID	2283	-		new
has property	variable	Event Type	NodeId	2284	-		new
has property	variable	Source Node	NodeId	2285	-		new
has property	variable	Source Name	QName	2286	-		new
has property	variable	Time	UtcTime	2287	-		new
has property	variable	Description	Localized Text	2288	-		new
has property	variable	Severity	UInt16	2289	-		new
has property	variable	Enabled	Boolean		-		new
has property	variable	Active	Boolean		-		new
has property	variable	ActiveTime	UtcTime		-		new
has property	variable	InactiveTime	UtcTime		-		new
has property	variable	Unacknowledged	Boolean		-		new
Has property	variable	AckTime	UtcTime				
Has property	variable	AuditEntryID	String				
Has property	variable	Comment	Localized Text				
Has property	variable	Sub Condition	NodeId				

This *condition event type* inherits all *properties* of the *base event type*. Their semantic is defined in [part 5].

#### 4.3.2 Enabled

Clients may enable and disable conditions, and the resulting behaviour is illustrated in Figure 2. Additional behaviours are noted below:

- The server may choose to continue to test for a condition while it is disabled. However, no event notifications will be generated while the condition is disabled, nor can it be acknowledged while it is disabled.
- It is server-specific as to whether or not the following condition properties are defined while in the disabled state: Active, ActiveSubCondition, Quality, Acked, LastAckTime, SubCondLastActive, CondLastActive, LastInactive, AcknowledgerID, and Comment.
- On a refresh, no event notifications will be generated for disabled conditions.
- When enabled, the Time attribute associated with the “Condition Active” event notification will either be the time the condition is first discovered after enabling, or the time it became active (server-specific).

#### 4.3.3 Active

The value provides indicates if the current sub-conditions is considered an active condition. For example a level alarm with a sub-condition of High is considered active and therefore the value of this condition would be true. If however the sub-condition changed to normal then the value would be false.

#### 4.3.4 ActiveTime

The time of the last transition into the active state. This is used to correlate condition acknowledgements with a particular transition into the sub-condition.

#### 4.3.5 InactiveTime

The time of the last transition into the inactive state. This is used to correlate condition acknowledgements with a particular transition into the sub-condition.

#### 4.3.6 Unacknowledged

Unacknowledged indicates the current acknowledgement state of the condition. The server determines when a condition state change requires an acknowledgement. Normally this would be when a condition transitions from the inactive to an active state however this is not always true nor is it the only case that can set the unacknowledged indicator.

ackTime – The time of the last acknowledgement of this condition. If the condition has not been acknowledged the time is null.

AuditEntryID – The audit entry ID provided with the last acknowledgement of this condition. If the condition has not been acknowledged the entry will be null.

Comment – The optional comment supplied by with last acknowledgement of the condition. It will be null if not supplied or if the condition is not been acknowledged.

#### 4.3.7 Sub condition

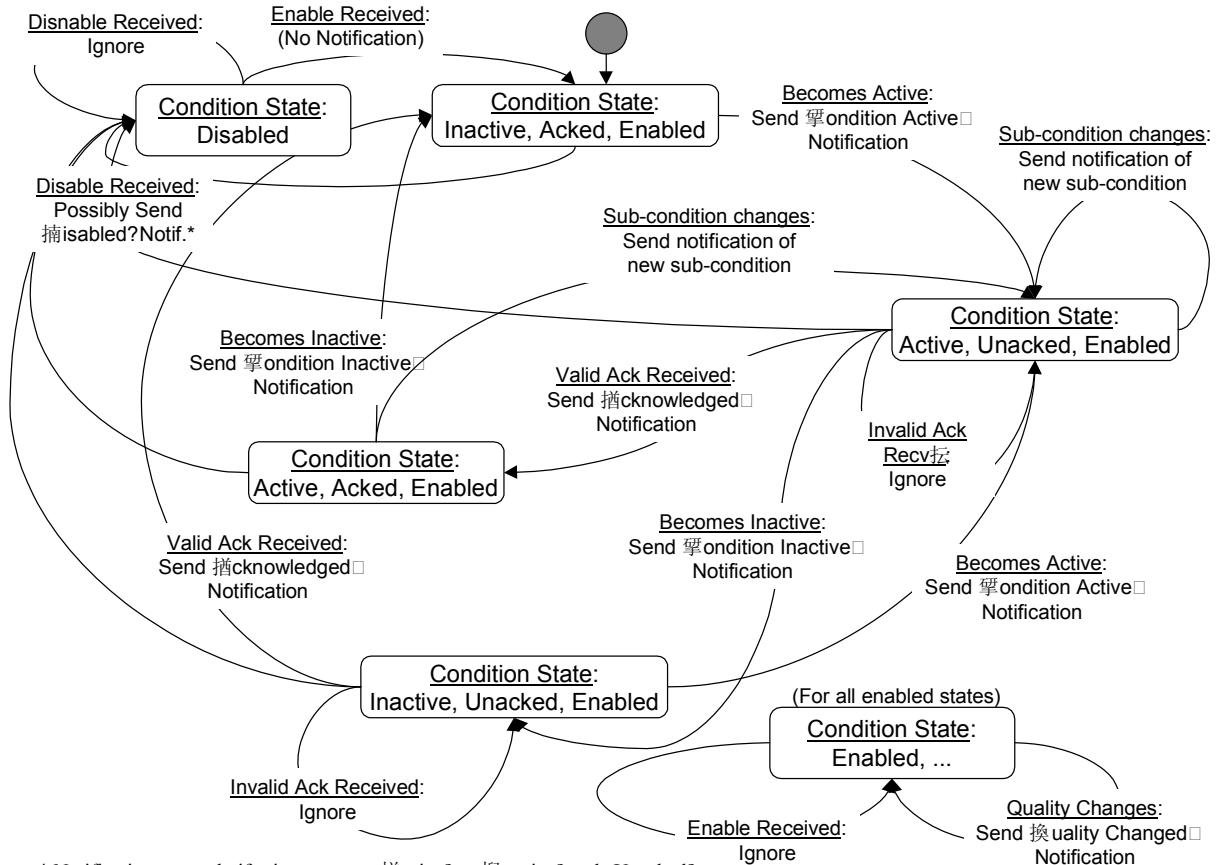
Sub Condition indicates the current state of the condition. The Active variable provides a Boolean indication of the conditions overall state. Sub conditions further define the state of the condition. The variable references one of the servers defined sub conditions see section 4.2.2 for a list of common sub conditions.

#### 4.3.8 Condition State

A Condition's state is defined by the combination of several properties of the condition. The state diagram below illustrates a typical server's handling of a condition.

Figure 2 shows a state machine for Condition which requires acknowledgement. Note that the intent of this diagram is to convey the expected behaviour of conditions, as viewed by a client of the Server. It is not intended to specify implementation, other than that the implementation must support the expected behaviour.

Each state transition is a Condition event. Condition Event notification messages are sent at each state transition.



**Figure 2 – Condition State Diagram**

Every condition event notification which requires acknowledgment includes the Name of the condition, the time that the condition most recently entered the active state or transitioned into a new sub-condition, and the sequence number which uniquely identifies the event notification. This information is provided back to the server by a Client when acknowledging the condition. This information is used by the Server to identify which specific event occurrence (state transition) is being acknowledged. If an acknowledgement is received with out-of-date information (this can occur due to latency in the system), the condition state does not become acknowledged.

Note that an acknowledgement affects the condition state only if it (the condition) is currently active or it is currently inactive and the most recent active condition was unacknowledged. If an inactive, unacknowledged condition again becomes active, all subsequent acknowledgements will be validated against the newly active condition state attributes. The server may optionally use the sequence number of the Notification to log acknowledgement of “old” condition activations, but such “late” acknowledgements have no affect on the current state of the condition.

Acknowledgment of the condition active state may come from the client or may be due to some logic internal to the Server. For example, acknowledgment of a related Condition may result in this Condition becoming acknowledged, or the Condition may be set up to automatically acknowledge itself when the condition becomes inactive.

For conditions that do not track or require acknowledgement, the state transitions are simpler - just between enabled/inactive, enabled-active, and disabled states.

Enabling a condition places it in the inactive-acked-enabled state. It is possible for the condition to become active very quickly after being enabled. No special scan/calculation are performed as part of the enabling action.

The server will generate audit events for enable and disable operations, rather than generating an event notification for each condition instance being enabled or disabled.

## 4.4 Process Conditions

### 4.4.1 Process Condition

**Table 4 – Process Condition Type Definition**

Attribute	Value						
References	Node Class	Browse Name	Data Type	Node Id	Instance Definition	Type Definition	Instantiation Rule
browse name		Process Condition Type					
node identifier		9002					
Is abstract		False					
has property	variable	Event Id	GUID	2283	-		new
has property	variable	Event Type	NodeId	2284	-		new
has property	variable	Source Node	NodeId	2285	-		new
has property	variable	Source Name	QName	2286	-		new
has property	variable	Time	UtcTime	2287	-		new
has property	variable	Description	Localized Text	2288	-		new
has property	variable	Severity	UInt16	2289	-		new
has property	variable	Enabled	Boolean		-		new
has property	variable	Active	Boolean		-		new
has property	variable	ActiveTime	UtcTime		-		new
has property	variable	InactiveTime	UtcTime		-		new
has property	variable	Unacknowledged	Boolean		-		new
Has property	variable	AckTime	UtcTime		-		new
Has property	variable	AuditEntryID	String		-		new
Has property	variable	Comment	Localized Text		-		new
Has property	variable	Sub Condition	NodeId		-		new

This *level alarm type* inherits all *properties* of the condition *event type*. Their semantic is defined in section 4.3.1.

#### 4.4.2 Level Alarm

**Table 5 – Level Alarm Type Definition**

Attribute	Value						
References	Node Class	Browse Name	Data Type	Node Id	Instance Definition	Type Definition	Instantiation Rule
browse name		Level Alarm Type					
node identifier		9003					
Is abstract		False					
has property	variable	Event Id	GUID	2283	-		new
has property	variable	Event Type	NodeId	2284	-		new
has property	variable	Source Node	NodeId	2285	-		new
has property	variable	Source Name	QName	2286	-		new
has property	variable	Time	UtcTime	2287	-		new
has property	variable	Description	Localized Text	2288	-		new
has property	variable	Severity	UInt16	2289	-		new
has property	variable	Enabled	Boolean		-		new
has property	variable	Active	Boolean		-		new
has property	variable	ActiveTime	UtcTime		-		new
has property	variable	InactiveTime	UtcTime		-		new
has property	variable	Unacknowledged	Boolean		-		new
Has property	variable	AckTime	UtcTime				
Has property	variable	AuditEntryID	String				
Has property	variable	Comment	Localized Text				
Has property	variable	Sub Condition	NodeId				

This *level alarm type* inherits all *properties* of the process condition *event type*. Their semantic is defined in section 5.4.1.

#### 4.4.3 Deviation Alarm

#### 4.4.4 Rate Alarm

#### 4.4.5 Change of State Alarm

#### 4.4.6 Trip Alarm

### 4.5 Maintenance Conditions

#### 4.5.1 Equipment Service Warning

#### 4.5.2 Equipment Failure Alarm

## 4.6 System Conditions

### 4.6.1 System Warning

### 4.6.2 System Fault Alarm

### 4.6.3 Safety Fault Alarm

## 5 Services

### 5.1 Read

Condition types can be instantiated in the UA address space. These instances can be used by the Read service like any other variable or property in the address space.

### 5.2 Historical Read

<ToDo>

### 5.3 Condition Data Subscriptions

Condition types can be instantiated in the UA address space. These instances can be used by the Subscription and monitor items services like any other variable or property in the address space.

### 5.4 Condition Event Subscriptions

Unlike events conditions maintain a state in the server. When a client establishes a subscription with the server the current state of conditions that are active or unacknowledged prior to the creation of the subscription may also be of interest. Using the optional filter “currentConditions” will result in the server issuing notifications for all conditions that are active or unacknowledged in addition to any subsequent notifications.

#### 5.4.1 Refresh

### 5.5 Acknowledge

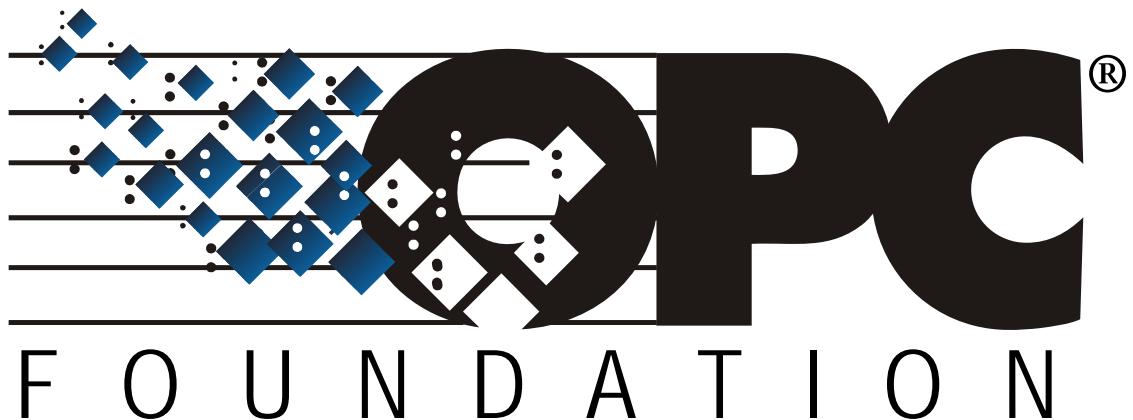
Conditions are acknowledged using the acknowledge call. The acknowledge call uses the UA base Call service as described in [part 4]. The details of the acknowledge call are described in Table 6.

**Table 6 – Acknowledge Request**

Name	Type	Description
<b>Request</b>		
requestHeader	RequestHeader	As defined in Part 4
methodId	structure	
objectId	NodeId	Set to the NodeId of the server object
methodTypeId	NodeId	Set to 'ConditionAcknowledge' method NodeId of the server object
inputArgument []	AckRequest	One or more Acknowledge Request structures
EventId	GUID	ID provided by the server notification of the condition being acknowledged
comment	LocalizedText	Optional comment to be associated with the acknowledgement
<b>Response</b>		
responseHeader	ResponseHeader	As defined in Part 4
outputArgument []	AckResponse	One Acknowledge response for each Acknowledge request
Status	StatusCode	Status of the acknowledgement
Diagnostic	String	Description of the status

Acknowledgments result in the server generating an audit event as explained in [part 3].





# **OPC Unified Architecture**

**Specification**

**Part 10: Programs**

**Version 1.00**

**January 29, 2007**

Specification Type	<u>Industry Standard Specification</u>		
Title:	OPC Unified Architecture <u>Part 10: Programs</u>	Date:	January 29, 2007
Version:	<u>Release 1.00</u>	Software Source:	<u>MS-Word OPC UA Part 10 - Programs 1.00 Specification.doc</u>
Author:	<u>OPC Foundation</u>	Status:	<u>Release</u>

## CONTENTS

	Page
FOREWORD .....	vii
<u>AGREEMENT OF USE</u> .....	vii
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, and abbreviations .....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 3 terms .....	2
3.3 OPC UA Program terms .....	2
3.3.1 Function .....	2
3.3.2 Program .....	2
3.3.3 Finite State Machine .....	2
3.3.4 ProgramType .....	2
3.3.5 Program Control Method .....	2
3.3.6 Program Invocation .....	2
3.4 Abbreviations and symbols .....	2
4 Concepts .....	3
4.1 General .....	3
4.2 Programs .....	4
4.2.1 Overview .....	4
4.2.2 Program Finite State Machine .....	4
4.2.3 Program States .....	5
4.2.4 State Transitions .....	6
4.2.5 Program State Transition Stimuli .....	6
4.2.6 Program Control Methods .....	6
4.2.7 Program State Transition Effects .....	7
4.2.8 Program Result Data .....	7
4.2.9 Program Lifetime .....	8
5 Model .....	8
5.1 General .....	8
5.2 ProgramType .....	9
5.2.1 Overview .....	9
5.2.2 ProgramType Properties .....	10
5.2.3 ProgramType Components .....	11
5.2.4 ProgramType Causes (Methods) .....	16
5.2.5 ProgramType Effects (Events) .....	17
5.2.6 ProgramTransitionAuditEventType .....	19
5.2.7 FinalResultData .....	20
5.2.8 ProgramDiagnosticType .....	20
Annex A - Program Example .....	22
A.1 Overview .....	22
A.2 DomainDownload Program .....	22
A.2.1 DomainDownload States .....	22
A.2.2 DomainDownload Transitions .....	23
A.2.3 DomainDownload Methods .....	23
A.2.4 DomainDownload Events .....	24

A.2.5 DomainDownload Model .....	24
----------------------------------	----

**FIGURES**

Figure 1 - Automation Facility Control .....	3
Figure 2 - Program Illustration .....	4
Figure 3 - Program States and Transitions .....	5
Figure 4 - Program Type.....	9
Figure 5 - Program FSM References.....	11
Figure 6 - ProgramType Causes and Effects .....	16
Figure 7 - Program Example 1 .....	22
Figure 8 – DomainDownload State Diagram .....	23
<i>Figure 8</i> .....	23
Figure 9 - DomainDownloadType Partial State Model .....	28
Figure 10 – Ready To Running Model .....	30
Figure 11 - Opening To Sending To Closing Model .....	31
Figure 12 .....	33
Figure 12 - Running To Suspended Model.....	33
Figure 13 .....	34
Figure 13 - Suspended To Running Model.....	34
<i>Figure 14</i> .....	35
Figure 14 - Running To Halted - Aborted Model .....	35
Figure 15 - Suspended To Aborted Model.....	36
<i>Figure 16</i> .....	36
Figure 16 - Running To Completed Model .....	37
Figure 17 - Sequence of Operations .....	38

**TABLES**

Table 1 - Program Finite State Machine.....	5
Table 2 - Program States .....	5
Table 3 - Program State Transitions .....	6
Table 4 - Program Control Methods .....	7
Table 5 - ProgramType.....	10
Table 6 - Program States .....	12
Table 7 - Program Transitions .....	14
Table 8 - ProgramType Causes .....	17
Table 9 - ProgramTransitionEventType.....	18
Table 10 – ProgramTransitionEvents .....	19
Table 11 - ProgramTransitionAuditEvent.....	20
Table 12 - ProgramDiagnosticType.....	20
Table 13 - DomainDownload States .....	23
Table 14 – DomainDownload Type .....	25
Table 15 - Transfer State Machine Type .....	26
Table 16 - Transfer State Machine - States .....	26
Table 17 - Finish State Machine Type.....	27
Table 18 - Finish State Machine - States.....	27
Table 19 – DomainDownload Type Property Attributes Variable Values .....	27
Table 20 - Additonal DomainDownload Transition Types .....	29
Table 21 - Start Method Additions .....	30
Table 22 - StartArguments .....	31
Table 23 - Intermediate Results Object .....	32
Table 24 - Immediate Result Data Variables .....	32
Table 25 - Final Result Data .....	36
Table 26 - Final Result Variables .....	37

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2006, 2007 OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these materials. Products developed using this specification may claim compliance or conformance with this specification if and

only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: <http://www.opcfoundation.org/errata>

## 1 Scope

This specification specifies the standard representation of *Programs* as part of the OPC Unified Architecture and its defined information model. This includes the description of the *NodeClasses*, standard *Properties*, *Methods* and *Events* and associated behaviour and information for *Programs*.

The complete address space model including all *NodeClasses* and *Attributes* is specified in [UA Part 3]. The services such as those used to invoke the *Methods* used to manage *Programs* are specified in [UA Part 4].

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.1 or later

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

## 3 Terms, definitions, and abbreviations

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] of this multi-part specification apply.

- 1) EventType
- 2) Information Model
- 3) Method
- 4) Node
- 5) NodeClass
- 6) Notification
- 7) Object
- 8) ObjectInstance
- 9) ObjectType
- 10) Program
- 11) Reference
- 12) ReferenceType
- 13) Service
- 14) Session
- 15) Subscription

## 16) Variable

### 3.2 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

1. TypeDefinition Node
2. Property
3. Variable

### 3.3 OPC UA Program terms

The following terms are defined for UA Programs.

#### 3.3.1 Function

A Function is a programmatic task performed at a server or device, usually accomplished by computer code execution.

#### 3.3.2 Program

A Program is a complex *Function* in a server or underlying system that can be invoked and managed by an OPC UA Client.

#### 3.3.3 Finite State Machine

A Finite State Machine is a sequence of states and valid state transitions along with the causes and effects of those state transitions that define the actions of a *Program* in terms of discrete stages.

#### 3.3.4 ProgramType

A ProgramType is an *ObjectType Node* that represents the type definition of a *Program* and is a subtype of the *StateMachineType*.

#### 3.3.5 Program Control Method

A Program Control Method is a *Method* specified by this specification having specific semantics designed for the control of a *Program* by causing a state transition.

#### 3.3.6 Program Invocation

A Program Invocation is a unique *Object* instance of a *Program* exiting on a Server. The Program Invocation is distinguished from other *Object* instances of the same *ProgramType* by the object node's unique browse path.

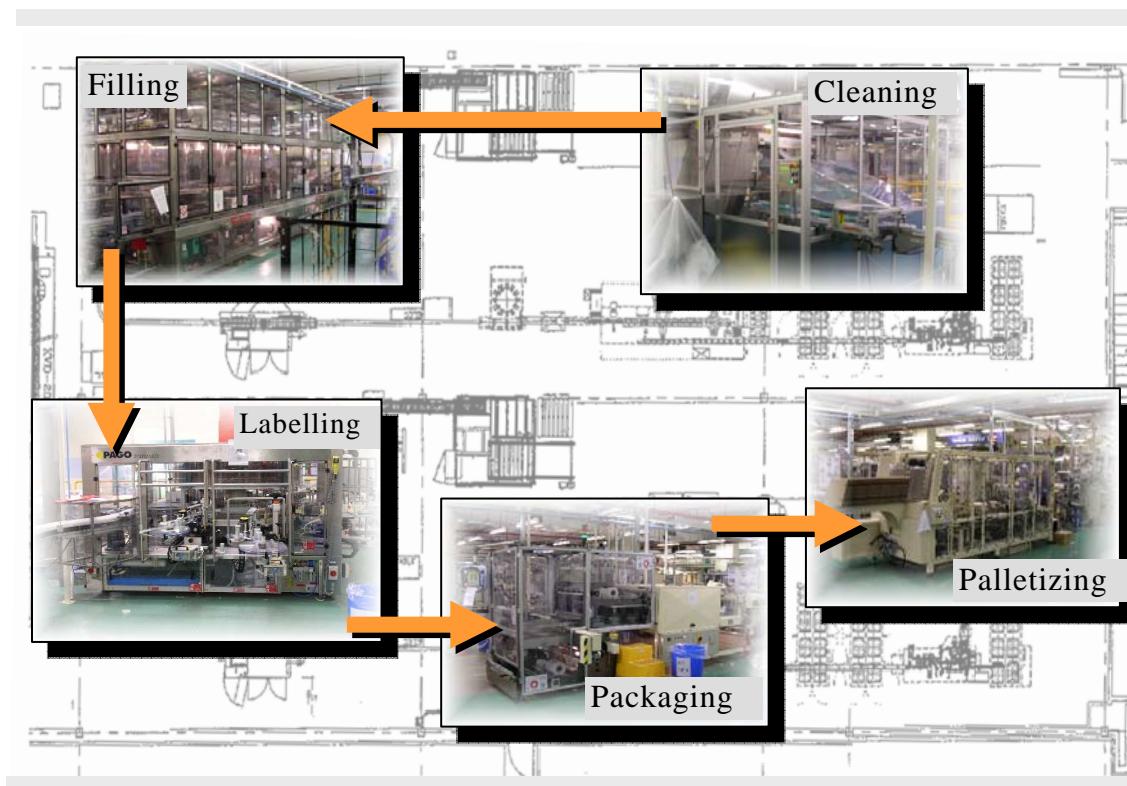
### 3.4 Abbreviations and symbols

API	Application Programming Interface
DA	Data Access
FSM	Finite State Machine
HMI	Human Machine Interfaces
PCM	Program Control Method
PGM	Program
PI	Program Invocation
PLC	Programmable Logic Controller
UA	Unified Architecture
UML	Unified Modelling Language

## 4 Concepts

### 4.1 General

Integrated automation facilities manage their operations through the exchange of data and coordinated invocation of system functions. Services are required to perform the data exchanges and to invoke the functions that constitute system operation. These functions may be invoked through human machine Interfaces, cell controllers, or other supervisory control and data acquisition type systems. OPC UA defines *Methods* and *Programs* as an interoperable way to advertise, discover, and request these functions. They provide a normalizing mechanism for the semantic description, invocation of, and result reporting of these functions. Together *Methods* and *Programs* complement the other OPC UA Services and *ObjectTypes* to facilitate the operation of an automation environment using a client server hierarchy.



**Figure 1 - Automation Facility Control**

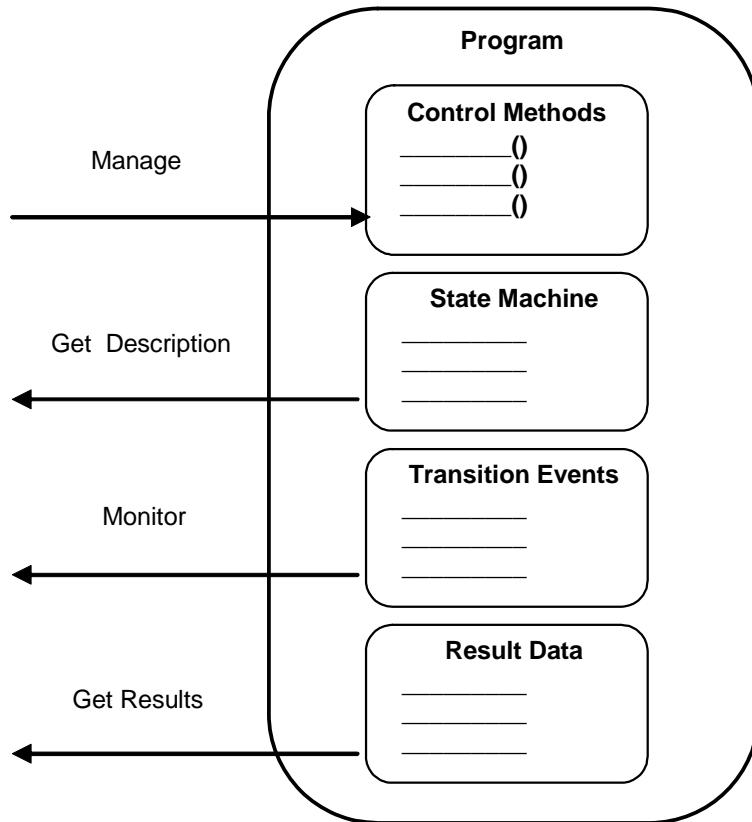
*Methods* and *Programs* model functions typically having different scopes, behaviours, lifetimes, and complexities in OPC Servers and the underlying systems. These functions are **not** normally characterized by the reading or writing of data which is accomplished with the OPC UA Attribute service set.

*Methods* represent basic functions in the server that can be invoked by a client. *Programs* by contrast, model more complex, stateful functionality in the system. For example, a method call may be used to perform a calculation or reset a counter. A *Program* is used to run and control a batch process, execute a machine tool part program, or manage a domain download. *Methods* and their invocation mechanism are described in [UA Part 3] – Address Space Model and [UA Part 4] - Services. This specification describes the extensions to, or specific use of the core capabilities defined in the first seven parts of the OPC UA multi-part specification required for *Programs*. Support for the feature described in this specification are described in [UA Part 7] - Profiles

## 4.2 Programs

### 4.2.1 Overview

*Programs* are complex functions in a server or underlying system that can be invoked and managed by an OPC UA Client. *Programs* can represent any level of functionality within a system or process in which client control or intervention is required and progress monitoring is desired.



**Figure 2 - Program Illustration**

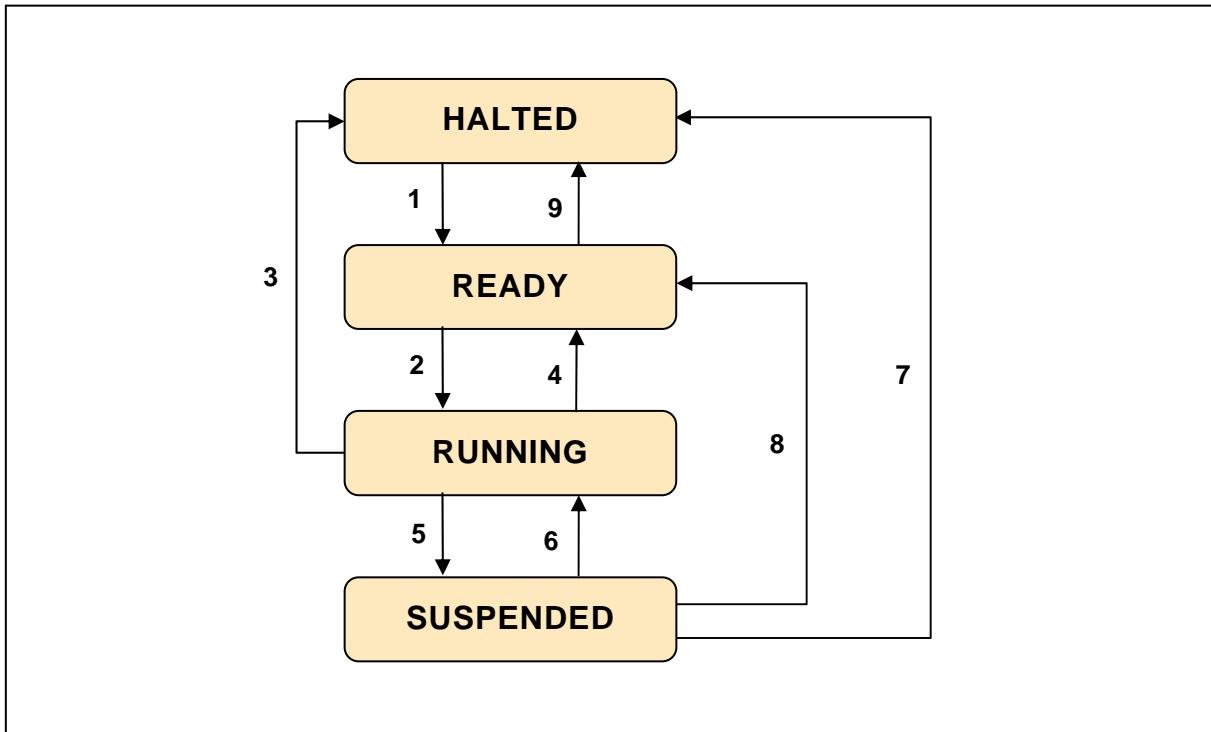
*Programs* are state full, transitioning through a prescribed sequence of states as they execute. Their behaviour is defined by a *Program Finite State Machine* (PFSM). The elements of the PFSM describe the phases of a *Program*'s execution in terms of valid transitions between a set of states, the stimuli or causes of those transitions, and the resultant effects of the transitions.

### 4.2.2 Program Finite State Machine

The statuses, transitions, causes and effects that compose the Program Finite State Machine are listed in Table 1 and illustrated in Figure 3.

**Table 1 - Program Finite State Machine**

No.	Transition Name	Cause	From State	To State	Effect
1	HaltedToReady	Reset Method	Halted	Ready	Report Transition 1 Event/Result
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method or Internal (Error)	Running	Halted	Report Transition 3 Event/Result
4	RunningToReady	Internal	Running	Ready	Report Transition 4 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
8	SuspendedToReady	Internal	Suspended	Ready	Report Transition 8 Event/Result
9	ReadyToHalted	Halt Method	Ready	Halted	Report Transition 9 Event/Result

**Figure 3 - Program States and Transitions**

#### 4.2.3 Program States

A standard set of base states are defined for *Programs* as part of the *Program Finite State Machine*. These states represent the stages in which a *Program* can exist at an instance in time as viewed by a client. This state is the *Program's Current State*. All *Programs* must support this base set. A *Program* may or may not require a client action to cause the state to change.

**Table 2 - Program States**

State	Description
Ready	The <i>Program</i> is properly initialized and may be started.
Running	The <i>Program</i> is executing making progress towards completion.
Suspended	The <i>Program</i> has been stopped prior to reaching a terminal state but may be resumed.
Halted	The <i>Program</i> is in a terminal or failed state, and it cannot be started or resumed without being reset.

The set of states defined to describe a *Program* can be expanded. Program sub states can be defined for the base states to provide more resolution to the process and to describe the cause and effects of additional stimuli and transitions. Standards bodies and industry groups may extend the base *Program Finite State Model* to conform to industry models. For example, the Halted state can include the sub states “Aborted” and “Completed” to indicate if the function achieved a successful conclusion prior to the transition to Halted. Transitional states such as “Starting” or “Suspending” might also be extensions of the running state, for example.

#### 4.2.4 State Transitions

A standard set of state transitions is defined for the *Program Finite State Machine*. These transitions define the valid changes to the *Program*’s current state in terms of an initial state and a resultant state.

**Table 3 - Program State Transitions**

Transition No.	Transition Name	Initial State	Resultant State
1	HaltedToReady	Halted	Ready
2	ReadyToRunning	Ready	Running
3	RunningToHalted	Running	Halted
4	RunningToReady	Running	Ready
5	RunningToSuspended	Running	Suspended
6	SuspendedToRunning	Suspended	Running
7	SuspendedToHalted	Suspended	Halted
8	SuspendedToReady	Suspended	Ready
9	ReadyToHalted	Ready	Halted

#### 4.2.5 Program State Transition Stimuli

The stimuli or causes for a *Program*’s state transitions can be internal to the Server or external. Completion of machining steps, the detection of an alarm condition, or the transmission of a data packet are examples of internal stimuli. *Methods* are an example of external stimuli. Standard *Methods* are defined which act as stimuli for the control of a *Program*.

#### 4.2.6 Program Control Methods

*Clients* manage a *Program* by calling *Methods*. The *Methods* impact a *Program*’s behaviour by causing specified state transitions. The state transitions dictate the action’s performed by the *Program*. This specification defines a set of standard *Program Control Methods*. These *Methods* provide sufficient means for a client to run a *Program*.

Table 4 lists the set of defined *Program Control Methods*. Each *Method* causes transitions from specified states and must be called when the *Program* is in one of those states.

Individual *Programs* can optionally support any subset of the *Program Control Methods*. For example, some *Programs* may not be permitted to suspend and so would not provide the *Suspend* and *Resume Methods*.

*Programs* can support additional user defined *Methods*. User defined *Methods* must not change the behaviour of the base *Program Finite State Machine*.

**Table 4 - Program Control Methods**

Method Name	Description
Start	Causes the Program to transition from the Ready state to the Running state.
Suspend	Causes the Program to transition from the Running state to the Suspended state.
Resume	Causes the Program to transition from the Suspended state to the Running state.
Halt	Causes the Program to transition from the Ready, Running or Suspended state to the Halted state.
Reset	Causes the Program to transition from the Halted state to the Ready state.

*Program Control Methods* can include arguments that are used by the *Program*. For example, a Start method may include an options argument that specifies dynamic options used to determine some program behaviour. The arguments can differ on each *ProgramType*. The Method Call service specified in [UA Part 4] defines a return status. This return status indicates the success of the *Program Control Method* or a reason for its failure.

#### 4.2.7 Program State Transition Effects

A *Program*'s state transition generally has a cause and also yields an effect. The effect is a by product of a *Program* state transition that can be used by a *Client* to monitor the progress of the *Program*. Effects can be internal or external. An external effect of a state transition is the generation of an event notification. Each *Program* state transition is associated with a unique event. These events reflect the progression and trajectory of the *Program* through its set of defined states. The internal effects of a state transition can be the performance of some programmatic action such as the generation of data.

#### 4.2.8 Program Result Data

##### 4.2.8.1 Overview

Result data is generated by a running *Program*. The result data can be intermediate or final. Result data may be associated with specific *Program* state transitions.

##### 4.2.8.2 Intermediate Result Data

Intermediate result data is transient and is generated by the *Program* in conjunction with non terminal state transitions. The data items that compose the intermediate results are defined in association with specific *Program* state transitions. Their values are relevant only at the transition.

Each *Program* state transition can be associated with different result data items. Alternately, a set of transitions can share a result data item. Percentage complete is an example of intermediate result data. The value of percentage complete is produced when the state transition occurs and is available to the client.

Clients acquire intermediate result data by subscribing to *Program* state transition events. The events specify the data items for each transition. When the transition occurs, the generated event conveys the result data values captured to the subscribed clients. If no *Client* is monitoring the *Program*, intermediate result data may be discarded.

##### 4.2.8.3 Terminal Result Data

Terminal result data is the final data generated by the *Program* as it ceases execution. Total execution time, number of widgets produced, and fault condition encountered are examples of terminal result data. When the *Program* enters the terminal state, this result data can be conveyed to the client by the transition event. Terminal result data is also available within the *Program* to be read by a client after the program stops. This data persists until the program instance is rerun or deleted.

#### 4.2.8.4 Monitoring Programs

Clients can monitor the activities associated with a *Program*'s execution. These activities include the invocation of the management methods, the generation of result data, and the progression of the *Program* through its states. Audit Events are provided for *Method Calls* and state transitions. These events allow a record to be maintained of the clients that interacted with any *Program* and the *Program* state transitions that resulted from that interaction.

### 4.2.9 Program Lifetime

#### 4.2.9.1 Overview

*Programs* can have different lifetimes. Some programs may always be present on a *Server* while others are created and removed. Creation and removal can be controlled by a *Client* or may be restricted to local means.

A *Program* can be *Client* creatable. If a *Program* is client creatable, then the *Client* can add the *Program* to the server. The *Object Create Method* defined in [UA Part 3] is used to create the *Program Instance*. The initial state of the *Program* can be *Halted* or *Ready*. Some *Programs*, for example, may require that a resource becomes available after its creation, before it is ready to run. In this case, it would be initialized in the *Halted* state and transition to *Ready* when the resource is delivered.

A *Program* can be *Client* removable. If the *Program* is client removable, then the *Client* can delete the *Program Instance* from the *Server*. The *Object DeleteNode Service* defined in [UA Part 4] is used to remove the *Program Instance*. The *Program* must be in a *Halted* state to be removed. A *Program* may also be auto removable. An auto removable *Program* deletes itself when execution has terminated.

#### 4.2.9.2 Program Instances

*Programs* can be multiple instanced or single instanced. A *Server* can support multiple instances of a *Program* if these *Program* instances can be run in parallel. For example, the *Program* may define a *Start Method* that has an input argument to specify which resource is acted upon by its functions. Each instance of the *Program* is then started designating use of different resources. The *Client* can discover all instances of a *Program* that are running on a *Server*. Each instance of a *Program* is uniquely identified on the *Server* and is managed independently by the *Client*.

#### 4.2.9.3 Program Recycling

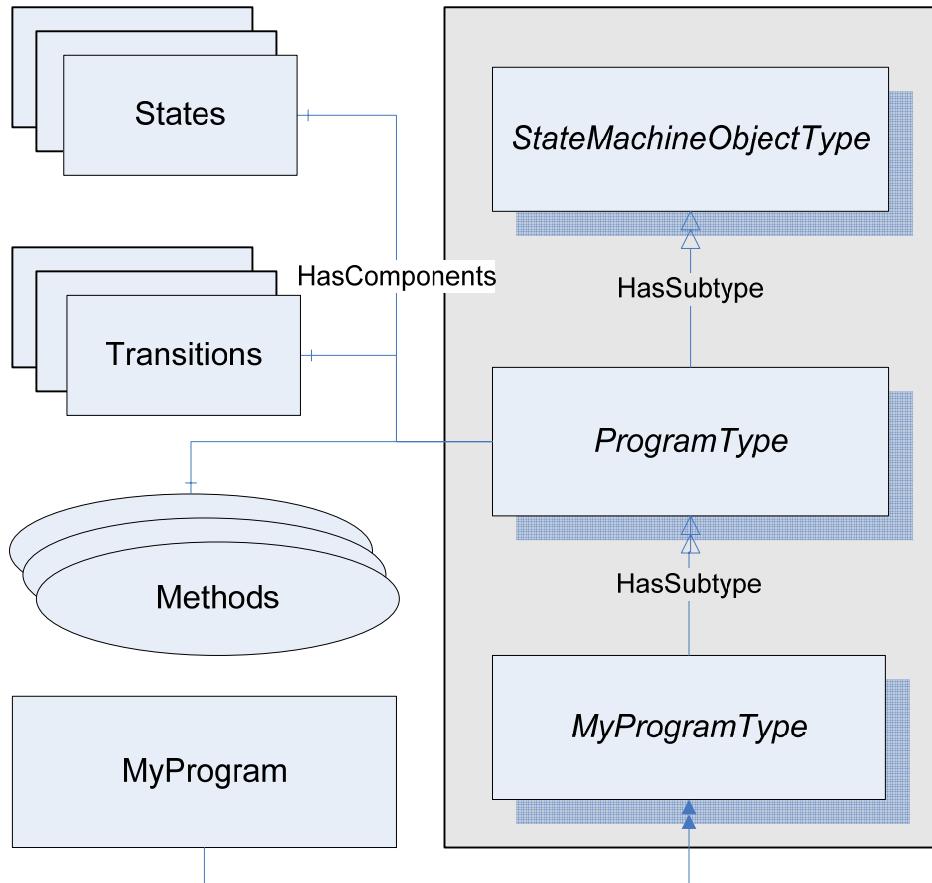
Programs can be run once or run multiple times (recycled). A program that is run once will remain in the *Halted state* indefinitely once it has run. The normal course of action would be to delete it following the inspection of its terminal results.

Recyclable *Programs* may have a limited or unlimited cycle count. These *Programs* may require a reset step to transition from the *Halted state* to the *Ready state*. This allows for replenishing resources or reinitializing parameters prior to restarting the *Program*. The *Program Control Method* "Reset" triggers this state transition and any associated actions or effects.

## 5 Model

### 5.1 General

The *Program Model* extends the *StateMachineType* and basic *ObjectType* Models presented in [UA Part 5]. Each *Program* has a type definition that is the subtype of the *StateMachineType*. The *ProgramType* describes the *Finite State Machine* model supported by any *Program Invocation* of that type. The *ProgramType* also defines the property set that characterize specific aspects of that *Program*'s behaviour such as lifetime and recycling as well as specifying the result data that is produced by the *Program*.



**Figure 4 - Program Type**

The base *ProgramType* defines the standard *Finite State Machine* specified for all *Programs*. This includes the states, transitions, transition causes (*Methods*) and effects (*Events*). Subtypes of the base *ProgramType* can be defined to extend or more specifically characterize the behaviour of an individual *Program* as illustrated with “*MyProgramType*” in Figure 4.

## 5.2 ProgramType

### 5.2.1 Overview

The additional properties and components that compose the *ProgramType* are listed in Table 5. No *ProgramType* specific semantics are assigned to the other base *ObjectType* or *StateMachineType* *Attributes* or *Properties*.

**Table 5 - ProgramType**

Attribute	Value				
	Includes all attributes specified for the StateMachineType				
BrowseName	ProgramType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
HasProperty	Variable	Creatable	Boolean	.PropertyType	None
HasProperty	Variable	Deletabe	Boolean	.PropertyType	New
HasProperty	Variable	AutoDelete	Boolean	.PropertyType	Shared
HasProperty	Variable	RecycleCount	Int32	.PropertyType	New
HasProperty	Variable	InstanceCount	UInt32	.PropertyType	None
HasProperty	Variable	MaxInstanceCount	UInt32	.PropertyType	None
HasProperty	Variable	MaxRecycleCount	UInt32	.PropertyType	None
HasComponent	Variable	ProgramDiagnostic		ProgramDiagnosticType	OptionalNew
InitialState	Object	(Halted or Ready)		StateType	
HasComponent	Object	Halted		StateType	New
HasComponent	Object	Ready		StateType	New
HasComponent	Object	Running		StateType	New
HasComponent	Object	Suspended		StateType	New
HasComponent	Object	HaltedToReady		TransitionType	New
HasComponent	Object	ReadyToRunning		TransitionType	New
HasComponent	Object	RunningToHalted		TransitionType	New
HasComponent	Object	RunningToReady		TransitionType	New
HasComponent	Object	RunningToSuspended		TransitionType	New
HasComponent	Object	SuspendedToRunning		TransitionType	New
HasComponent	Object	SuspendedToHalted		TransitionType	New
HasComponent	Object	SuspendedToReady		TransitionType	New
HasComponent	Object	ReadyToHalted		TransitionType	New
HasComponent	Method	Start			OptionalNew
HasComponent	Method	Suspend			OptionalNew
HasComponent	Method	Reset			OptionalNew
HasComponent	Method	Halt			OptionalNew
HasComponent	Method	Resume			OptionalNew
HasComponent	Object	FinalResultData		BaseObjectType	OptionalNew

### 5.2.2 ProgramType Properties

The *Creatable* Property is a Boolean that specifies if *Program Invocations* of this *ProgramType* can be created by an OPC Client. If False, these *Program Invocations* are persistent or may only be created by the server.

The *Deletable* Property is a Boolean that specifies if a *Program Invocation* of this *ProgramType* can be deleted by an OPC Client. If False, these *Program Invocations* can only be deleted by the server.

The *AutoDelete* Property is a Boolean that specifies if *Program Invocations* of this *ProgramType* are removed by the Server when execution terminates. If False, these *Program Invocations* persist on the server until they are deleted by the Client. When the Program Invocation is deleted, any result data associated with the instance is also removed.

The *RecycleCount* Property is an unsigned integer that specifies the number of times a *Program Invocation* of this type has been recycled or restarted from its starting point (not resumed). Note: The Reset Method may be required to prepare a Program to be restarted.

The *MaxRecycleCount Property* is an integer that specifies the maximum number of times a *Program Invocation* of this type can be recycled or restarted from its starting point (not resumed). If the value is less than 0, there is no limit to the number of restarts. If the value is zero, the Program may not be recycled or restarted.

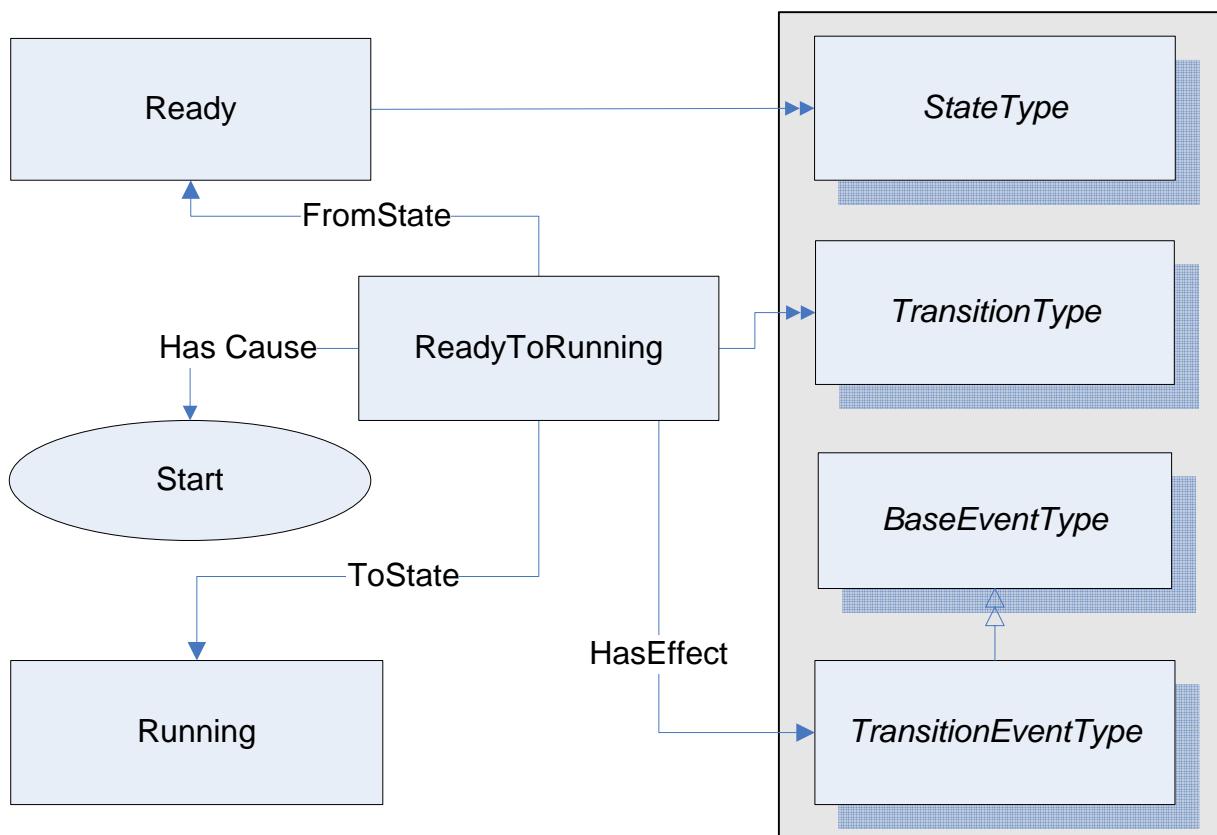
The *InstanceCount Property* is an unsigned integer that specifies the number of *Program Invocations* of this type that currently exist.

The *MaxInstanceCount Property* is an integer that specifies the maximum number of *Program Invocations* of this type that can exist simultaneously on this Server. If the value is less than 0, there is no limit.

### 5.2.3 ProgramType Components

#### 5.2.3.1 Overview

The *ProgramType Components* consists of a set of references to the object instances of *StateTypes*, *TransitionTypes*, *EventTypes* and the *Methods* that collectively define the *Program FiniteStateMachine*.



**Figure 5 - Program FSM References**

Figure 5 illustrates the *Component References* that define the associations between two of the *ProgramType's* states, *Ready* and *Running*. The complementary *ReferenceTypes* have been omitted to simplify the illustration.

#### 5.2.3.2 ProgramType States

Table 6 specifies the *ProgramType's* State Objects. These State Objects are instances of the *StateType* defined in [UA Part 5] - SM Appendix. Each State is assigned a unique *StateNumber* value. Subtypes of the *ProgramType* can add references from any state to a subordinate or nested *StateMachine Object* to extend the *FiniteStateMachine*.

**Table 6 - Program States**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>States</b>					
Halted	HasProperty	StateNumber	1	.PropertyType	
	ToTransition	HaltedToReady		TransitionType	
	FromTransition	RunningToHalted		TransitionType	
	FromTransition	SuspendedToHalted		TransitionType	
	FromTransition	ReadyToHalted		TransitionType	
Ready	HasProperty	StateNumber	2	.PropertyType	
	FromTransition	HaltedToReady		TransitionType	
	ToTransition	ReadyToRunning		TransitionType	
	FromTransition	RunningToReady		TransitionType	
	ToTransition	ReadyToHalted		TransitionType	
Running	HasProperty	StateNumber	3	.PropertyType	
	ToTransition	RunningToHalted		TransitionType	
	ToTransition	RunningToReady		TransitionType	
	ToTransition	RunningToSuspended		TransitionType	
	FromTransition	ReadyToRunning		TransitionType	
	FromTransition	SuspendedToRunning		TransitionType	
Suspended	HasProperty	StateNumber	4	.PropertyType	
	ToTransition	SuspendedToRunning		TransitionType	
	ToTransition	SuspendedToHalted		TransitionType	
	ToTransition	SuspendedToReady		TransitionType	
	FromTransition	RunningToSuspended		TransitionType	

The *Halted* state is the idle state for a *Program*. It can be an initial state or a terminal state. As an initial state, the Program Invocation can not yet begin execution due to conditions at the server. As a terminal state, *Halted* can indicate either a failed or completed *Program*. A subordinate state or result can be used to distinguish the nature of the termination. The *Halted* state references four *Transition Objects*, which identify the allowed state transitions to the *Ready* state and from the *Ready*, *Running*, and *Suspended* States.

The *Ready* state indicates that the *Program* is prepared begin execution. *Programs* that are ready to begin upon their creation may transition immediately to the *Ready* State. The Ready state references four *Transition Objects*, which identify the allowed state transitions to the *Running* and *Halted* states and from the *Halted* and *Ready* states.

The *Running* state indicates that the *Program* is actively performing its function. The Ready state references five *Transition Objects*, which identify the allowed state transitions to the *Halted*, *Ready*, and *Suspended* states and from the *Ready* and *Suspended* States.

The *Suspended* state indicates that the *Program* has stopped performing its function, but retains the ability to resume the function at the point at which it was executing when suspended. The *Suspended* state references four *Transition Objects*, which identify the allowed state transitions to the *Ready* state and from the *Ready*, *Running*, and *Suspended* States.

### 5.2.3.2.1 Program Initial State

The initial state of a *ProgramType* is the state that the Program Invocation assumes upon creation. This *State* is formally specified in the *Program Finite State Machine* by defining an *InitialState Reference* in the *ProgramType*. The *InitialState Reference Type* is defined in [UA Part 5] - SM Appendix. If specified, the target of the *InitialState Reference* must be either the *Halted State* or the *Ready State*. In some *Programs*, the initial state can vary and would not be referenced. If these *Halted* or *Ready States* contain subordinate *Finite State Machines*, the *FSM* can include an *InitialState* reference.

### 5.2.3.3 ProgramType Transitions

*ProgramType* Transitions are instances of *Objects* of the *TransitionType* defined in [UA Part 5] - SM Appendix which also includes the definitions of the *ToState*, *FromState*, *HasCause*, and *HasEffect* references used. Table 7 specifies the Transitions defined for the *ProgramType*. Each Transition is assigned a unique *TransitionNumber*.

**Table 7 - Program Transitions**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>Transitions</b>					
HaltedToReady	HasProperty	TransitionNumber	1	.PropertyType	
	ToState	Ready		.StateType	
	FromState	Halted		.StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
ReadyToRunning	HasProperty	TransitionNumber	2	.PropertyType	
	ToState	Running		.StateType	
	FromState	Ready		.StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
RunningToHalted	HasProperty	TransitionNumber	3	.PropertyType	
	ToState	Halted		.StateType	
	FromState	Running		.StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
RunningToReady	HasProperty	TransitionNumber	4	.PropertyType	
	ToState	Ready		.StateType	
	FromState	Running		.StateType	
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
RunningToSuspended	HasProperty	TransitionNumber	5	.PropertyType	
	ToState	Running		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
SuspendedToRunning	HasProperty	TransitionNumber	6	PropertyParams	
	ToState	Running		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
SuspendedToHalted	HasProperty	TransitionNumber	7	PropertyParams	
	ToState	Halted		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
SuspendedToReady	HasProperty	TransitionNumber	8	PropertyParams	
	ToState	Ready		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
ReadyToHalted	HasProperty	TransitionNumber	9	PropertyParams	
	ToState	Halted		.StateType	
	FromState	Ready		.StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional

The *HaltedToReady* transition specifies the Transition from the *Halted* to *Ready* States. It may be caused by the *Reset Method*.

The *ReadyToRunning* transition specifies the Transition from the *Ready* to *Running* States. It is caused by the *Start Method*.

The *RunningToHalted* transition specifies the Transition from the *Running* to *Halted* States. It is caused by the *Halt Method*.

The *RunningToReady* transition specifies the Transition from the *Running* to *Ready* States. The *RunningToSuspended* transition specifies the Transition from the *Running* to *Suspended* States. It is caused by the *Suspend Method*. When this transition occurs,

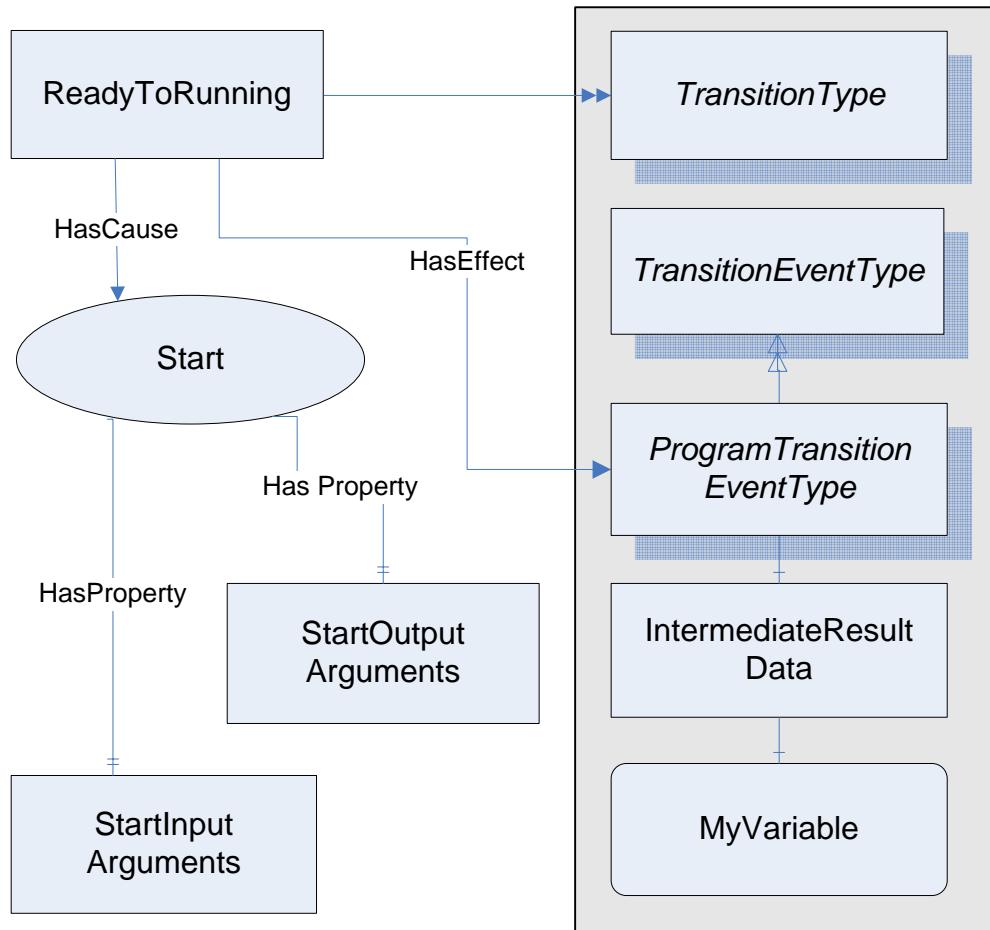
The *SuspendedToRunning* transition specifies the Transition from the *Suspended* to *Running* States. It is caused by the *Resume Method*.

The *SuspendedToHalted* transition specifies the Transition from the *Suspended* to *Halted* States. It is caused by the *Halt Method*.

The *SuspendedToReady* transition specifies the Transition from the *Suspended* to *Ready* States. It is caused by the *Halt Method*.

The *ReadyToHalted* transition specifies the Transition from the *Ready* to *Halted* States. It is caused by the *Halt Method*.

Two *HasEffect* references are specified for each Program Transition. These effects are events of type *ProgramTransitionEventType* and *ProgramTransitionAuditEventType* defined in xxx. The *ProgramTransitionEventType* notifies *Clients* of the Program Transition and conveys result data. The *ProgramTransitionAuditEventType* can optionally be used to audit transitions that result from *Program Control Methods*.



**Figure 6 - ProgramType Causes and Effects**

#### 5.2.4 ProgramType Causes (Methods)

##### 5.2.4.1 Overview

The *ProgramType* includes references to the *Causes* of specific *Program* state transitions. These causes refer to *Method* instances. *Programs* that do not support a *Program Control Method*, omit the *Causes* reference to that *Method* from the *ProgramType* references. If a *Method*'s *Causes* reference is omitted from the *ProgramType*, a *Client* cannot cause the associated state transition. The *Method* instances referenced by the *ProgramType* identify the *InputArguments* and *OutputArguments* required for the *Method* calls to *Program Invocations* of that *ProgramType*. Table 8 - *ProgramType Causes* specifies the *Methods* defined as *Causes* for *ProgramTypes*. Figure 6 - *ProgramType Causes and Effects* illustrates the references associating the components and properties of *Methods* and *Events* with *ProgramTransitions*.

**Table 8 - ProgramType Causes**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>Causes</b>					
Start	HasProperty	InputArguments		.PropertyType	Optional
	HasProperty	OutputArguments		.PropertyType	Optional
Suspend	HasProperty	InputArguments		.PropertyType	Optional
	HasProperty	OutputArguments		.PropertyType	Optional
Resume	HasProperty	InputArguments		.PropertyType	Optional
	HasProperty	OutputArguments		.PropertyType	Optional
Halt	HasProperty	InputArguments		.PropertyType	Optional
	HasProperty	OutputArguments		.PropertyType	Optional
Reset	HasProperty	InputArguments		.PropertyType	Optional
	HasProperty	OutputArguments		.PropertyType	Optional

The *Start Method* causes the *ReadyToRunning Program* transition.

The *Suspend Method* causes the *RunningToSuspended Program* transition.

The *Resume Method* causes the *SuspendedToRunning Program* transition.

The *Halt Method* causes the *RunningToHalted*, *SuspendedToHalted*, or *ReadyToHalted Program* transition depending on the *CurrentState* of the *Program*.

The *Reset Method* causes the *HaltedToReady Program* transition.

#### 5.2.4.2 Standard Attributes

The *Executable Method* attribute indicates if a method can currently be executed. For *Program Control Methods*, this means that the owning *Program* has a *CurrentState* that supports the transition caused by the *Method*.

#### 5.2.4.3 Standard Properties

##### 5.2.4.3.1 Overview

*Methods* can reference a set of *InputArguments*. For each *ProgramType*, a set of *InputArguments* may be defined for the supported *Program Control Methods*. The data passed in the arguments supplements the information required by the *Program* to perform its function. All calls to a *Program Control Method* for each *Program Invocation* of that *ProgramType* must pass the specified arguments.

*Methods* can reference a set of *OutputArguments*. For each *ProgramType*, a set of *OutputArguments* is defined for the supported *Program Control Methods*. All calls to a *Program Control Method* for each *Program Invocation* of that *ProgramType* must pass the specified arguments.

#### 5.2.5 ProgramType Effects (Events)

##### 5.2.5.1 Overview

The *ProgramType* includes component references to the *Effects* of each of the *Program's state transitions*. These *Effects* are *Events*. Each *Transition* must have a *HasEffect* reference to a

*ProgramTransitionEventType* and can have a *ProgramTransitionAuditEventType*. When the transition occurs, event notifications of the referenced type are generated for subscribed *Clients*. The Program Invocation may serve as the *EventNotifier* for these events or an owning object or the Server Object may provide the notifications.

*ProgramTransitionEventTypes* provide the means for delivering result data and confirming state transitions for subscribed *Clients* on each defined *Program State Transition*. *ProgramTransitionAuditEventTypes* allows the auditing of changes to the *Program's State* in conjunction with auditing client *Method Calls*.

### 5.2.5.2 ProgramTransitionEventType

The *ProgramTransitionEventType* is a subtype of the *TransitionEventType*. It is used with Programs to acquire intermediate or final results or other data associated with a state transition. A Program can have a unique *ProgramTransitionEventType* definition for any transition. Each *ProgramTransitionEventType* specifies the *IntermediateResult* data specific to the designated state transition on that program type. Each transition can yield different Intermediate result data. Table 9 specifies the *ProgramTransitionEventType*. Table 10 identifies the *ProgramTransitionEventTypes* that are specified for *ProgramTypes*.

**Table 9 - ProgramTransitionEventType**

Attribute	Value				
BrowseName	ProgramTransitionEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherits the <i>Properties</i> of the base <i>TransitionEventType</i> defined in [UA- Part 5] SM appendix.					
HasComponent	Object	IntermediateResult		BaseObjectType	OptionalNew

The *TransitionNumber* property is a *Variable* that identifies the *Program Transition* that triggered the event.

The *FromStateNumber* property is a *Variable* that is the *StateNumber* of the originating state of the *Program Transition*.

The *ToStateNumber* property is a *Variable* that is the *StateNumber* of the terminal state in The *Program Transition*.

The *IntermediateResult* is an object that aggregates a set of variables whose values are relevant for the Program at the instant of the associated transition. The *ObjectType* for the *IntermediateResult* specifies the collection of variables using a set of *HasComponent* references.

**Table 10 – ProgramTransitionEvents**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>Effects</b>					
HaltedToReadyEvent					
	HasProperty	TransitionNumber	1	.PropertyType	
	HasProperty	FromStateNumber	1	.PropertyType	
	HasProperty	ToStateNumber	2	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToRunningEvent					
	HasProperty	TransitionNumber	2	.PropertyType	
	HasProperty	FromStateNumber	2	.PropertyType	
	HasProperty	ToStateNumber	3	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToHaltedEvent					
	HasProperty	TransitionNumber	3	.PropertyType	
	HasProperty	FromStateNumber	3	.PropertyType	
	HasProperty	ToStateNumber	1	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToReadyEvent					
	HasProperty	TransitionNumber	4	.PropertyType	
	HasProperty	FromStateNumber	3	.PropertyType	
	HasProperty	ToStateNumber	2	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToSuspendedEvent					
	HasProperty	TransitionNumber	5	.PropertyType	
	HasProperty	FromStateNumber	3	.PropertyType	
	HasProperty	ToStateNumber	4	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToRunningEvent					
	HasProperty	TransitionNumber	6	.PropertyType	
	HasProperty	FromStateNumber	4	.PropertyType	
	HasProperty	ToStateNumber	3	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToHaltedEvent					
	HasProperty	TransitionNumber	7	.PropertyType	
	HasProperty	FromStateNumber	4	.PropertyType	
	HasProperty	ToStateNumber	1	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToReadyEvent					
	HasProperty	TransitionNumber	8	.PropertyType	
	HasProperty	FromStateNumber	4	.PropertyType	
	HasProperty	ToStateNumber	2	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToHaltedEvent					
	HasProperty	TransitionNumber	9	.PropertyType	
	HasProperty	FromStateNumber	2	.PropertyType	
	HasProperty	ToStateNumber	1	.PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional

### 5.2.6 ProgramTransitionAuditEventType

The *ProgramTransitionAuditEventType* is a subtype of the *AuditEventType*. This *EventType* inherits all *Properties* of the *AuditEventType* defined in [UA- Part 5]. It is used with *Programs* to provide a means to audit the *Program State Transitions* associated with any *Client* invoked *Program Control Method*. Table 11 specifies the definition of the *ProgramTransitionAuditEventType*.

**Table 11 - ProgramTransitionAuditEvent**

Attribute	Value				
BrowseName	ProgramTransitionAuditEventType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Inherits the <i>Properties</i> of the <i>AuditEventType</i> defined in [UA- Part 5]					
HasProperty	Variable	TransitionNumber	UInt32	.PropertyType	New

The *Status Property*, specified in [UA- Part 5] *AuditEventType*, identifies whether the state transition resulted from a Program Control Method call (set *Status* to TRUE) or not (set *Status* to FALSE).

The *ClientUserId Property*, specified in [UA- Part 5] *AuditEventType*, identifies the user of the *Client* that issued the *Program Control Method* if it is associated with this *Program State Transition*.

The *ActionTimeStamp Property*, specified in [UA- Part 5] *AuditEventType*, identifies when the time the *Program State Transition* that resulted in the event being generated occurred.

The *TransitionNumber* property is a *Variable* that identifies the Transition that triggered the event.

### 5.2.7 FinalResultData

The *FinalResultData ObjectType* specifies the *VariableTypes* that are preserved when the *Program* has completed its function. The *ObjectType* includes a *HasComponent* for a *VariableType* of each variable that comprises the *FinalResultData*. The values of the Variables

### 5.2.8 ProgramDiagnosticType

#### 5.2.8.1 Overview

The *ProgramDiagnosticType* provides information that can be used to aid in the diagnosis of *Program* problems. This object contains a collection of *Variables* that chronicle the *ProgramInvocation's* activity. Table 12 specifies the *Variables* that compose the *ProgramDiagnosticType*.

**Table 12 - ProgramDiagnosticType**

Attribute	Value				
BrowseName	ProgramDiagnosticsType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType / TypeDefinition	Modelling Rule	
Subtype of the <i>BaseObjectType</i> defined in [UA Part 5].					
HasComponent	Variable	CreateSessionId	Int32	New	
HasComponent	Variable	CreateClientName	String	New	
HasComponent	Variable	InvocationCreationTime	UTCTime	New	
HasComponent	Variable	LastTransitionTime	UTCTime	New	
HasComponent	Variable	LastMethodCall	String	New	
HasComponent	Variable	LastMethodSessionId	Int32	New	
HasComponent	Variable	LastMethodInputArguments	InputArguments	New	
HasComponent	Variable	LastMethodOutputArguments	OutputArguments	New	
HasComponent	Variable	LastMethodCallTime	UTCTime	New	
HasComponent	Variable	LastMethodReturnStatus	returnStatus	New	

The *CreateSessionId* contains the *SessionId* of the session on which the call to the *Create Service* was issued to create the Program Invocation.

The *CreateClientName* is the name of the client of the session that created the *Program Invocation*.

The *InvocationCreationTime* identifies the time the *Program Invocation* was created.

The *LastTransitionTime* identifies the time of the last program state transition that occurred.

The *LastMethodCall* identifies the last *Program Method* called on the *Program Invocation*.

The *LastMethodSessionId* contains the *SessionId* of the session on which the last Program Control Method call to the *Program Invocation* was issued.

The *LastMethodClientName* is the name of the client of the session that made the last Method call to the *Program Invocation*.

The *LastMethodInputArguments* preserves the values of the input arguments on the last *Program Method* call.

The *LastMethodOutputArguments* preserves the values of the output arguments on the last *Program Method* call.

The *LastMethodCallTime* identifies the time of the last Method call to the *Program Invocation*.

The *LastMethodReturnStatus* preserves the value of the *returnStatus* for the last Program Control Method requested for this *Program Invocation*.

## Annex A - Program Example

### A.1 Overview

This example illustrates the use of a UA *Program* to manage a domain download into a control system as depicted in Figure 7. The download requires the segmented transfer of control operation data from a secondary storage device to the local memory within a control system.

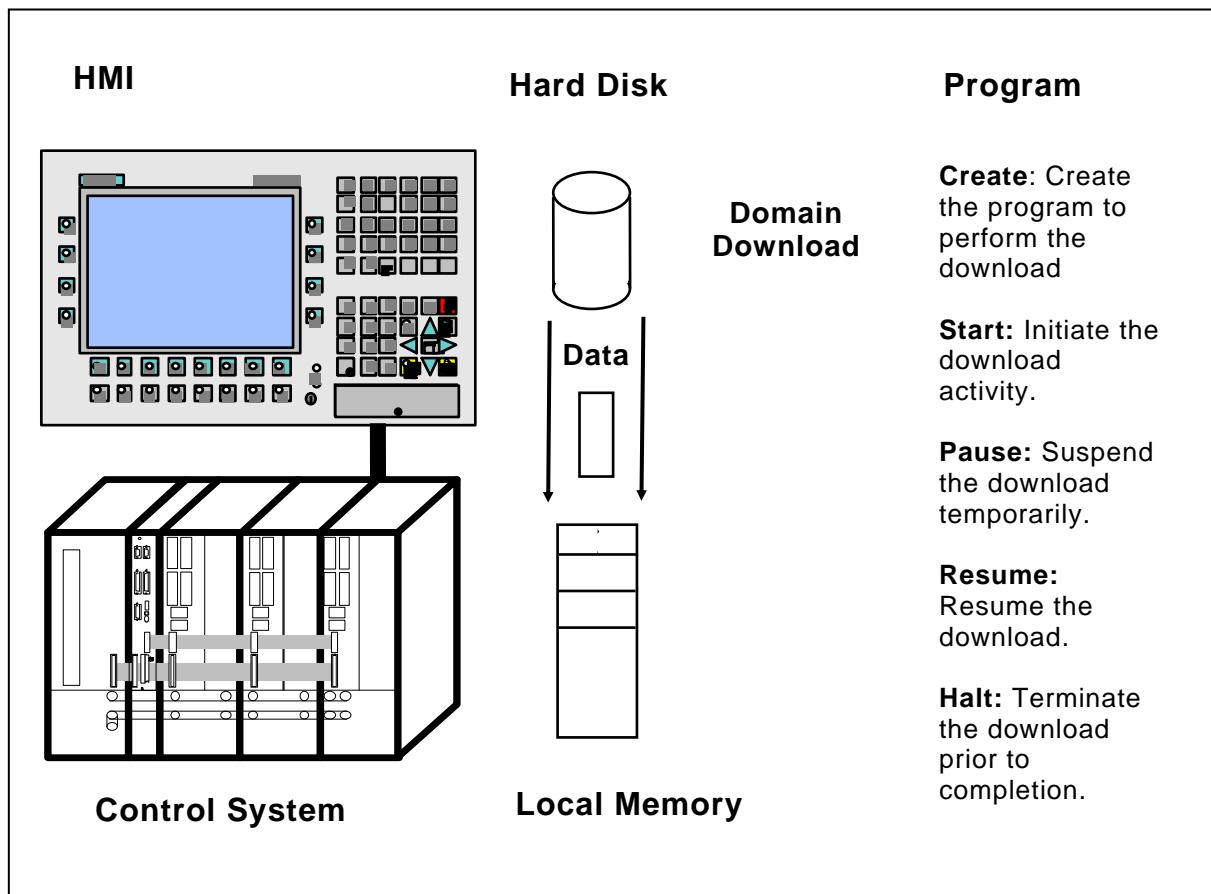


Figure 7 - Program Example 1

The Domain Download has a source and a target location which are identified when the download is initiated. Each time a segment of the domain is successfully transferred, the client is notified and informed of the amount of data that has been downloaded. The client is also notified when the download is finished. The percentage of the total data received is reported periodically while the download continues. If the download fails, the cause of the failure is reported. At the completion of the download, performance information is persisted at the UA Server.

### A.2 DomainDownload Program

The *UA Client* uses the “**DomainDownload**” *Program* to manage and monitor the download of a domain at the *UA Server*.

#### A.2.1 DomainDownload States

The basic state model for the *DomainDownload Program* is presented in Figure 8. The *Program* has three primary states, *Ready*, *Running*, and *Halted* which are aligned with the standard states of a *ProgramType*. Additionally, the *DomainDownloadType* extends the *UA ProgramType* by defining subordinate state machines for the *Program*’s *Running* and *Halted* States. The subordinate states

describe the download operations in greater detail and allow the UA *Client* to monitor the activity of the download at a finer resolution.

An instance (Program Invocation) of a DownloadDomain *Program* is created by the client each time a download is to be performed. The instance exists until explicitly removed by the client. The initial state of the *Program* is *Ready* and the terminal state is *Halted*. The DomainDownload can be temporarily suspended and then resumed or aborted. Once halted, the program may not be restarted.

**Error! Objects cannot be created from editing field codes.**

**Figure 8 – DomainDownload State Diagram**

**Figure 8** illustrates the sequence of state transitions. Once the download is started, The Program progresses to the Opening state. After the source of the data is opened, a sequence of transfers occurs in the Sending state. When the transfer completes the objects are closed in the Closing State. If the transfer is terminated before all of the data is downloaded or an error is encountered, the download is halted, and the Program transitions to the Aborted state, otherwise the programs halts in the Completed state. The states are presented in Table 13 along with the state transitions.

### A.2.2 DomainDownload Transitions

The valid state transitions specified for the DomainDownload *Program* and are specified in Table 13. Each of the transitions defines a start state and end state for the transition and is identified by a unique number. Five of the transitions are from the base *ProgramType* and retain the transition identifier numbers specified for *Programs*. The additional transitions relate the base program states with the subordinate states defined for the DomainDownload. These states have been assigned unique transition identifier numbers distinct from the base Program transition identifiers. In cases where transitions occur between substates and the Program's base states, two transitions are specified. One transition identifies the base state change and a second the sub-state change. For example, Ready to Running and to Opening occurs at the same time.

The table also specifies the defined states, causes for the transitions, and the effects of each transition. Program Control Methods are used by the UA *Client* to "run" the DomainDownload. The *Methods* cause or trigger the specified transitions. The transition effects are the specified EventTypes which notify the client of *Program* activity.

**Table 13 - DomainDownload States**

No.	Transition Name	Cause	From State	To State	Effect
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method/Error or Internal.	Running	Halted	Report Transition 3 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
10	OpeningToSend	Internal	Opening	Sending	Report Transition 10 Event/Result
11	SendingToSend	Internal	Sending	Sending	Report Transition 11 Event/Result
12	SendingToClosing	Internal	Sending	Closing	Report Transition 12 Event/Result
13	SendingToAborted	Halt Method/Error	Opening	Aborted	Report Transition 13 Event/Result
14	ClosingToCompleted	Internal	Closing	Completed	Report Transition 14 Event/Result
15	SendingToSuspended	Suspend Method	Sending	Suspended	Report Transition 16 Event/Result
16	SuspendedToSend	Resume Method	Suspended	Sending	Report Transition 17 Event/Result
18	SuspendedToAborted	Halt Method	Suspended	Aborted	Report Transition 18 Event/Result
17	ToOpening	Internal	Ready	Opening	Report Transition 19 Event/Result

### A.2.3 DomainDownload Methods

Four standard Program *Methods* are specified for running the DomainDownload *Program*, *Start*, *Suspend*, *Resume*, and *Halt*. No additional *Methods* are specified. The base behaviours of these methods are defined by the *ProgramType*. The *Start Method* initiates the download activity and

passes the source and destination locations for the transfer. The *Suspend Method* is used to pause the activity temporarily. The *Resume Method* reinitiates the download, when paused. The *Halt Method* aborts the download. Each of the methods causes a *Program* state transition and a sub state transition. The specific state transition depends on the current state at the time the *Method* is called. If a *Method Call* is made when the *DomainDownload* is in a state for which that *Method* has no associated transition, the *Method* returns an error status indicating invalid state for the *method*.

### A.2.3.1 Method Arguments

The *Start Method* specifies three input arguments to be passed when it is called; *Domain Name*, *DomainSource*, and *DomainDestination*. The other *Methods* require no input arguments. No output arguments are specified for the *DomainDownload* methods. The resultant error status for the *Program* is part of the *Call* service.

## A.2.4 DomainDownload Events

A *ProgramTransitionEventType* is specified for each of the *DomainDownload Program* transitions. The event types trigger a specific event notification to the *UA Client* when the associated state transition occurs in the running *Program* instance. The event notification identifies the transition. The *SendingToSending* state transition also includes intermediate result data.

### A.2.4.1 Event Information

The *SendingToSending* program transition event relays intermediate result data to the *UA Client* along with the notification. Each time the transition occurs, data items describing the amount and percentage of data transferred is sent to the *UA Client*.

### A.2.4.2 Final Result Data

The *DomainDownload Program* retains final result data following a completed or aborted download. The data includes the total transaction time and the size of the domain. In the event of an aborted download, the reason for the termination is retained.

## A.2.5 DomainDownload Model

### A.2.5.1 Overview

The UA Model for the *DomainDownload.Program* is presented in the following tables and figures. Collectively they define the components that constitute this program. For clarity, the figures present a progression of portions of the model that complement the contents of the tables and illustrate the *Program*'s composition.

The type definition for the *DomainDownload Program* precisely represents the behaviour of the program in terms of *UA* components. These components can be browsed by a *UA Client* to interpret or validate the actions of the *Program*.

### A.2.5.2 DomainDownloadType

The *DomainDownloadType* is a subtype derived from the *UA ProgramType*. It specifies the use or non-use of optional *ProgramType* components, valid extensions such as subordinate state machines, and constrained attribute values applied to instances of *DomainDownload Programs*.

Table 14 specifies the optional and extended components defined by the *DomainDownload Type*. Note the references to two sub State Machine Types, *TransferStateMachine* and *FinishStateMachine*. The *DomainDownloadType* omits references to the *Reset Program Control Method* and its associated state transition (*HaltedToReady*), which it does not support.

Figure 9 illustrates the components of *DomainDownloadType* and its instance, including their ownership and derivation. Transitions and *EventTypes* are represented by their numeric Identifiers

in the figure. The references that exist between *StatesTypes*, *TransitionTypes*, *EventNotificationTypes*, and *Methods* are not shown in Figure 9.

**Table 14 – DomainDownload Type**

Attribute	Value				
	Includes all non-optional attributes specified for the ProgramType				
BrowseName	DomainDownloadType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
HasSubStateMachine	Object	TransferStateMachine		StateMachineType	New
HasSubStateMachine	Object	FinishStateMachine		StateMachineType	New
HasComponent	Variable	ProgramDiagnostic		ProgramDiagnosticType	New
InitialState	Object	Ready		StateType	
HasComponent	Object	ReadyToRunning		TransitionType	New
HasComponent	Object	RunningToHalted		TransitionType	New
HasComponent	Object	RunningToSuspended		TransitionType	New
HasComponent	Object	SuspendedToRunning		TransitionType	New
HasComponent	Object	SuspendedToHalted		TransitionType	New
HasComponent	Method	Start			New
HasComponent	Method	Suspend			New
HasComponent	Method	Halt			New
HasComponent	Method	Resume			New
HasComponent	Object	FinalResultData		BaseObjectType	New

Table 15 Specifies the Transfer State Machine type that is a sub state machine of the DomianDownload Program Type. This State Machine Type definition identifies the State types that compose the sub states for the Program's Running State type.

**Table 15 - Transfer State Machine Type**

Attribute	Value				
	Includes all attributes specified for the StateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
InitialState	Object	Opening		StateType	
HasComponent	Object	Opening		StateType	New
HasComponent	Object	Sending		StateType	New
HasComponent	Object	Closing		StateType	New
HasComponent	Object	ReadyToOpening		TransitionType	New
HasComponent	Object	OpeningToSending		TransitionType	New
HasComponent	Object	SendingToClosing		TransitionType	New
HasComponent	Object	SendingToAborted		TransitionType	New
HasComponent	Object	SendingToSuspended		TransitionType	New
HasComponent	Object	SuspendedToSending		TransitionType	New
HasComponent	Method	Start			New
HasComponent	Method	Suspend			New
HasComponent	Method	Halt			New
HasComponent	Method	Resume			New

Figure 15 specifies the *StateTypes* associated with the Transfer State Machine Type. All of these states are sub states of the *Running* state of the base *ProgramType*.

The Opening State is the preparation state for the domain download.

The Sending State is the activity state for the transfer in which the data is moved from the source to destination.

The Closing State is the cleanup phase of the download.

**Table 16 - Transfer State Machine - States**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>States</b>					
Opening	HasProperty	StateNumber	5	.PropertyType	
	ToTransition	OpeningToSending		TransitionType	
	FromTransition	ToOpening		TransitionType	
	ToTransition	OpeningToSending		TransitionType	
Sending	HasProperty	StateNumber	6	PropertyParams	
	FromTransition	OpeningToSending		TransitionType	
	ToTransition	SendingToSending		TransitionType	
	ToTransition	SendingToClosing		TransitionType	
	ToTransition	SendingToSuspended		TransitionType	
	FromTransition	ToSending		TransitionType	
Closing	HasProperty	StateNumber	7	PropertyParams	
	ToTransition	ClosingToCompleted		TransitionType	
	ToTransition	ClosingToAborted		TransitionType	
	FromTransition	SendingToClosing		TransitionType	

Table 17 specifies the Finish State Machine type that is a sub state machine of the DomianDownload *ProgramType*. This State Machine Type definition identifies the State types that compose the sub states for the Program's Halted State type.

**Table 17 - Finish State Machine Type**

Attribute	Value				
	Includes all attributes specified for the StateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
HasComponent	Object	Completed		StateType	New
HasComponent	Object	Aborted		StateType	New

Table 18 Specifies the State Types associated with the Finish State Machine Type. Note these are final states and that they have no associated transitions between them.

**Table 18 - Finish State Machine - States**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>States</b>					
Aborted	HasProperty	StateNumber	8	.PropertyType	
	FromTransition	OpeningToAborted		TransitionType	
	FromTransition	ClosingToAborted		TransitionType	
Completed	HasProperty	StateNumber	9	PropertyParams	
	FromTransition	ClosingToCompleted		TransitionType	

The Aborted State is the terminal state that indicates an incomplete or failed domain download operation.

The Completed State is the terminal state that indicates a successful domain download.

Table 19 specifies constraining behaviour of a DomainDownload.

**Table 19 – DomainDownload Type Property Attributes Variable Values**

NodeClass	BrowseName	Data Type	Data Value	Modelling Rule
Variable	Creatable	Boolean	True	None
Variable	Deletable	Boolean	True	New
Variable	AutoDelete	Boolean	False	Shared
Variable	RecycleCount	Int32	0	New
Variable	InstanceCount	UInt32	PropertyType	None
Variable	MaxInstanceCount	UInt32	500	None
Variable	MaxRecycleCount	UInt32	0	None

A DomainDownload *Program Invocation* can be created and also destroyed by a *UA Client*. The *Program Invocation* will not delete itself when halted, but will persist until explicitly removed by the *UA Client*. A DomainDownload *Program Invocation* can not be reset to restart. The *UA Server* will support up to 500 concurrent DomainDownload *Program Invocations*.

Figure 9 presents a partial DomainDownloadType model that illustrates the association between the states and the DomainDownload, Transfer, and Finish state machines. Note that the current state number for the sub state machines is only valid when the DomainDownload active base state references the sub state machine, Running for the Transfer current state and Halted for the Finish current state.

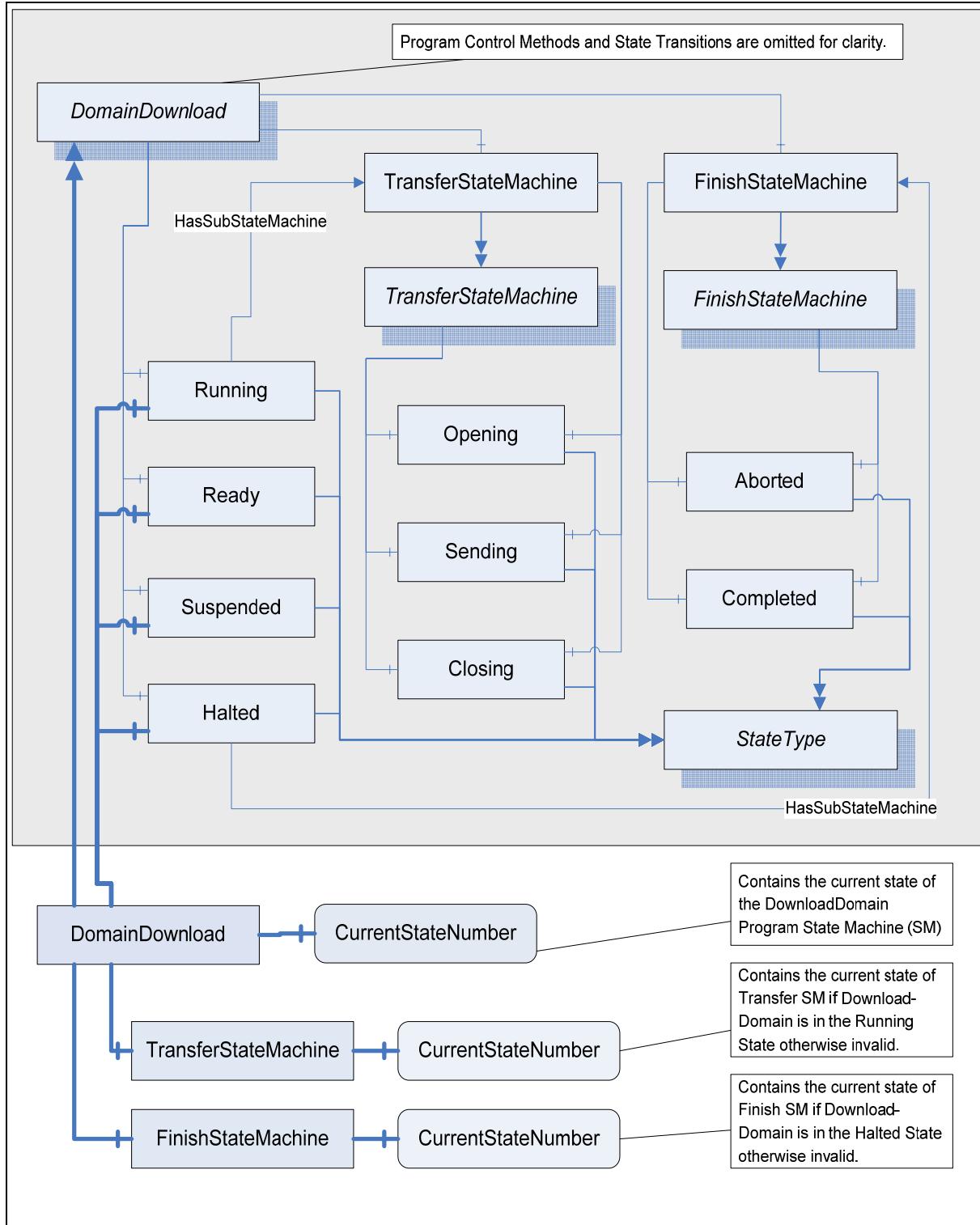


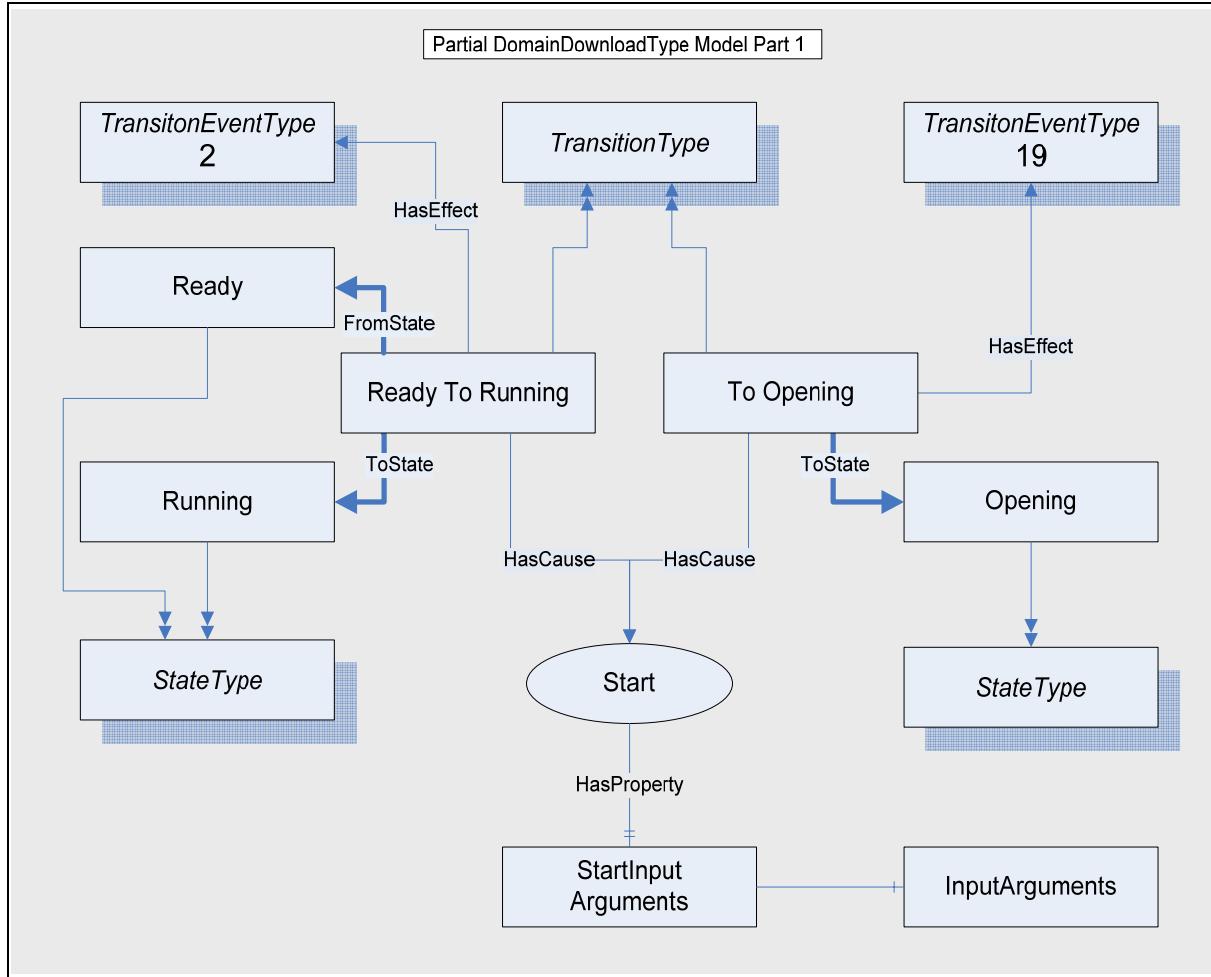
Figure 9 - DomainDownloadType Partial State Model

Table 20 specifies the *ProgramTransitionTypes* that are defined in addition to the Standard UA *ProgramTransitionTypes* specified for *Programs* in Table 7. These types associate the Transfer and Finish sub state machine states with the states of the base *Program*.

**Table 20 - Additional DomainDownload Transition Types**

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
<b>Transitions</b>					
ToSending	HasProperty	TransitionNumber	10	.PropertyType	
	ToState	Sending		.StateType	
	FromState	Opening		.StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
SendingToSending	HasProperty	TransitionNumber	11	.PropertyType	
	ToState	Sending		.StateType	
	FromState	Sending		.StateType	
	HasEffect	ProgramTransitionEventType			
SendingToClosing	HasProperty	TransitionNumber	12	.PropertyType	
	ToState	Closing		.StateType	
	FromState	Sending		.StateType	
	HasEffect	ProgramTransitionEventType			
SendingToAborted	HasProperty	TransitionNumber	13	.PropertyType	
	ToState	Aborted		.StateType	
	FromState	Closing		.StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
ClosingToCompleted	HasProperty	TransitionNumber	14	.PropertyType	
	ToState	Completed		.StateType	
	FromState	Closing		.StateType	
	HasEffect	ProgramTransitionEventType			
SendingToSuspended	HasProperty	TransitionNumber	15	.PropertyType	
	ToState	Suspended		.StateType	
	FromState	Sending		.StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
SuspendedToSending	HasProperty	TransitionNumber	16	.PropertyType	
	ToState	Sending		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
SuspendedToAborted	HasProperty	TransitionNumber	18	.PropertyType	
	ToState	Aborted		.StateType	
	FromState	Suspended		.StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional
ReadyToOpening	HasProperty	TransitionNumber	17	.PropertyType	
	ToState	Opening		.StateType	
	FromState	Ready		.StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	ProgramTransitionAuditEventType			Optional

Figure 10 Through **Figure 16** illustrate portions of the DomainDownloadType model. In each figure, the referenced States, Methods, Transitions and EventTypes are identified for one or two state transitions.



**Figure 10 – Ready To Running Model**

Figure 10 illustrates the model for the ReadyToRunning Program Transition. The transition is caused by the Start Method. The Start Method requires three input arguments. The Method *Call* service is used by the *UA Client* to invoke the *Start Method* and pass the arguments. When successful, the Program Invocation enters the Running State and the subordinate Transfer Opening State. The *UA Server* issues two event notifications, ReadyToRunning (2) and ToOpening (19).

**Table 21 - Start Method Additions**

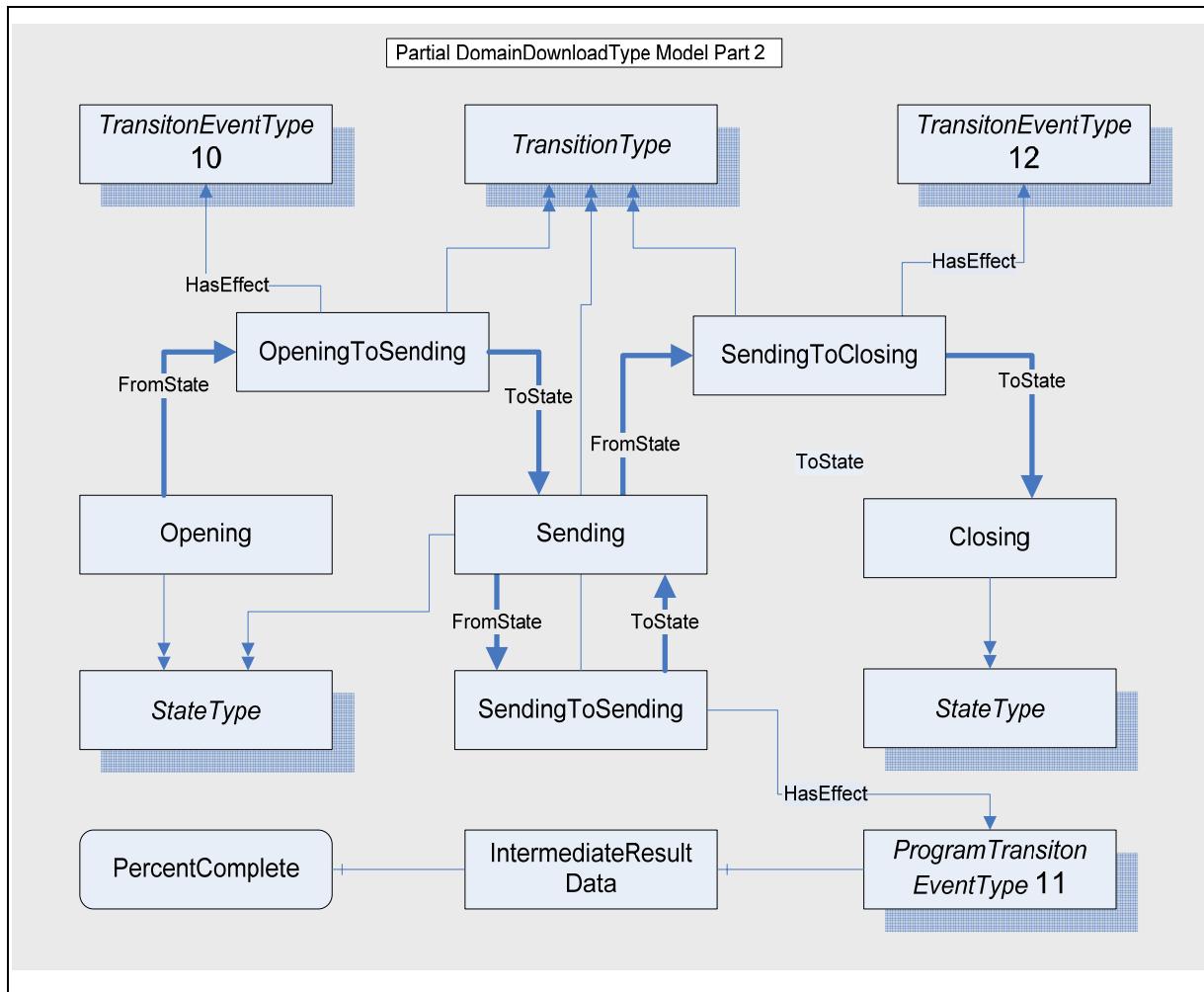
Attribute	Value				
BrowseName	Start				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArgument	Argument[]	.PropertyType	--

Table 21 specifies that the Start Method for the DomainDownloadType requires input arguments. Table 22 identifies the Start Arguments required.

**Table 22 - StartArguments**

Name	Type	Value
Argument 1	structure	
name	String	SourcePath
dataType	NodeId	StringNodeId
arraySize	Int32	-1
description	LocalizedText	The source specifier for the domain.
Argument 2	structure	
Name	String	DesinationPath
dataType	NodeId	StringNodeId
arraySize	Int32	-1
description	LocalizedText	The destination specifier for the domain.
Argument 3	structure	
name	String	DomainName
dataType	NodeId	StringNodeId
arraySize	Int32	-1
description	LocalizedText	The name of the domain.

Figure 11 illustrates the model for the Opening To Sending and the Sending to Closing Program Transitions. As specified in the transition table, these state transitions require no *Methods* to occur, but rather are driven by the internal actions of the server. Event notifiers are generated for each state transition (10-12), when they occur.

**Figure 11 - Opening To Sending To Closing Model**

Notice that a state transition can initiate and terminate at the same state (Sending). In this case the transition serves a purpose. The ProgramTransitionEventType effect referenced by the SendingToSending State Transition has an IntermediateResultData object reference. The

IntermediateResultData object serves to identify two variables whose values are obtained each time the state transition occurs. The values are sent to the *UA Client* with the event notification.

Table 24 identifies these variables as AmountTransferred and PercentageTransferred.

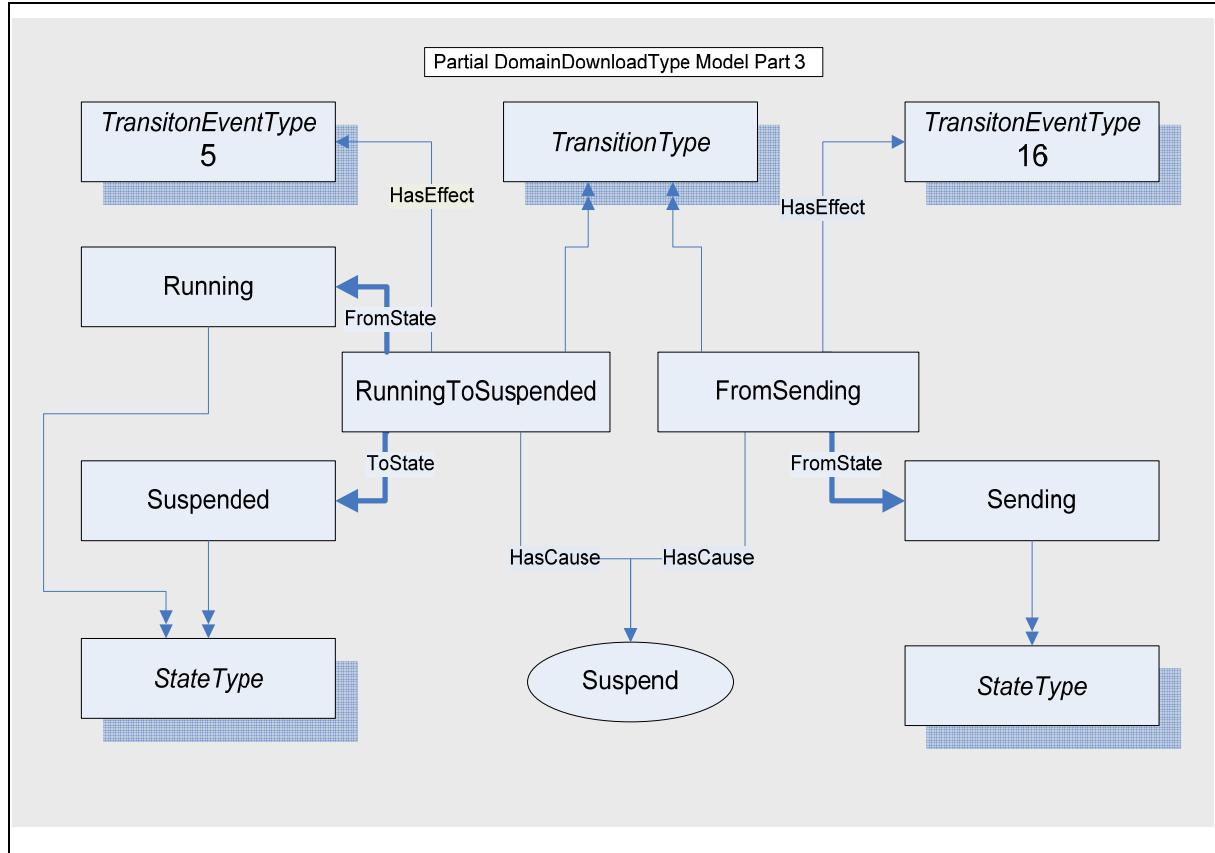
**Table 23 - Intermediate Results Object**

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	IntermediateResults				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
HasComponent	Variable	AmountTransferred	Long	VariableType	New
HasComponent	Variable	PercentageTransferred	Long	VariableType	New

**Table 24 - Immediate Result Data Variables**

Immediate Result Variables	Type	Value
Variable 1	Structure	
Name	String	AmountTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Bytes of domain data transferred.
Variable 2	Structure	
Name	String	PercentageTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Percentage of domain data transferred..

**Figure 12** Illustrates the model for the Running To Suspended state transition. The cause for this transition is the *Suspend Method*. The *UA Client* can pause the download of domain data to the control. The transition from Running to Suspended evokes the event notifiers for TransitionEventTypes 5 and 16. Note that there is no longer a valid current state for the Transfer State Machine.



**Figure 12 - Running To Suspended Model**

**Figure 13** illustrates the model for the Suspended To Running state transition. The cause for this transition is the *Resume Method*. The *UA Client* can resume the download of domain data to the control. The transition from Suspended to Running evokes the event notifiers for TransitionEventTypes 6 and 17. Now that the Running state is active, the Sending State of the Transfer State Machine is again specified for the CurrentStateNumber.

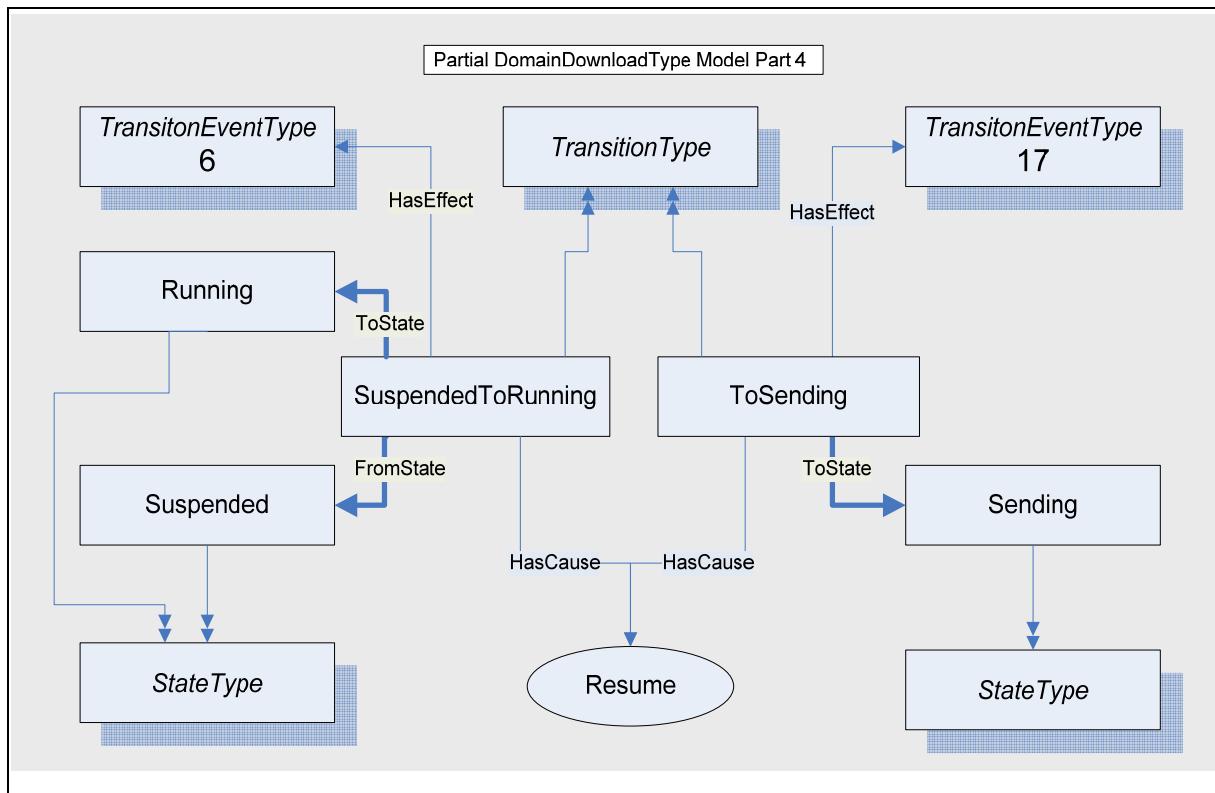
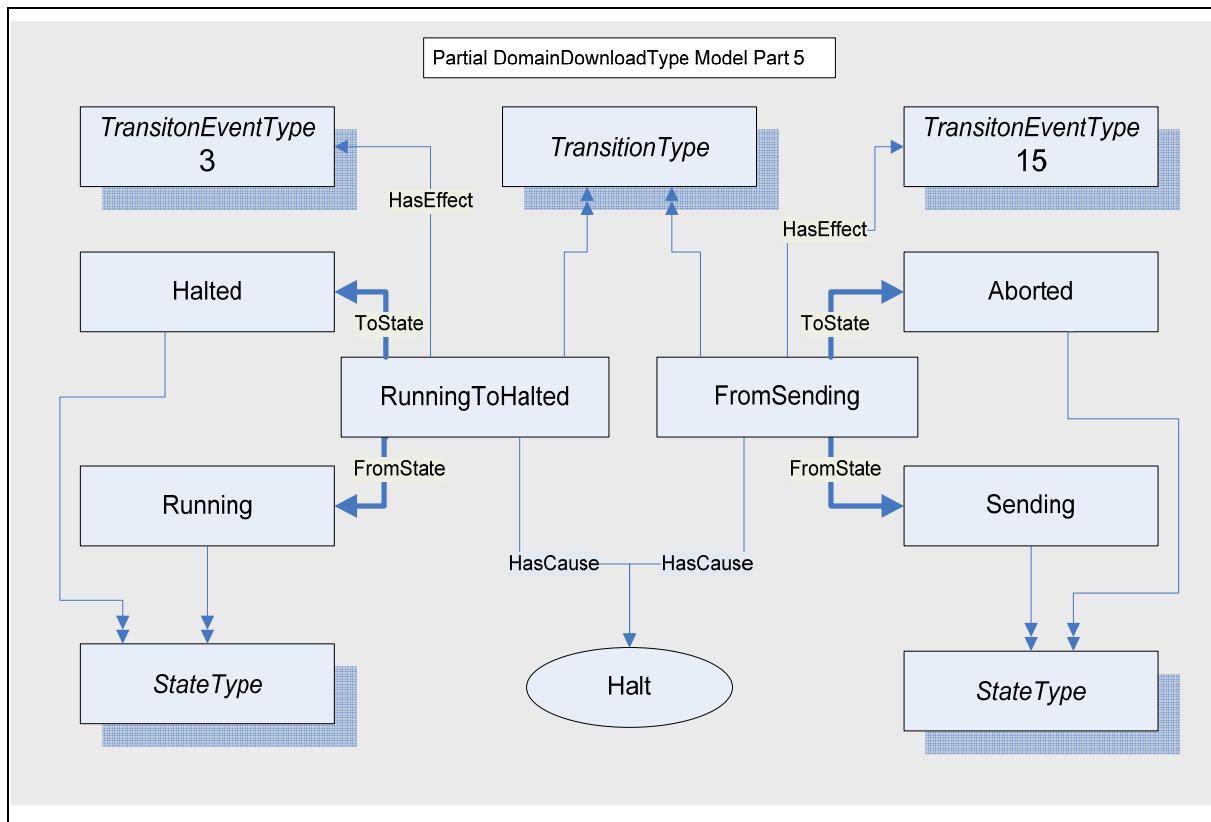


Figure 13 - Suspended To Running Model

**Figure 14** illustrates the model for the Running To Halted state transition for an abnormal termination of the domain download. The cause for this transition is the *Halt Method*. The *UA Client* can terminate the download of domain data to the control. The transition from Running To Halted evokes the event notifiers for TransitionEventTypes 3 and 15. The TransitionEventType 15 indicates the transition from the Sending State as the Running State is exited and to the Aborted State as the Halted State is entered.



**Figure 14 - Running To Halted - Aborted Model**

Figure 15 illustrates the model for the Suspended To Halted state transition for an abnormal termination of the domain download. The cause for this transition is the *Halt Method*. The *UA Client* can terminate the download of domain data to the control while it is suspended. The transition from Suspended To Halted evokes the event notifiers for TransitionEventTypes 7 and 18.

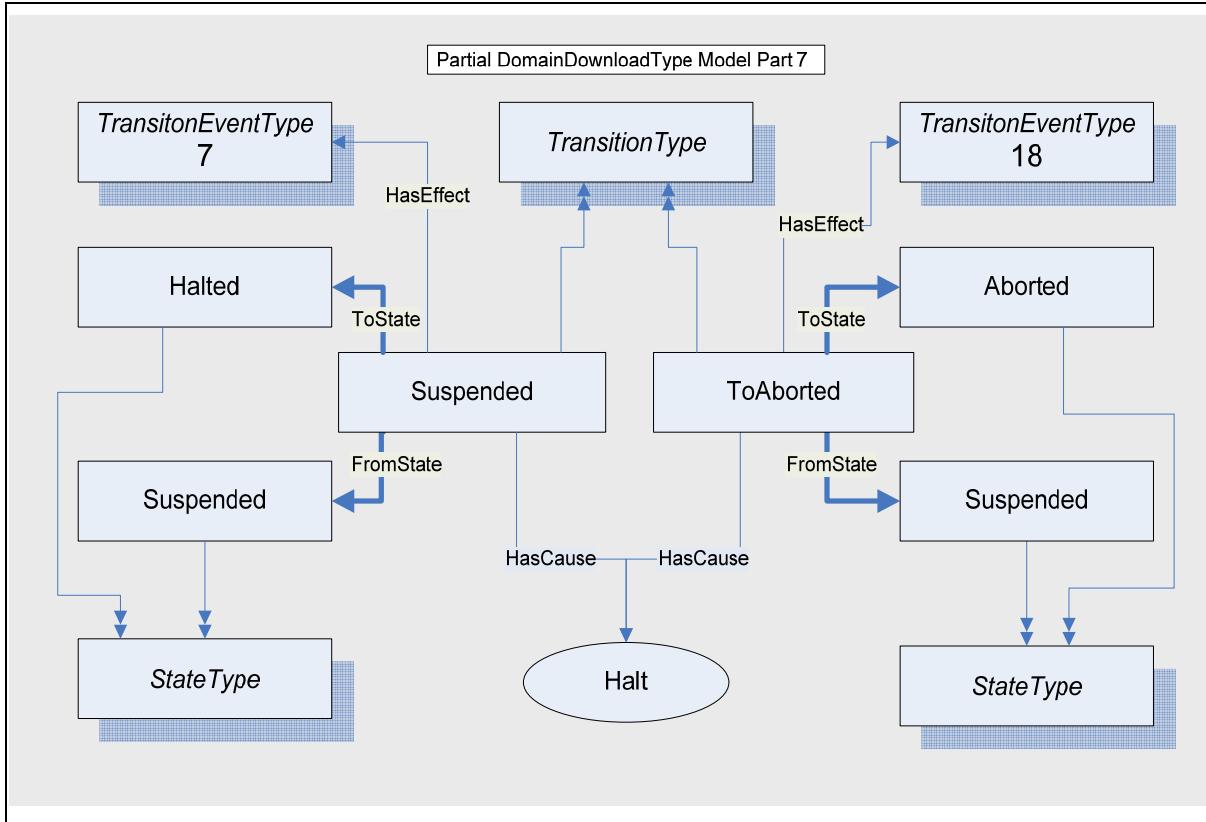


Figure 15 - Suspended To Aborted Model

**Figure 16** illustrates the model for the Running To Completed state transition for a normal termination of the domain download. The cause for this transition is internal. The transition from Closing To Halted evokes the event notifiers for TransitionEventTypes 3 and 14. The TransitionEventType 14 indicates the transition from the Closing State as the Running State is exited and to the Completed State as the Halted State is entered.

The DomainDownloadType includes a component reference to a FinalResultData object. This object references variables that persists information about the domain download once it has completed. This data can be read by UA Clients who are not subscribed to event notifications.

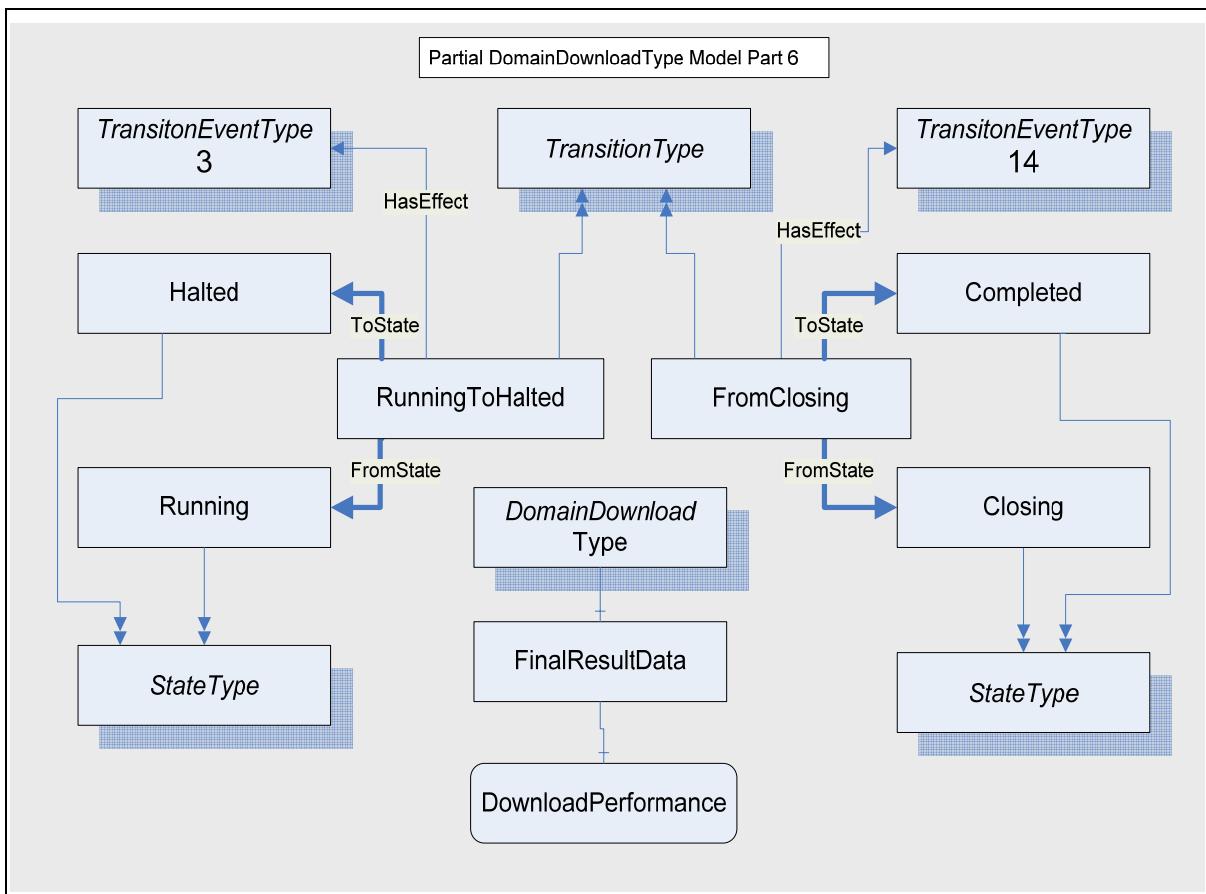
Table 25 - Final Result Data

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	FinalResultData				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
HasComponent	Variable	DownloadPerformance	Long	VariableType	New
HasComponent	Variable	FailureDetails	Long	VariableType	New

The Domain Download net transfer data rate and detailed reason for aborted downloads is retained as final result data for each Program Invocation.

**Table 26 - Final Result Variables**

Final Result Variables	Type	Value
Variable 1	Structure	
Name	String	DownloadPerformance
dataType	NodeId	Double
description	LocalizedText	Data rate for domain data transferred.
Variable 2	Structure	
Name	String	FailureDetails
dataType	NodeId	StringNodeId
description	LocalizedText	Description of reason for abort.

**Figure 16 - Running To Completed Model**

### A.2.5.3 Sequence of Operations

Figure 17 illustrates a normal sequence of service exchanges between a *UA Client* and *UA Server* that would occur during the life cycle of a *DomainDownloadType Program Invocation*.

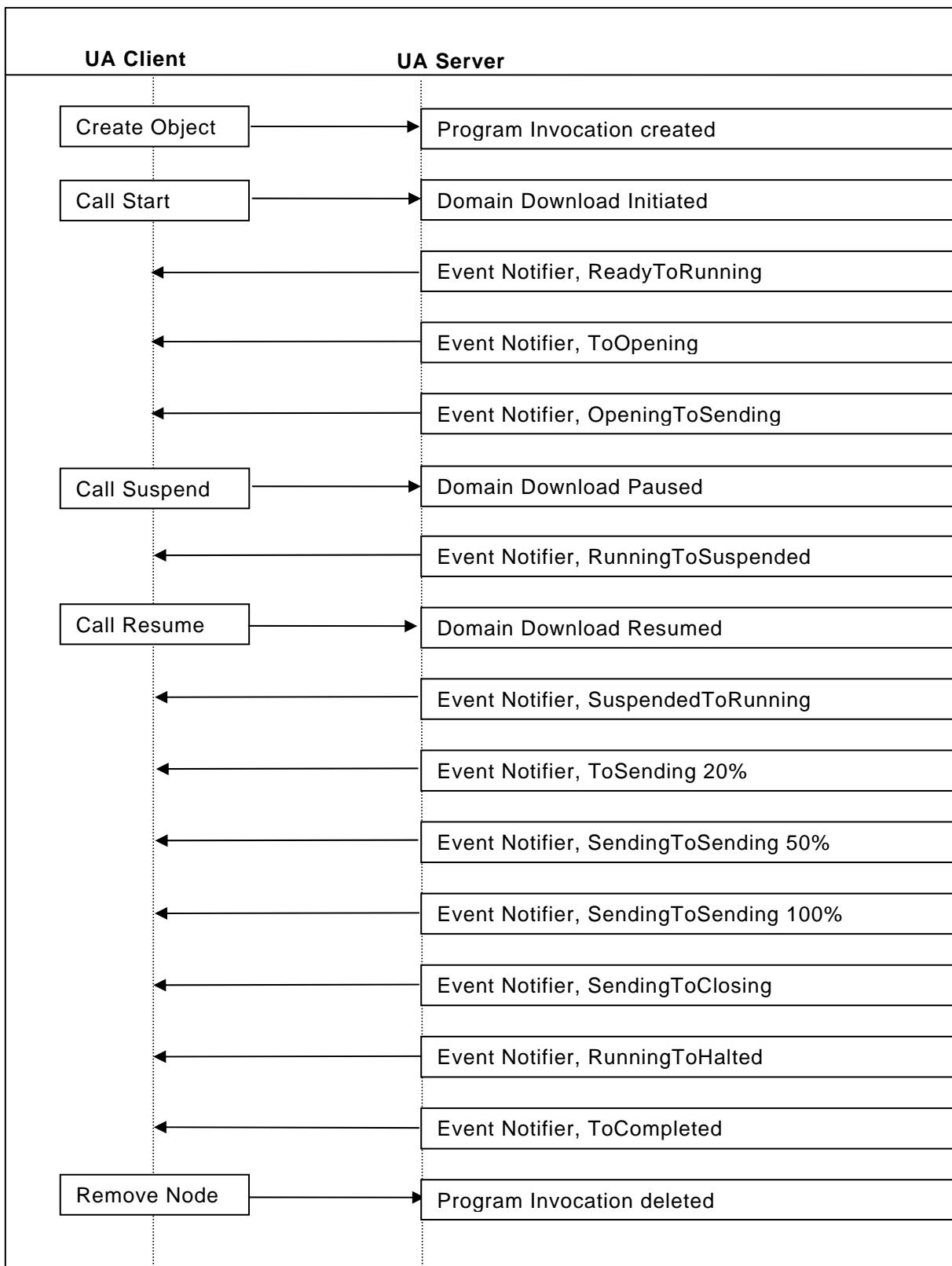
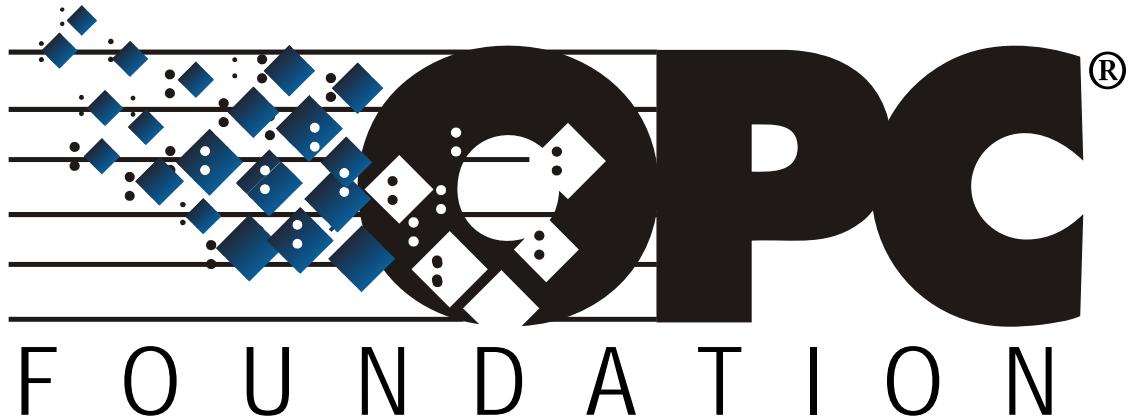


Figure 17 - Sequence of Operations

Final result Data



# **OPC Unified Architecture**

## **Specification**

### **Part 11: Historical Access**

**Version 1.00**

**January 5, 2007**

Specification Type	<u>Industry Standard Specification</u>		
Title:	OPC Unified Architecture	Date:	January 5, 2007
<u>Historical Access</u>			
Version:	<u>Release 1.00</u>	Software Source:	<u>MS-Word OPC UA Part 11 - Historical Access 1.00 Specification.doc</u>
Author:	<u>OPC Foundation</u>	Status:	<u>Release</u>

## CONTENTS

	Page
FOREWORD.....	vi
<u>AGREEMENT OF USE</u> .....	vi
1 Scope.....	1
2 Reference documents .....	1
3 Terms, definitions, and abbreviations .....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 3 terms.....	2
3.3 OPC UA Part 4 terms.....	2
3.4 OPC UA Historical Access terms.....	2
3.4.1 Aggregate .....	2
3.4.2 Annotation .....	2
3.4.3 BoundingValues .....	2
3.4.4 HistoricalNode .....	2
3.4.5 HistoricalDataNode .....	3
3.4.6 HistoricalEventNode.....	3
3.4.7 Interpolated data.....	3
3.4.8 Modified values .....	3
3.4.9 Raw data .....	3
3.4.10 StartTime / EndTime .....	3
3.4.11 TimeDomain .....	4
3.5 Abbreviations and symbols .....	4
4 Concepts.....	5
4.1 General .....	5
4.2 Data representation .....	5
4.3 Timestamps.....	6
4.4 Bounding values and time domain.....	6
4.5 Changes in AddressSpace over time.....	7
4.6 Historical Audit Events .....	8
4.7 Model .....	8
4.7.1 HistoricalDataNodes.....	8
4.7.2 HistoricalDataNodes Address Space Model .....	10
4.7.3 HistoricalDataNodes Attributes .....	10
4.7.4 HistoricalEventNodes .....	11
4.7.5 HistoricalEventNodes Address Space model.....	12
4.7.6 HistoricalEventNodes Attributes .....	13
4.8 History Objects .....	14
4.8.1 General .....	14
4.8.2 HistoryServerCapabilitiesType .....	14
4.8.3 HistoryAggregateContainerType .....	16
4.8.4 HistoryAggregateType .....	17
4.9 History DataType definitions .....	18
4.9.1 Annotation DataType.....	18
5 Historical Access specific usage of Services .....	20
5.1 General .....	20
5.2 Historical Nodes StatusCodes .....	20
5.2.1 Overview .....	20

5.2.2	Operation level result codes .....	20
5.2.3	Historian Information Bits .....	21
5.2.4	Semantics changed.....	21
5.3	HistoryReadDetails parameters .....	21
5.3.1	Overview .....	21
5.3.2	ReadEventDetails structure.....	22
5.3.3	ReadRawModifiedDetails structure .....	23
5.3.4	ReadProcessedDetails structure.....	24
5.3.5	ReadAtTimeDetails structure.....	25
5.4	HistoryData parameters .....	26
5.4.1	Overview .....	26
5.4.2	HistoryData type .....	26
5.4.3	HistoryEvent type.....	26
5.5	HistoryUpdateDetails parameter.....	26
5.5.1	Overview .....	26
5.5.2	UpdataDataDetails structure.....	27
5.5.3	UpdateEventDetails structure .....	28
5.5.4	DeleteRawModifiedDetails structure .....	29
5.5.5	DeleteAtTimeDetails structure .....	29
5.5.6	DeleteEventDetails structure .....	30
5.6	Aggregate Details .....	30
5.6.1	General .....	30
5.6.2	Common characteristics .....	30
5.6.3	Aggregate specific characteristics .....	33
6	Client conventions .....	57
6.1	How clients may request timestamps.....	57

**FIGURES**

Figure 1 - Possible OPC UA Server supporting Historical Access .....	5
Figure 2 – Representation of a <i>Variable</i> with History in the AddressSpace .....	10
Figure 3 – Representation of an <i>Event</i> with History in the AddressSpace .....	13

**TABLES**

Table 1 – Bounding Value Examples .....	7
Table 2 – HistoricalConfigurationType Definition.....	9
Table 3 – ExceptionDeviationFormat Values.....	9
Table 4 – HistoricalEventConfigurationType Definition.....	11
Table 5 – HistoryServerCapabilitiesType Definition.....	15
Table 6 – HistoryAggregatesType Definition .....	16
Table 7 – HistoryAggregateType Definition .....	17
Table 8 – Standard HistoryAggregateType BrowseNames.....	18
Table 9 – Annotation Structure .....	18
Table 10 – Annotation Definition.....	19
Table 11 – Bad operation level result codes .....	20
Table 12 – Uncertain operation level result codes.....	20
Table 13 – Good operation level result codes .....	20
Table 14 – Data Location .....	21
Table 15 – Additional Information .....	21
Table 16 – HistoryReadDetails parameterTypeld Values.....	21
Table 17 – ReadEventDetails .....	22
Table 18 – ReadRawModifiedDetails .....	23
Table 19 – ReadProcessedDetails.....	24
Table 20 – ReadAtTimeDetails .....	25
Table 21 – HistoryData Details .....	26
Table 22 – HistoryEvent Details .....	26
Table 23 – HistoryUpdateDetails parameterTypeld Values .....	27
Table 24 – UpdateDataDetails.....	27
Table 25 – UpdateEventDetails .....	28
Table 26 – DeleteRawModifiedDetails .....	29
Table 27 – DeleteAtTimeDetails .....	29
Table 28 – DeleteEventDetails .....	30
Table 29 – History Aggregate Interval Information .....	31
Table 30 – Standard History Aggregate Data Type Information .....	32
Table 31 – Time Keyword Definitions.....	58
Table 32 – Time Offset Definitions.....	58

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

This specification is for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

**Copyright © 2007, OPC Foundation, Inc.**

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

##### COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance

with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

#### TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

#### GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

#### ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here:

<http://www.opcfoundation.org/errata>

## 1 Scope

This specification is part of the overall OPC Unified Architecture specification series and defines the information model associated with Historical Access (HA). It particularly includes additional and complementary descriptions of the *NodeClasses* and *Attributes* needed for Historical Access, additional standard *Properties*, and other information and behaviour.

The complete *AddressSpace* model including all *NodeClasses* and *Attributes* is specified in [UA Part 3]. The predefined information model is defined in [UA Part 5]. The services to detect and access historical data and events, and description of the *ExtensibleParameter* types are specified in [UA Part 4].

## 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part1/>

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part3/>

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part4/>

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part5/>

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part7/>

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part8/>

[UA Part 9] OPC UA Specification: Part 9 – Alarm & Conditions, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part9/>

## 3 Terms, definitions, and abbreviations

### 3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

- 1) AddressSpace
- 2) Attribute
- 3) BrowseName
- 4) Event
- 5) Node
- 6) Nodeld
- 7) Notification
- 8) Object
- 9) ObjectType

### 3.2 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

- 1) DataVariable
- 2) EventType
- 3) Property
- 4) Variable

### 3.3 OPC UA Part 4 terms

The following terms defined in [UA Part 4] apply.

- 1) ExtensibleParameter
- 2) StatusCode
- 3) ServerTimestamp
- 4) SourceTimestamp

## 3.4 OPC UA Historical Access terms

### 3.4.1 Aggregate

Provides summarized data values of process data.

An *Aggregate* is a way to produce a set of values derived from the raw data in the historian. Clients may specify an *Aggregate* when using the *ReadHistory* service. Complete details of the various standard *Aggregates* and their behaviour are outlined in Clause 5.6. Common *Aggregates* include averages over a given time range, minimum over a time range and maximum over a time range.

### 3.4.2 Annotation

An *Annotation* is a user entered comment that is associated with an item at a given instance in time. There does not have to be a value stored at that time.

### 3.4.3 BoundingValues

*BoundingValues* are the values that are associated with the starting and ending time of an interval specified when reading from the historian. *BoundingValues* are required by clients to determine the starting and ending values when requesting raw data over a time range. If a raw data value exists at the start or end point, it is considered the bounding value even though it is part of the data request. If no raw data value exists at the start or end point, then the server will determine the boundary value, which may require data from a data point outside of the requested range. See Clause 4.4 for details on using *BoundingValues*.

### 3.4.4 HistoricalNode

A *HistoricalNode* is a term used in this document to represent any *Object*, *Variable*, *Property* or *View* in the *AddressSpace* for which a client may read and/or update historical data or events. The terms “*HistoricalNode’s history*” or “*history of a HistoricalNode*” will refer to the time series data or events stored for this *HistoricalNode* where *HistoricalNode* is an *Object*, *Variable*, *Property* or *View*. The term *HistoricalNode* refers to both *HistoricalDataNodes* and *HistoricalEventNodes*, and is used when referencing aspects of the specification that apply to accessing historical data and events.

### 3.4.5 HistoricalDataNode

A *HistoricalDataNode* represents any *Variable* or *Property* in the *AddressSpace* for which a client may read and/or update historical data. The terms “*HistoricalDataNode’s history*” or “history of a *HistoricalDataNode*” will refer to the time series data stored for this *HistoricalNode* where *HistoricalNode* is an *Object*, *Variable*, *Property* or *View*. Some examples of such data are:

- device data (like temperature sensors)
- calculated data
- status information (open/closed, moving)
- dynamically changing system data (like stock quotes)
- diagnostic data

The term *HistoricalDataNodes* is used when referencing aspects of the specification that apply to accessing historical data only.

### 3.4.6 HistoricalEventNode

A *HistoricalEventNode* represents any *Object* or *View* in the *AddressSpace* for which a client may read and/or update historical events. The terms “*HistoricalEventNode’s history*” or “history of a *HistoricalEventNode*” will refer to the time series events stored in some historical system. Some examples of such data are:

- notifications
- system alarms
- operator action events
- system triggers (such as new orders to be processed)

The term *HistoricalEventNode* is used when referencing aspects of the specification that apply to accessing historical events only.

### 3.4.7 Interpolated data

Interpolated data is data that is calculated from the data in the archive. An interpolated value is calculated from the stored data points on either side of the requested timestamp.

### 3.4.8 Modified values

A modified value is a *HistoricalDataNode’s* value that has been changed (or deleted) after it was stored in the historian. A lab data entry value is not a modified value, but if a user corrects a lab value, the original value would be considered a modified value, and would be returned during a request for modified values. Unless specified otherwise, all historical services operate on the current, or most recent, value for the specified *HistoricalDataNode* at the specified timestamp. Requests for modified values are used to access values that have been superseded.

### 3.4.9 Raw data

Raw data is data that is stored within the historian for a *HistoricalDataNode*. The data may be all data collected for the *DataItem* or it may be some subset of the data depending on the historian and the storage rules invoked when the item values were saved.

### 3.4.10 StartTime / EndTime

The *StartTime* and *EndTime* specify the bounds of a history request and define the time domain of the request. For all requests, a value falling at the end time of the time domain is not included in the domain, so that requests made for successive, contiguous time domains will include every value in the archive exactly once. See the examples in Clause 5.6.2.2

### 3.4.11 TimeDomain

The interval of time covered by a particular request, or by a particular response. In general, if the start time is earlier than or the same as the end time, the time domain is considered to begin at the start time and end just before the end time; if the end time is earlier than the start time, the time domain still begins at the start time and ends just before the end time, with time "running backward" for the particular request and response. In both cases, any value which falls exactly at the end time of the *TimeDomain* is not included in the *TimeDomain*. See the examples in section 4.4. *BoundingValues* effect the time domain as described in section 4.4.

All timestamps which can legally be represented in a *UtcTime DataType* are valid timestamps, and the server may not return an invalid argument result code due to the timestamp being outside of the range for which the server has data. See [UA Part 3] for a description of the range and granularity of this *DataType*. Servers are expected to handle out-of-bounds timestamps gracefully, and return the proper *StatusCodes* to the clients

## 3.5 Abbreviations and symbols

DA	Data Access
DCS	Distributed Control System
HD	Historical Data
PLC	Programmable Logic Controller
UA	Unified Architecture

## 4 Concepts

### 4.1 General

The OPC UA Historical Access specification defines the representation of historical time series data and historical event data in the OPC Unified Architecture. Included is the specification of the representation of historical data and events in the OPC UA AddressSpace and the definition of aggregates used in processed data retrieval.

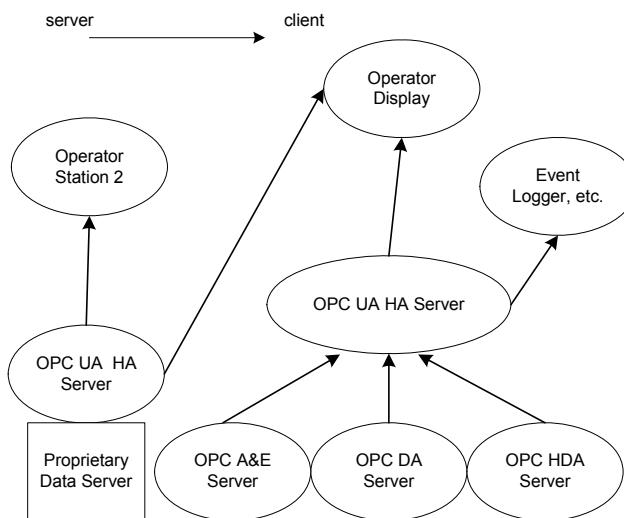
### 4.2 Data representation

An OPC UA Server supporting Historical Access provides one or more OPC UA Clients with transparent access to different historical data and/or historical event sources (e.g. process historians, event historians etc.).

The historical data or events may be located in a proprietary data archive, database or a short term buffer within memory. An OPC UA Server supporting Historical Access may or may not provide historical data and events for some or all available *Variables*, *Objects* or *Views* within the server *AddressSpace*. As with the other information models, the *AddressSpace* of an OPC UA Server supporting Historical Access is accessed via the View or Query service sets.

An OPC UA Server supporting Historical Access provides a way to access or communicate to a set of historical data and/or historical event sources. The types of sources available are a function of the server implementation.

Figure 1 illustrates how the *AddressSpace* of a UA server might consist of a broad range of different historical data and/or historical event sources.



**Figure 1 - Possible OPC UA Server supporting Historical Access**

The server may be implemented as a stand alone OPC UA Server that collects data from another OPC UA Server, a legacy OPC HDA Server, a legacy OPC DA Server, a legacy OPC A&E Server or another data source. The clients that reference the OPC UA Server supporting Historical Access for historical data may be simple trending packages that just desire values over a given time frame or they may be complex reports that require data in multiple formats.

### 4.3 Timestamps

The nature of OPC UA Historical Access requires that a single timestamp reference be used to relate the multiple data points or events, and clients may request which timestamp will be used as the reference. See [UA Part 4] for details on the *TimestampsToReturn* enumeration. An OPC UA Server supporting Historical Access will treat the various timestamp settings as described below.

For *HistoricalDataNodes*:

- SOURCE Return the *SourceTimestamp*. *SourceTimestamp* is used to determine which historical data values are returned.
- SERVER Return the *ServerTimestamp*. *ServerTimestamp* is used to determine which historical data values are returned.
- BOTH Return both the *SourceTimestamp* and *ServerTimestamp*. *SourceTimestamp* is used to determine which historical data values are returned.
- NEITHER This is not a valid setting for any HistoryRead accessing *HistoricalDataNodes*.

For *HistoricalEventNodes*:

- SOURCE Return the *SourceTimestamp*. *SourceTimestamp* is used to determine which historical events are returned.
- SERVER This is not valid setting for any HistoryRead accessing *HistoricalEventNodes*.
- BOTH This is not valid setting for any HistoryRead accessing *HistoricalEventNodes*.
- NEITHER This is not valid setting for any HistoryRead accessing *HistoricalEventNodes*.

Any reference to Timestamps through out this specification will represent either *ServerTimestamp* or *SourceTimestamp* as dictated by the type requested in the *ReadHistory* service. Some servers may not support historizing both *SourceTimestamp* and *ServerTimestamp*, but it is expect that all servers will support historizing *SourceTimestamp* (see [UA Part 7] for details on Server Profiles).

### 4.4 Bounding values and time domain

When accessing *HistoricalDataNodes* via the *ReadHistory* Service, requests can set a flag, *returnBounds*, indicating that a *BoundingBox* are requested. For a complete description of the extensible Parameter *HistoryReadDetails* that include all of these parameters see section 5.3. The concept of bounding values and how they affect the time domain that is requested as part of the *ReadHistory* request is further explained in this section. This section also provides examples of *TimeDomains* to further illustrate the expected behaviour.

When making a request for historical data using the *ReadHistory* Service, required parameters include a *startTime* and *endTime*. These two parameters define the *TimeDomain* of the *ReadHistory* request. This *TimeDomain* includes all values between the *StartTime* and *EndTime*, and any value that falls exactly on the *StartTime*, but not any value that falls exactly on the *EndTime*. For example, assuming bounding values are not requested, if data is requested from 1:00 to 1:05, and then from 1:05 to 1:10, a value that exists at exactly 1:05 would be included in the second request, but not in the first.

Given that a historian has values stored at 5:00, 5:02, 5:03, 5:05 and 5:06, the data returned from a RAW data call is given by Table 1. In the table, FIRST stands for a tuple with a value of *DateTime.Min*, a timestamp of the specified *StartTime*, and a *StatusCode* of *Bad\_NoBound*. LAST stands for a tuple with a value of *DateTime.Max*, a timestamp of the specified *EndTime*, and a *StatusCode* of *Bad\_NoBound*

**Table 1 – Bounding Value Examples**

Start Time	End Time	numValuesPer Node	Bounds	Data Returned
5:00	5:05	0	Yes	5:00, 5:02, 5:03, 5:05
5:00	5:05	0	No	5:00, 5:02, 5:03
5:01	5:04	0	Yes	5:00, 5:02, 5:03, 5:05
5:01	5:04	0	No	5:02, 5:03
5:05	5:00	0	Yes	5:05, 5:03, 5:02, 5:00
5:05	5:00	0	No	5:05, 5:03, 5:02
5:04	5:01	0	Yes	5:05, 5:03, 5:02, 5:00
5:04	5:01	0	No	5:03, 5:02
4:59	5:05	0	Yes	FIRST, 5:00, 5:02, 5:03, 5:05
4:59	5:05	0	No	5:00, 5:02, 5:03
5:01	5:07	0	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST
5:01	5:07	0	No	5:02, 5:03, 5:05, 5:06
5:00	5:05	3	Yes	5:00, 5:02, 5:03
5:00	5:05	3	No	5:00, 5:02, 5:03
5:01	5:04	3	Yes	5:00, 5:02, 5:03
5:01	5:04	3	No	5:02, 5:03
5:05	5:00	3	Yes	5:05, 5:03, 5:02
5:05	5:00	3	No	5:05, 5:03, 5:02
5:04	5:01	3	Yes	5:05, 5:03, 5:02
5:04	5:01	3	No	5:03, 5:02
4:59	5:05	3	Yes	FIRST, 5:00, 5:02
4:59	5:05	3	No	5:00, 5:02, 5:03
5:01	5:07	3	Yes	5:00, 5:02, 5:03
5:01	5:07	3	No	5:02, 5:03, 5:05
5:00	DateTime.Max	3	Yes	5:00, 5:02, 5:03
5:00	DateTime.Max	3	No	5:00, 5:02, 5:03
5:00	DateTime.Max	6	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST
5:00	DateTime.Max	6	No	5:00, 5:02, 5:03, 5:05, 5:06
DateTime.Min	5:06	3	Yes	5:06,5:05,5:03
DateTime.Min	5:06	3	No	5:06,5:05,5:03
DateTime.Min	5:06	6	Yes	5:06,5:05,5:03,5:02,5:00,FIRST
DateTime.Min	5:06	6	No	5:06, 5:05, 5:03, 5:02, 5:00
4:48	4:48	0	Yes	FIRST,5:00
4:48	4:48	0	No	NODATA
4:48	4:48	1	Yes	FIRST
4:48	4:48	1	No	NODATA
4:48	4:48	2	Yes	FIRST,5:00
5:00	5:00	0	Yes	5:00,5:02
5:00	5:00	0	No	5:00
5:00	5:00	1	Yes	5:00
5:00	5:00	1	No	5:00
5:01	5:01	0	Yes	5:00, 5:02
5:01	5:01	0	No	NODATA
5:01	5:01	1	Yes	5:00
5:01	5:01	1	No	NODATA

#### 4.5 Changes in AddressSpace over time.

Clients use the browse Services of the View Service Set to navigate through the AddressSpace to discover the Properties supported by one or more specified Nodes. See [UA Part 4] These Services provide the most current information about the AddressSpace. It

is possible and probable that the *AddressSpace* of a *Server* will change over time (i.e. *TypeDefinitions* may change, *NodeIds* may be modified, added or deleted).

Server developers and administrators need to be aware that modifying the *AddressSpace* may impact a *Client*'s ability to access historical information. If the history for a *HistoricalNode* is still required, but the *HistoricalNode* is no longer an active point, the object should be maintained in the address space, with the appropriate *Access Level* attribute and Historizing attribute settings (see [UA Part 3] for details on access levels).

## 4.6 Historical Audit Events

*AuditEvents* are generated as a result of an action taken on the server by a client of the server. For example, in response to a client issuing a write to a *Variable*, the server would generate an *AuditEvent* describing the *Variable* as the source and the user and client session as the initiators of the *Event*.

Servers must generate events of the *AuditUpdateEventType* or a sub-type of this type for all invocations of the *HistoryUpdate* service on any *HistoricalNode*. See [UA Part 3] and [UA Part 5] for details on the *AuditUpdateEventType* Model. In the case where the *HistoryUpdate* service is invoked to insert *HistoricalEvents*, the *AuditUpdateEvent* must include the *EventId* of the inserted event and a description that indicates that the event was inserted. All other updates must follow the guidelines provided in the *AuditUpdateEventType* Model.

## 4.7 Model

### 4.7.1 HistoricalDataNodes

#### 4.7.1.1 General

The Historical Data model defines additional *ReferenceTypes*, *ModellingRules* and *ObjectTypes*. These descriptions also include required use cases for *HistoricalDataNodes*.

#### 4.7.1.2 HasHistoricalConfiguration

The *HasHistoricalConfiguration ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to bind a *DataVariable* or a *Property* to its *HistoricalConfigurationType Object*. All *DataVariables* and *Properties* that expose historical data must have exactly one *HasHistoricalConfiguration* reference.

The *SourceNode* of this *ReferenceType* must be a *DataVariable* or *Property*. The *TargetNode* must be an *Object* of the *ObjectType HistoricalConfigurationType*.

Multiple *DataVariables* or *Properties* may reference the same *HistoricalConfigurationType Object*.

#### 4.7.1.3 OptionalNew

*ModellingRules* are an extendable concept in OPC UA; [UA Part 3] defines the rules "None", "Shared" and "New". Some Historical Access properties, however, are optional and this part therefore also uses *OptionalNew ModellingRule*. This *ModellingRule* is defined in [UA Part 8]

#### 4.7.1.4 HistoricalConfigurationType

The Historical Access Data model extends the standard type model by defining an additional *ObjectType*, the *HistoricalConfigurationType*. This *HistoricalConfigurationType* defines the general characteristics of a node that defines the historical configuration of any variable or property that is defined to contain history. It is formally defined in Table 2.

**Table 2 – HistoricalConfigurationType Definition**

<b>Attribute</b>	<b>Value</b>				
<b>BrowseName</b>	HistoricalConfigurationType				
<b>IsAbstract</b>	False				
<b>References</b>	<b>NodeClass</b>	<b>BrowseName</b>	<b>DataType</b>	<b>TypeDefinition</b>	<b>ModellingRule</b>
Subtype of the <i>BaseObjectType</i> defined in [UA Part 5]					
HasProperty	Variable	Stepped	Boolean	.PropertyType	New
HasProperty	Variable	Definition	String	.PropertyType	Optional New
HasProperty	Variable	MaxTimeInterval	Duration	.PropertyType	Optional New
HasProperty	Variable	MinTimeInterval	Duration	.PropertyType	Optional New
HasProperty	Variable	ExceptionDeviation	Double	.PropertyType	Optional New
HasProperty	Variable	ExceptionDeviationFormat	Enum	.PropertyType	Optional New

**Stepped** specifies whether the historical data was collected in such a manner that it should be displayed as interpolated (sloped Lines between point) or as Stepped (vertically-connected horizontal lines between points) when raw data is examined. This property also effect how some aggregates are calculated. A value of True indicates stepped mode. A value of False indicates interpolated mode. The default value is False.

**Definition** is a vendor-specific, human readable string that specifies how the value of this *HistoricalDataNode* is calculated. Definition is non-localized and will often contain an equation that can be parsed by certain clients.

Example:      *Definition ::= "(TempA - 25) + TempB"*

**MaxTimeInterval** specifies the maximum interval between data points in the history repository regardless of their value change (see [UA Part 4] for definition of *Duration*).

**MinTimeInterval** specifies the minimum interval between data points in the history repository regardless of their value change (see [UA Part 4] for definition of *Duration*).

**ExceptionDeviation** specifies the minimum amount that the data for the *HistoricalDataNode* must change in order for the change to be reported to the history database.

**ExceptionDeviationFormat** specifies how the ExceptionDeviation is determined. Its values are defined in Table 3.

**Table 3 – ExceptionDeviationFormat Values**

<b>Value</b>	<b>Description</b>
ABSOLUTE_VALUE_0	ExceptionDeviation is an absolute Value.
PERCENT_OF_RANGE_1	ExceptionDeviation is a percent of InstrumentRange (See [UA Part 8])
PERCENT_OF_VALUE_2	ExceptionDeviation is a percent of Value.

#### 4.7.2 HistoricalDataNodes Address Space Model

*HistoricalDataNodes* are always part of other *Nodes* in the AddressSpace. They are never defined by themselves. A simple example of a container for *HistoricalDataNodes* would be a "Folder Object". But it can be an *Object* of any other type.

Figure 2 illustrates the basic AddressSpace model of a DataVariable that includes History.

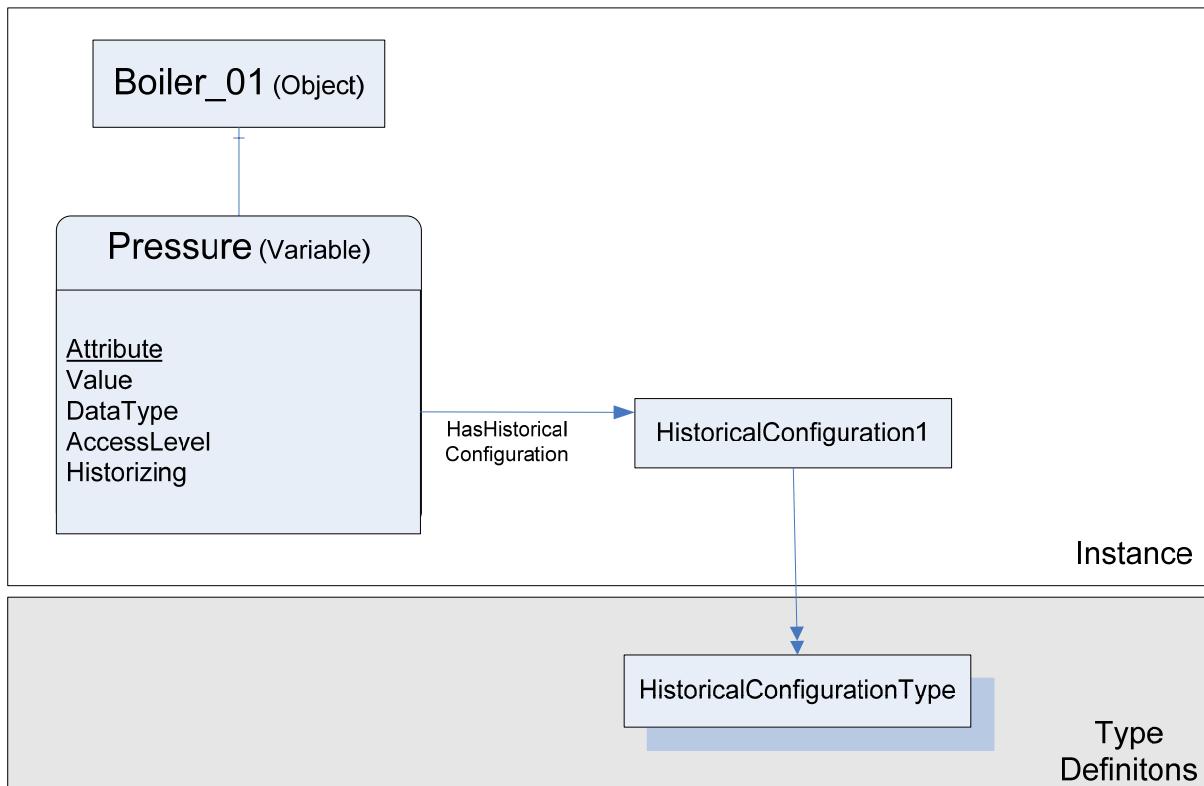


Figure 2 – Representation of a *Variable* with History in the AddressSpace

Each *Variable* with history must have the *Historizing* attribute (see [UA Part 3]) defined and include a *HasHistoricalConfiguration* reference. The *HistoricalConfigurationType* Instance must define the *stepped* property, but may also define any of the optional properties.

Not every *Variable* in the *AddressSpace* might contain history data. To see if history data is available, a client will look for the *HistoryRead/Write* states in the *AccessLevel Attribute* (see [UA Part 3] for details on use of this *Attribute*).

Figure 2 only shows a subset of *Attributes* and *Properties*. Other *Attributes* as that are defined for *Variables* in [UA Part 3], and in the following sections – may also be available.

#### 4.7.3 HistoricalDataNodes Attributes

This section lists the *Attributes* of *Variables* that have particular importance for historical data. They are specified in detail in [UA Part 3]. The following *Attributes* are particularly important for *HistoricalDataNodes*.

- Value
- DataType
- AccessLevel
- Historizing

*Value* is the value of the *Variable*. Its data type is defined by the *DataType Attribute*. This is the *Attribute* for which historical data is collected. The *AccessLevel* attribute defines the server's basic ability to access history data for this *Variable*.

When a client requests the *Value* attribute, the server in addition always returns a *StatusCode* (the quality and the server's ability to access/provide the value) and a *ServerTimestamp* and/or a *SourceTimestamp*. See [UA Part 4] for details on *StatusCodes* and the meaning of the two timestamps. Specific *StatusCodes* for *HistoricalDataNodes* are defined in Clause 5.2.

#### 4.7.4 HistoricalEventNodes

##### 4.7.4.1 General

The Historical Event model defines additional *ReferenceTypes*, *ModellingRules* and *ObjectTypes*. These descriptions also include required use cases for *HistoricalEventNodes*.

##### 4.7.4.2 HasHistoricalEventConfiguration

The *HasHistoricalEventConfiguration ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantics of this *ReferenceType* is to bind an *Object* which exposes an *EventNotifier* that exposes historical events (i.e. has the *EventNotifier Attribute* for HistoryRead or HistoryWrite set to one) to a *HistoricalEventConfigurationType Object*. All objects which expose *EventNotifiers* that expose historical events must have exactly one *HasHistoricalEventConfiguration* reference.

The *SourceNode* of this *ReferenceType* must be an *Object* which exposes a *EventNotifier* that exposes historical events. The *TargetNode* must be an *Object* of the *ObjectType* *HistoricalEventConfigurationType*.

Multiple *EventNotifiers* may reference the same *HistoricalEventConfigurationType Object*.

##### 4.7.4.3 OptionalNew

*ModellingRules* are an extendable concept in OPC UA; [UA Part 3] defines the rules "None", "Shared" and "New". Some Historical Access properties, however, are optional and this part therefore also uses *OptionalNew ModellingRule*. This *ModellingRule* is defined in [UA Part 8]

##### 4.7.4.4 HistoricalEventConfigurationType

The Historical Access Event model extends the standard type model by defining an additional *ObjectType*, the *HistoricalEventConfigurationType*. This *HistoricalEventConfigurationType* defines the general characteristics of a node that defines the historical configuration of any *Object* that exposes an *EventNotifier* that exposes historical events. It is formally defined in Table 4

**Table 4 – HistoricalEventConfigurationType Definition**

Attribute	Value		
BrowseName	HistoricalEventConfigurationType		
IsAbstract	False		
References	NodeClass	BrowseName	ModellingRule
Subtype of the <i>BaseObjectType</i> defined in [UA Part 5]			
GeneratesEvent	ReferenceType	HistoricalEvents01	OptionalNew

**HistoricalEvents:** The semantic of this *ReferenceType* is to relate *EventTypes* that are being historized to the object that they are available from. This *ReferenceType* and any subtypes

are intended to be used for discovery of types of historical *Events* in a server. They are not required to be present for a server to historize *Events*. This *ReferenceType* is as described in [UA Part 3]. This application of this *ReferenceType* further restricts the use as follows:

The *SourceNode* of this *ReferenceType* must be a *Node* that is of type *HistoricalEventConfigurationType*

The *TargetNode* of this *ReferenceType* must be the *EventType* that is available as historical events.

The *Object* of *HistoricalEventConfigurationType* can expose more than one of these references. The resulting list of *EventType Nodes* (and their sub types) is the summary list the types of *Events* that are available as historical events. A server that does not historize all attributes associated with a given *EventType* should define a new *EventType* that describes the attributes that are being historized and add a *Reference* to it from its *HistoricalEventConfigurationType Object*. The *BrowseName* of the reference can be any name that is unique for the *Object* of *HistoricalEventConfigurationType* and follows the naming requirements of *BrowseNames*. A user should review all *GenerateEvent* references in the *Object* of *HistoricalEventConfigurationType* that is associated with the *Object* that is exposing historical events

#### 4.7.5 HistoricalEventNodes Address Space model

*HistoricalEventNodes* are *Objects* or *Views* in the *AddressSpace* that expose historical *Events*. These *Nodes* are identified via the *EventNotifier Attribute*, and provide some historical subset of the *Events* generated by the server.

Each *HistoricalEventNode* is represented by an *Object* or *View* with a specific set of *Attributes*. Additional characteristics of *HistoricalEventNodes* are defined using *Properties* (i.e. *Variables* that are referenced using *HasProperty References*). For a detailed description of *Variable* and *Properties* see [UA Part 3]. This specification defines *Properties* that have been found useful for a large range of historical event clients.

Not every *Object* or *View* in the *AddressSpace* may be a *HistoricalEventNode*. To qualify as *HistoricalEventNodes*, a *Node* has to contain historical events. To see if historical events are available, a client will look for the *HistoryRead/Write* states in the *EventNotifier Attribute*. See [UA Part 3] for details on use of this *Attribute*.

Figure 3 illustrates the basic *AddressSpace* model of an *Event* that includes History.

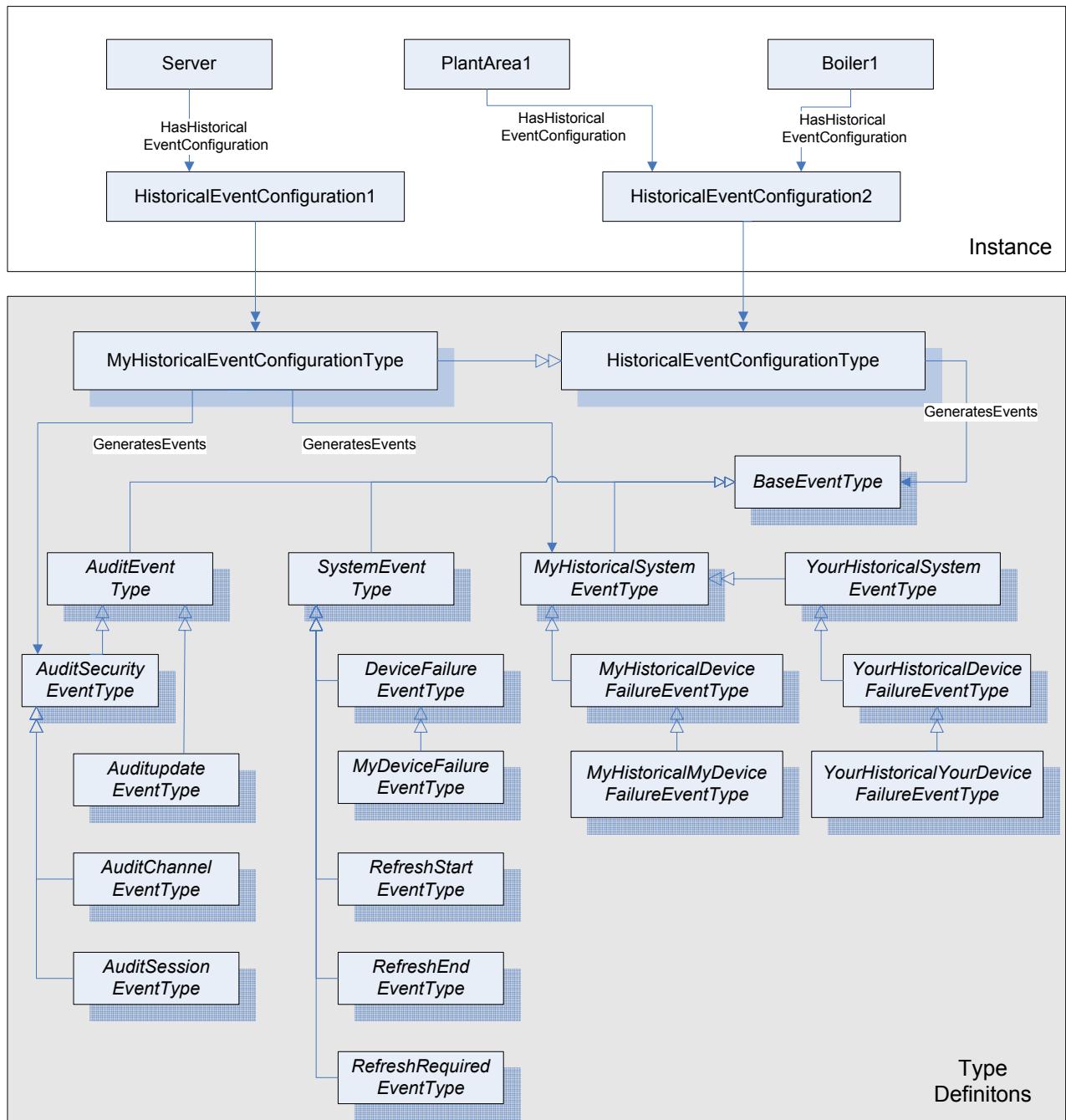


Figure 3 – Representation of an *Event* with History in the AddressSpace

#### 4.7.6 HistoricalEventNodes Attributes

This section lists the *Attributes* of *Objects* or *Views* that have particular importance for historical events. They are specified in detail in [UA Part 3]. The following *Attributes* are particularly important for *HistoricalEventNodes*.

- EventNotifier

The *EventNotifier Attribute* is used to indicate if the *Node* can be used to read and/or update historical Events.

## 4.8 History Objects

### 4.8.1 General

OPC UA servers can support several different functionalities and capabilities. The following standard *Objects* are used to expose these capabilities in a common fashion, and there are several standard defined concepts that can be extended by vendors.

### 4.8.2 HistoryServerCapabilitiesType

The *ServerCapabilitiesType Objects* for any OPC UA Server supporting Historical Access must contain a *Reference* to a *HistoryServerCapabilitiesType Object*.

The content of this *BaseObjectType* is already defined by its type definition in [UA Part 5]. The *Object* extensions are formally defined in Table 5.

**Table 5 – HistoryServerCapabilitiesType Definition**

Attribute	Value				
References	Node Class	Browse Name	Data Type	Type Definition	Instantiation Rule
BrowseName		HistoryServerCapabilitiesType			
IsAbstract		False			
ArraySize		-1			
HasProperty	Variable	AccessHistoryDataCapability	Boolean	.PropertyType	New
HasProperty	Variable	AccessEventsCapability	Boolean	.PropertyType	New
HasProperty	Variable	MaxReturnValues	UInt32	.PropertyType	New
HasProperty	Variable	TreatUncertainAsBad	Boolean	.PropertyType	New
HasProperty	Variable	PercentDataBad	UInt8	.PropertyType	New
HasProperty	Variable	PercentDataGood	UInt8	.PropertyType	New
HasProperty	Variable	SteppedInterpolationMode	Boolean	.PropertyType	New
HasProperty	Variable	InsertDataCapability	Boolean	.PropertyType	New
HasProperty	Variable	ReplaceDataCapability	Boolean	.PropertyType	New
HasProperty	Variable	UpdateCapability	Boolean	.PropertyType	New
HasProperty	Variable	DeleteRawCapability	Boolean	.PropertyType	New
HasProperty	Variable	DeleteAtTimeCapability	Boolean	.PropertyType	New
HasProperty	Object	HistoryAggregates	--	HistoryAggregateContainerType	New

All UA server that support Historical data access must include the HistoryServerCapabilities as part of its ServerCapabilities. If any of these *Properties* do not contain a valid value, the client application should use the default values.

The *AccessHistoryDataCapability Variable* defines if the server supports access to historical data values. A value of True indicates the server supports access to history for *HistoricalNodes*, a value of False indicates the server does not support access to history for *HistoricalNodes*. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessEventsCapability* must have a value of True for the server to be a valid OPC UA Server supporting Historical Access.

The *AccessHistoryEventCapability Variable* defines if the server supports access to historical events. A value of True indicates the server supports access to history of events, a value of False indicates the server does not support access to history of events. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessEventsCapability* must have a value of True for the server to be a valid OPC UA Server supporting Historical Access.

The *MaxReturnValues Variable* defines maximum number of values that can be returned by the server for each *HistoricalNode* accessed during a request. A value of 0 indicates that the server forces no limit on the number of values it can return. It is valid for a server to limit the number of returned values and return a continuation point even if *MaxReturnValues* = 0. For example, it is possible that although the server does not impose any restrictions, the underlying system may impose a limit that the server is not aware of. The default value is 0.

The *TreatUncertainAsBad Variable* indicates how the server treats data returned with a *StatusCode* severity *Uncertain* with respect to aggregate calculations. A value of True indicates the server considers the severity equivalent to *Bad*, a value of False indicates the server considers the severity equivalent to *Good*. The default value is True.

The *PercentDataBad Variable* indicates the Maximum percentage of bad data in a given interval above which would cause the *StatusCode* for the given interval for processed data request to be set to *Bad*. (*Uncertain* is treated as defined above). For values equal to or below this percentage the **StatusCode** would be *Uncertain* or *Good*. For details on which aggregates use the *PercentDataBad Variable*, see the definition of each aggregate. The default value is 0.

The *PercentDataGood Variable* indicates the minimum percentage of *Good* data in a given interval which would cause the *StatusCode* for the given interval for the processed data requests to be set to *Good*. For values below this percentage the **StatusCode** would be

Uncertain or Bad. For details on which aggregates use the *PercentDataGood Variable*, see the definition of each aggregate. The default value is 100.

The *SteppedInterpolationMode Variable* indicates how the server interpolates data when no boundary value exists (i.e. interpolating into the future from the last known value). A value of False indicates that the server will use a stepped format, and hold the last known value constant. A value of True indicates the server will project the value using straight line interpolation. The default value is False.

The *InsertDataCapability Variable* indicates support for the Insert capability. A value of True indicates the server supports the capability to insert new values in history, but not overwrite existing values. The default value is False.

The *ReplaceDataCapability Variable* indicates support for the Replace capability. A value of True indicates the server supports the capability to replace existing values in history, but will not insert new values. The default value is False.

The *UpdateCapability Variable* indicates support for the Update capability. A value of True indicates the server supports the capability to insert new values into history if none exists, and replace values that currently exist. The default value is False.

The *DeleteRawCapability Variable* indicates support for the delete raw values capability. A value of True indicates the server supports the capability to delete raw values in history. The default value is False.

The *DeleteAtTimeCapability Variable* indicates support for the delete at time capability. A value of True indicates the server supports the capability to delete a value at a specified time. The default value is False.

The *HistoryAggregates Object* defines the aggregate capabilities supported by the UA server. This Object has 'HasComponent' references to zero or more *HistoryAggregate Objects* which define a specific aggregate supported by the server. The *HistoryAggregate Objects* are instances of the *HistoryAggregateType ObjectType* defined in Clause 4.8.3.

#### 4.8.3 HistoryAggregateContainerType

This *ObjectType* defines a container for the standard OPC UA Server supporting Historical Access and vendor aggregates supported by the UA server. This *ObjectType* is formally defined in Table 6. All servers that expose aggregates must define this *ObjectType* and define the aggregates that it exposes.

**Table 6 – HistoryAggregatesType Definition**

Attribute	Value				
BrowseName	HistoryAggregateContainerType				
IsAbstract	False				
References	Node Class	BrowseName	DataType	Type Definition	Mod. Rule
Subtype of the <i>BaseObjectType</i> defined in [UA Part 5]					

#### 4.8.4 HistoryAggregateType

This *ObjectType* defines an aggregate supported by a UA server. This object is formally defined in Table 7.

**Table 7 – HistoryAggregateType Definition**

Attribute	Value				
BrowseName	HistoryAggregateType				
IsAbstract	False				
References	Node Class	BrowseName	DataType	Type Definition	Mod. Rule
Subtype of the <i>BaseObjectType</i> defined in [UA Part 5]					

For the *HistoryAggregateType*, the *Description Attribute* (inherited from the *Base NodeClass*), is mandatory. The *Description Attribute* provides a localized description of the aggregate

Table 8 outlines the *BrowseName* and *Description* for the standard aggregates.

**Table 8 – Standard HistoryAggregateType BrowseNames**

BrowseName	Description
	<b>Interpolation Aggregate</b>
Interpolative	Do not retrieve an aggregate. This is used for retrieving interpolated Values.
	<b>Data Averaging and Summation Aggregates</b>
Average	Retrieve the average data over the resample interval.
TimeAverage	Retrieve the time weighted average data over the resample interval.
Total	Retrieve the sum of the data over the resample interval.
TotalizeAverage	Retrieve the totalized Value (time integral) of the data over the resample interval.
	<b>Data Variation Aggregates</b>
Minimum	Retrieve the minimum Value in the resample interval.
Maximum	Retrieve the maximum Value in the resample interval.
MinimumActualTime	Retrieve the minimum value in the resample interval and the Timestamp of the minimum value.
MaximumActualTime	Retrieve the maximum value in the resample interval and the Timestamp of the maximum value.
Range	Retrieve the difference between the minimum and maximum Value over the sample interval.
	<b>Counting Aggregates</b>
AnnotationCount	Retrieve the number of annotations in the interval.
Count	Retrieve the number of raw Values over the resample interval.
DurationInState0	Retrieve the time (in seconds) a Boolean was in a 0 state
DurationInState1	Retrieve the time (in seconds) a Boolean was in a 1 state
NumberOfTransitions	Retrieve the number of state changes a Boolean value experienced in the interval
	<b>Time Aggregates</b>
Start	Retrieve the Value at the beginning of the resample interval. The time stamp is the time stamp of the beginning of the interval.
End	Retrieve the Value at the end of the resample interval. The time stamp is the time stamp of the end of the interval.
Delta	Retrieve the difference between the first and last Value in the resample interval.
	<b>Data Quality Aggregates</b>
DurationGood	Retrieve the duration (in seconds) of time in the interval during which the data is good.
DurationBad	Retrieve the duration (in seconds) of time in the interval during which the data is bad.
PercentGood	Retrieve the percent of data (0 to 100) in the interval which has good StatusCode.
PercentBad	Retrieve the percent of data (0 to 100) in the interval which has bad StatusCode.
WorstQuality	Retrieve the worst StatusCode of data in the interval.

## 4.9 History DataType definitions

### 4.9.1 Annotation DataType

This *DataType* describes annotation information for the history data items. Its elements are defined in Table 9.

**Table 9 – Annotation Structure**

Name	Type	Description
Annotation	structure	
message	String	Annotation message or text
userName	String	The user that added the annotation, as supplied by underlying system.
annotationTime	UtcTime	The time the annotation was added. This will probably be different than the SourceTimestamp

Its representation in the *AddressSpace* is defined in Table 10.

**Table 10 – Annotation Definition**

Attributes	Value
Browse Name	Annotation

## 5 Historical Access specific usage of Services

### 5.1 General

[UA Part 4] specifies all Services needed for OPC UA Historical Access. In particular:

- The *Browse Service Set* or *Query Service Set* to detect *HistoricalNodes* and their configuration.
- The *HistoryRead* and *HistoryUpdate* Services of the *Attribute Service Set* to read and update history of *HistoricalNodes*.

### 5.2 Historical Nodes StatusCodes

#### 5.2.1 Overview

This section defines additional codes and rules that apply to the *StatusCode* when used for *HistoricalNodes*.

The general structure of the *StatusCode* is specified in [UA Part 4]. It includes a set of common operational result codes which also apply to historical data and/or events.

#### 5.2.2 Operation level result codes

In OPC UA Historical Access the *StatusCode* is used to indicate the conditions under which a *Value* or *Event* was stored, and thereby can be used as an indicator it's usability. Due to the nature of historical data and/or events, additional information beyond the basic quality and call result code needs to be conveyed to the client. For example, whether the value is actually stored in the data repository, was the result interpolated, were all data inputs to a calculation of good quality, etc.

In the following, Table 11 contains codes with *Bad* severity indicating a failure; Table 12 contains codes with *Uncertain* severity indicating that the value has been retrieved under sub-normal conditions. Table 13 contains *Good* (success) codes. It is Important to note, that these are the codes that are specific for OPC UA Historical Access and supplement the codes that apply to all types of data and are therefore defined in [UA Part 4] and [UA Part 8].

**Table 11 – Bad operation level result codes**

Symbolic Id	Description
Bad_NoData	No data exists for the requested time range or event filter
Bad_NoBound	No data found to provide upper or lower bound value.
Bad_DataLost	Data is missing due to collection started / stopped / lost.
Bad_EntryExists	The data or event was not successfully inserted because a matching entry exists.
Bad_NoEntryExists	The data or event was not successfully updated because no matching entry exists.
Bad_TimestampNotSupported	The client requested history using a timestamp format the server does not support (i.e requested ServerTimestamp when server only supports SourceTimestamp)

**Table 12 – Uncertain operation level result codes**

Symbolic Id	Description
Uncertain_SubNormal	The value is derived from multiple values and has less than the required number of <i>Good</i> values.

**Table 13 – Good operation level result codes**

Symbolic Id	Description
Good_NoData	No data exists for the requested time range or event filter.
Good_MoreData	There is more data to be returned than could be returned in a single request.
Good_EntryInserted	The data or event was successfully inserted into the historical database

Good_EntryReplaced	The data or event field was successfully replaced in the historical database
--------------------	--

### 5.2.3 Historian Information Bits

These bits are set only when reading historical data of *HistoricalDataNodes*. They indicate where the data value came from and provide information that affects how the client uses the data value. Table 14 lists the bit settings which indicate the data location (i.e. is the value stored in the underlying data repository, or is the value the result of data aggregation). These bits are mutually exclusive.

**Table 14 – Data Location**

StatusCode	Description
Raw	A raw data value.
Calculated	A data value which was calculated.
Interpolated	A data value which was interpolated.

In the case where interpolated data is requested, and there is an actual raw value for that timestamp, the server should set the 'Raw' bit in the *StatusCode* of that value.

Table 15 lists the bit settings which indicate additional important information about the data values returned.

**Table 15 – Additional Information**

StatusCode	Description
Partial	A data value which was calculated with an incomplete interval.
Extra Data	A raw data value that hides other data at the same timestamp.
Multiple Values	Multiple values match the aggregate criteria (i.e. multiple minimum values or multiple worst quality at different timestamps within the same interval)

The conditions under which these information bits are set depend on how the historical data has been requested and state of the underlying data repository.

### 5.2.4 Semantics changed

The *StatusCode* in addition contains an informational bit called *Semantics Changed*. (See [UA Part 4])

UA Servers that implement OPC UA Historical Access should not set this Bit, rather propagate the *StatusCode* which has been stored in the data repository. Clients should be aware that the returned data values may have this bit set.

## 5.3 HistoryReadDetails parameters

### 5.3.1 Overview

The *HistoryRead* service defined in [UA Part 4] can perform several different functions. The *historyReadDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See [UA Part 4] for the definition of *Extensible Parameter*. Table 16 lists the valid values for the *parameterTypeId* parameter which specifies which function the *HistoryRead* service will perform, and what structure will be contained in the *parameterData* field .

**Table 16 – HistoryReadDetails parameterTypeId Values**

Name	Value	Description	parameterData Structure
EVENTS	1	This parameter selects a set of events from the history database by specifying a filter and a time domain for one or more <i>Objects</i> or <i>Views</i> .	ReadEventDetails (See Clause 5.3.2)

		This parameter is only valid for <i>Objects</i> that have the <i>EventNotifier</i> attribute set to TRUE (See [UA Part 3]).	
RAW	2	This parameter selects a set of values from the history database by specifying a time domain for one or more <i>Variables</i>	ReadRawModifiedDetails (See Clause 5.3.3)
MODIFIED	3	This parameter selects a set of modified values from the history database by specifying a time domain for one or more <i>Variables</i> . A modified value is a value that has been replaced by another value at the same timestamp in the history database. If there are multiple replaced values the server must return all of them.  The server indicates that modified data exists by setting the <i>ExtraData</i> bit in the <i>StatusCodes</i> associated with a <i>DataValues</i> returned during a RAW, PROCESSED, or ATTIME request.	ReadRawModifiedDetails (See Clause 5.3.3)
PROCESSED	4	This parameter selects a set of aggregate values from the history database by specifying a time domain for one or more <i>Variables</i> . This function is intended to provide Values calculated with respect to the resample interval. For example, this function can provide hourly statistics such as Maximum, Minimum, Average, etc. for each item during the specified time domain when resample interval is 1 hour.	ReadProcessedDetails (See Clause 5.3.4)
ATTIME	5	This parameter selects a set of raw or interpolated values from the history database by specifying a series of timestamps for one or more <i>Variables</i> . This function is intended to provide values to correlate with other values with a known timestamp. For example, the values of sensors when lab samples were collected.	ReadAtTimeDetails (See Clause 5.3.5)

### 5.3.2 ReadEventDetails structure

Table 17 defines the *ReadEventDetails* structure. Two of the three parameters, numValuesPerNode, startTime, and endTime must be specified.

**Table 17 – ReadEventDetails**

Name	Type	Description
ReadEventDetails	structure	Specifies the details used to perform an event history read.
numValuesPerNode	Counter	The maximum number of values returned for any node over the time range. If only one time is specified, the time range must extend to return this number of values. The default value of 0 indicates that there is no maximum.
startTime	UtcTime	Beginning of period to read. The default value of <i>DateTime.Min</i> indicates that there is no start time.
endTime	UtcTime	End of period to read. The default value of <i>DateTime.Min</i> indicates that there is no end time.
filter	EventFilter	A filter used by the Server to determine which HistoricalEventNode should be included. This parameter must be specified and at least one <i>EventFieldId</i> is required. The <i>EventFilter</i> parameter type is an extensible parameter type. It is defined and used in the same manner as defined for monitored data items which are specified [UA Part 4]. This filter also specifies the <i>EventFields</i> that are to be returned as part of the request.

The EVENTS parameter reads the events from the history database for the specified time domain for one or more HistoricalEventNodes. The events are filtered based on the filter structure provided. This filter includes the *eventFields* that are to be returned. For a complete description of filter refer to [UA Part 4], in particular *MonitoredItems*.

The time domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data will be returned in reverse order, with later data coming first, as if time were flowing backward. If all three are specified, the call shall return up to numValuesPerNode results going from startTime to endTime, in either ascending or descending order depending on the relative values of startTime and endTime. If numValuesPerNode is 0, then all the values in the range are returned. The default value is used to indicate when startTime, endTime or numValuesPerNode is not specified.

It is specifically allowed for the startTime and the endTime to be identical. This allows the client to request the event at a single instance in time. When the startTime and endTime are identical, time is presumed to be flowing forward. If no data exists at the time specified then the server must return the *Good\_NoData StatusCode*.

If more than numValuesPerNode results exist within that time range, the *StatusCodes* returned for that Variable must be *Good\_MoreData*, and the *continuationPoint* must be returned. When

*Good\_MoreData* is returned, clients wanting the next numValuesPerNode values should call HistoryRead again with the continuationPoint.

For an interval in which no data exists, the corresponding StatusCode shall be *Good\_NoData*.

The *filter* parameter is used to determine which historical events and their corresponding fields are returned. It is possible that the fields of an *EventType* are available for real time updating, but not available from the historian. In this case a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* must be *Bad\_NoData*.

### 5.3.3 ReadRawModifiedDetails structure

#### 5.3.3.1 ReadRawModifiedDetails structure Overview

Table 18 defines the *ReadRawDetails* structure. Two of the three parameters, numValuesPerNode, startTime, and endTime must be specified.

**Table 18 – ReadRawModifiedDetails**

Name	Type	Description
ReadRawModifiedDetails	Structure	Specifies the details used to perform a “raw” or “modified” history read.
isReadModified	Boolean	TRUE for MODIFIED, FALSE for RAW. Default value is FALSE.
startTime	UtcTime	Beginning of period to read. Set to default value of <i>DateTime.Min</i> if no specific start time is specified.
endTime	UtcTime	End of period to read. Set to default value of <i>DateTime.Min</i> if no specific end time is specified.
numValuesPerNode	Counter	The maximum number of values returned for any node over the time range. If only one time is specified, the time range must extend to return this number of values. The default value 0 indicates that there is no maximum.
returnBounds	Boolean	A boolean parameter with the following values: TRUE bounding values should be returned FALSE all other cases.

#### 5.3.3.2 RAW usage

When this structure is used for reading Raw Values (isReadModified is set to False); it reads the values, qualities, and timestamps from the history database for the specified time domain for one or more *HistoricalDataNodes*. This parameter is intended for use by clients wanting the actual data saved within the historian. The actual data may be compressed or may be all data collected for the item depending on the historian and the storage rules invoked when the item values were saved. When returnBounds is TRUE, the bounding values for the time domain are returned. The optional bounding values are provided to allow clients to interpolate values for the start and end times when trending the actual data on a display.

The time domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data will be returned in reverse order, with later data coming first, as if time were flowing backward. If all three are specified, the call shall return up to numValuesPerNode results going from startTime to endTime, in either ascending or descending order depending on the relative values of startTime and endTime. If numValuesPerNode is 0, then all the values in the range are returned. A default value of *DateTime.Min* is used to indicate when startTime or endTime is not specified.

It is specifically allowed for the startTime and the endTime to be identical. This allows the client to request just one value. When the startTime and endTime are identical, time is presumed to be flowing forward. It is specifically not allowed for the server to return an *Bad\_InvalidArgument StatusCode* if the requested time domain is outside of the server's range. Such a case shall be treated as an interval in which no data exists.

If more than numValuesPerNode results exist within that time range, the *StatusCode* entry for that variable shall be *Good\_MoreData*, and the continuationPoint will be set. When *Good\_MoreData* is returned, clients wanting the next numValuesPerNode values should call ReadRaw again with the continuationPoint set.

If bounding values are requested and a non-zero numValuesPerNode was specified, any bounding values returned are included in the numValuesPerNode count. If numValuesPerNode is 1, then only the start bound is returned (the End bound if reverse order is needed). If numValuesPerNode is 2, the start bound and the first data point is returned (the End bound if reverse order is needed).

When bounding values are requested and no bounding value is found, the corresponding *StatusCode* entry will be set to *Bad\_NoBound*, a timestamp equal to the start or end time, as appropriate, and a value of Null. How far back or forward to look in history for bounding values is server dependent.

For an interval in which no data exists, if bounding values are not requested, the corresponding *StatusCode* must be *Good\_NoData*. If bounding values are requested and one or both exist, the result code returned is Success and the bounding value(s) are returned.

For cases where there are multiple values for a given timestamp, all but the most recent are considered to be Modified values and the server must return the most recent value. If the server returns a value which hides other values at a timestamp then it must set the *ExtraData* bit in the *StatusCode* associated with that value.

### 5.3.3.3 MODIFIED usage

When this structure is used for reading Modified Values (isReadModified is set to true); it reads the values, qualities, timestamps, user identifier, and timestamp of the modification from the history database for the specified time domain for one or more *HistoricalDataNodes*.

The purpose of this function is to read values from history that have been modified/replaced. If ReadRaw, ReadProcessed, or ReadAtTime has returned a *StatusCode* of with the *ExtraData* bit set then there are values which have been superseded in the history database. This parameter allows clients to read those values which were superseded. Only values that have been modified/replaced or deleted are read by this function

The domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data shall be returned in reverse order, with later data coming first. If all three are specified, the call shall return up to numValuesPerNode results going from StartTime to EndTime, in either ascending or descending order depending on the relative values of StartTime and EndTime. If more than numValuesPerNode results exist within that time range, the *StatusCode* entry for that variable shall be *Good\_MoreData*. If numValuesPerNode is 0, then all the values in the range are returned.

If a value has been modified multiple times, all values for the time are returned. This means that a timestamp can appear in the array more than once. The order of the returned values with the same timestamp should be from most recent to oldest modified value, if startTime is less than or equal to endTime. If endTime is less than startTime, the order of the returned values will be from oldest modified value to most recent. It is server dependent whether multiple modifications are kept or only the most recent.

### 5.3.4 ReadProcessedDetails structure

Table 19 defines the structure of the ReadProcessedDetails structure.

**Table 19 – ReadProcessedDetails**

Name	Type	Description
ReadProcessedDetails	structure	Specifies the details used to perform a “processed” history read
startTime	UtcTime	Beginning of period to read.
endTime	UtcTime	End of period to read.
resampleInterval	Duration	Interval between returned aggregate values. The value 0 indicates that there is no interval defined.
aggregateType	NodeId	The NodeId of the HistoryAggregate object that indicates the aggregate to be used when retrieving processed history. See for details.

See Table 8 for possible *NodeId* values for the HistoryAggregateType parameter.

The PROCESSED function computes aggregate values, qualities, and timestamps from data in the history database for the specified time domain for one or more *HistoricalDataNodes*. The time domain is divided into subintervals of duration resampleInterval. The specified aggregateType is calculated for each subinterval beginning with startTime by using the data within the next resampleInterval.

For example, this function can provide hourly statistics such as Maximum, Minimum, Average, etc. for each item during the specified time domain when resampleInterval is 1 hour.

The domain of the request is defined by startTime, endTime, and resampleInterval. All three must be specified. If endTime is less than startTime, the data shall be returned in reverse order, with later data coming first. If startTime and endTime are the same, the server shall return *Bad\_InvalidArgument*, as there is no meaningful way to interpret such a case.

The values used in computing the aggregate for each subinterval shall include any value that falls exactly on the timestamp beginning the subinterval, but shall not include any value that falls directly on the timestamp ending the subinterval. Thus, each value shall be included only once in the calculation. If the time domain is in reverse order, we consider the later timestamp to be the one beginning the subinterval, and the earlier timestamp to be the one ending it. Note that this means that simply swapping the start and end times will not result in getting the same values back in reverse order, as the subintervals being requested in the two cases are not the same.

If the last subinterval computed is not a complete subinterval (the time domain of the request is not evenly divisible by the resample interval), the last aggregate returned shall be based upon that incomplete subinterval, and the corresponding *StatusCode* shall be *PARTIAL*.

For MinimumActualTime and MaximumActualTime, if more than one instance of the value exists within a subinterval, which instance (time stamp) of the value returned is server dependent. In any case, the server must set the *MultipleValue* bit in the *StatusCode* to let the caller know that there are other timestamps with that value.

If resampleInterval is 0, the server must create one aggregate value for the entire time range. This allows aggregates over large periods of time. A value with a timestamp equal to endTime will be excluded from that aggregate, just as it would be excluded from a subinterval with that ending time.

The timestamp returned with the aggregate must be the time at the beginning of the interval, except where the aggregate specifies a different value.

For all Aggregates that do not specify otherwise the following rule applies to determining the status associated with a given computed value. If the percentage of the values used in computing the aggregate value that have *Good* quality meets or exceeds the PercentDataGood parameter, the *StatusCode* of the aggregate must be *Good*. If the percentage of the values used in computing the aggregate value that have *Bad* quality meets or exceeds the PercentDataBad parameter, the *StatusCode* of the aggregate must be *Bad*. Otherwise the *StatusCode* of the aggregate must be *Uncertain\_SubNormal*.

If no data exists for a given *HistoricalDataNode* in any subinterval in the time domain, the server shall return *Bad\_NoData* in the *StatusCode* for that *HistoricalDataNode*.

If data does exist in at least one subinterval for that *HistoricalDataNode*, the server shall return a timestamp, *StatusCode*, and value for each subinterval in the time domain.

### 5.3.5 ReadAtTimeDetails structure

Table 20 defines the ReadAtTimeDetails structure.

**Table 20 – ReadAtTimeDetails**

Name	Type	Description
ReadAtTimeDetails	Structure	Specifies the details used to perform an “at time” history read
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be read.

The ATTIME parameter reads the values and qualities from the history database for the specified timestamps for one or more *HistoricalDataNodes*. This function is intended to provide values to correlate with other values with a known timestamp. For example, a client may need to read the values of sensors when lab samples were collected.

The order of the values and qualities returned shall match the order of the time stamps supplied in the request.

When no value exists for a specified timestamp, a value shall be interpolated from the surrounding values to represent the value at the specified timestamp. The interpolation will follow the same rules as the standard Interpolated aggregate as outlined in Clause 5.6.3.5

If a value is found for the specified timestamp, the server will set the *StatusCode InfoBits* to be *Raw*. If the value is interpolated from the surrounding values, the server will set the *StatusCode InfoBits* to be *Interpolated*.

## 5.4 HistoryData parameters

### 5.4.1 Overview

The *HistoryRead* service returns different types of data depending on whether the request asked for the value attribute of a node or the history events of a node. The *historyData* is an *Extensible Parameter* whose structure depends on the functions to perform for the *historyReadDetails* parameter. See [UA Part 4] for details on *Extensible Parameters*.

### 5.4.2 HistoryData type

Table 21 defines the structure of the *HistoryData* used for the data to return in a *HistoryRead*.

**Table 21 – HistoryData Details**

Name	Type	Description
dataValue[]	DataValue	An array of values of history data for the node. The size of the array depends on the requested data parameters.

### 5.4.3 HistoryEvent type

**Table 22 – HistoryEvent Details**

Name	Type	Description
historyEvent[]	EventNotification	An array of Event <i>Notification</i> data. The size of the array depends on the requested data parameters.

## 5.5 HistoryUpdateDetails parameter

### 5.5.1 Overview

The *HistoryUpdate* service defined in [UA Part 4] can perform several different functions. The *historyUpdateDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See [UA Part 4] for the definition of *Extensible Parameter*. Table 23 lists the valid values for the *parameterTypeid* parameter which specifies which function the *HistoryUpdate* service will perform, and what structure will be contained in the *parameterData* field .

**Table 23 – HistoryUpdateDetails parameterTypeId Values**

Name	Value	Description	parameterData Structure
INSERTDATA	1	This function inserts new values into the history database at the specified timestamps for one or more HistoricalDataNodes. The variable's value is represented by a composite value defined by the <i>WithValue</i> data type.	UpdateDataDetails (See Clause 5.5.2)
REPLACEDATA	2	This function replaces existing values into the history database at the specified timestamps for one or more HistoricalDataNodes. . The variable's value is represented by a composite value defined by the <i>WithValue</i> data type.	UpdateDataDetails (See Clause 5.5.2)
UPDATEDATA	3	This function inserts or replaces values into the history database at the specified timestamps for one or more HistoricalDataNodes. . The variable's value is represented by a composite value defined by the <i>WithValue</i> data type.	UpdateDataDetails (See Clause 5.5.2)
INSERTEVENT	4	This function inserts new events into the history database for one or more HistoricalEventNodes.	UpdateEventDetails (See Clause 5.5.3)
REPLACEEVENT	5	This function replaces values of fields in existing events into the history database for one or more HistoricalEventNodes.	UpdateEventDetails (See Clause 5.5.3)
UPDATEEVENT	6	This function inserts new events or replaces values of fields in existing events into the history database for one or more HistoricalEventNodes.	UpdateEventDetails (See Clause 5.5.3)
DELETERAW	7	This function deletes all values from the history database for the specified time domain for one or more HistoricalDataNodes.	DeleteDataDetails (See Clause 5.5.4)
DELETETMODIFIED	8	Some historians may store multiple values at the same Timestamp. This function will delete specified values and qualities for the specified timestamp for one or more HistoricalDataNodes. .	DeleteDataDetails (See Clause 5.5.4)
DELETEATTIME	9	This function deletes all values in the history database for the specified timestamps for one or more HistoricalDataNodes. .	DeleteAtTimeDetails (See Clause 5.5.5)
DELETEEVENT	10	This function deletes events from the history database for the specified filter for one or more HistoricalEventNodes.	DeleteEventDetails (See Clause 5.5.6)

The HistoryUpdate service is used to update or delete both *DataValues* and *Events*. For simplicity the term “entry” will be used to mean either *WithValue* or *Event* depending on the context in which it is used. Auditing requirements for History services is described in [UA Part 4]. This description assumes the user issuing the request and the server that is processing the request supports Updating entries. See [UA Part 3] for a description of *Attributes* that expose the support of Historical Updates.

## 5.5.2 UpdataDataDetails structure

### 5.5.2.1 UpdataDataDetails structure Overview

Table 24 defines the *UpdateDataDetails* structure.

**Table 24 – UpdateDataDetails**

Name	Type	Description
UpdateDataDetails	Structure	The details for insert, replace, and insert/replace history updates.
performInsert	Boolean	TRUE means perform INSERT, FALSE means do not perform INSERT. Default is FALSE.
performReplace	Boolean	TRUE means perform REPLACE, FALSE means do not perform REPLACE. Default is FALSE.
nodeId	NodeId	Node id of the variable to be updated.
updateValue	historyData	New value to be inserted or replaced

### 5.5.2.2 INSERTDATA usage

The *INSERTDATA* parameter inserts entries into the history database at the specified timestamps for one or more *HistoricalDataNodes*. If an entry exists at the specified timestamp, the new entry shall not be inserted; instead the *StatusCode* shall indicate *Bad\_EntryExists*.

This function is intended to insert new entries at the specified timestamps; e.g., the insertion of lab data to reflect the time of data collection.

#### 5.5.2.3 REPLACEDATA usage

The REPLACEDATA parameter replaces entries in the history database at the specified timestamps for one or more *HistoricalDataNodes*. If no entry exists at the specified timestamp, the new entry shall not be inserted; otherwise the *StatusCode* shall indicate *Bad\_NoEntryExists*.

This function is intended to replace existing entries at the specified timestamp; e.g., correct lab data that was improperly processed, but inserted into the history database.

#### 5.5.2.4 UPDATEDATA usage

The UPDATEDATA parameter inserts or replaces entries in the history database for the specified timestamps for one or more *HistoricalDataNodes*. If the item has a entry at the specified timestamp, the new entry will replace the old one. If there is no entry at that timestamp, the function will insert the new data.

This function is intended to unconditionally insert/replace values and qualities; e.g., correction of values for bad sensors.

*Good* as a *StatusCode* for an individual entry is allowed when the server is unable to say whether there was already a value at that timestamp. If the server can determine whether the new entry replaces a entry that was already there, it should use *Good\_EntryInserted* or *Good\_EntryReplaced* to return that information.

### 5.5.3 UpdateEventDetails structure

Table 24 defines the UpdateEventDetails structure.

**Table 25 – UpdateEventDetails**

Name	Type	Description
UpdateEventDetails	Structure	The details for insert, replace, and insert/replace history event updates.
performInsert	Boolean	TRUE means perform INSERT, FALSE means do not perform INSERT. Default is FALSE.
performReplace	Boolean	TRUE means perform REPLACE, FALSE means do not perform REPLACE. Default is FALSE.
nodeId	NodeId	Node id of the Node to be updated.
filter	EventFilter	If the history of <i>Notification</i> conforms to the <i>EventFilter</i> , the history of the <i>Notification</i> is updated.
eventData	EventNotification	Event <i>Notification</i> data to be inserted or updated.

#### 5.5.3.1 INSERTEVENT usage

The INSERTEVENT parameter inserts entries into the event history database for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* must specify the *EventId* Property. If any entry exists matching the specified filter, the new entry shall not be inserted; instead *StatusCode* shall indicate *Bad\_EntryExists*.

If the new entry is incomplete or not correctly specified in the *EventNotification*, the server may return a *StatusCode* of *Bad\_InvalidArgument*.

This function is intended to insert new entries; e.g., backfilling of historical events.

#### 5.5.3.2 REPLACEEVENT usage

The REPLACEEVENT parameter replaces entries in the event history database for the specified filter for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* must specify the *EventId* Property. If no entry exists matching the specified filter, the new entry shall not be inserted; otherwise the *StatusCode* shall indicate *Bad\_NoEntryExists*.

If the new entry is incomplete or not correctly specified in the *EventNotification*, the server may return a *StatusCode* of *Bad\_InvalidArgument*.

This function is intended to replace fields in existing event entries; e.g., correct event data that contained incorrect data due to a bad sensor.

### 5.5.3.3 UPDATEEVENT usage

The UPDATEEVENT parameter inserts or replaces entries in the event history database for the specified filter for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* must specify fields to uniquely identify the event (i.e. EventId or combination of identifying fields). If any entry exists matching the specified filter, the new event data will replace the existing data. If no matching entry is found, the function will insert the new event.

This function is intended to unconditionally insert/replace events; e.g., synchronizing a backup event database.

*Good* as a *StatusCode* for an individual entry is allowed when the server is unable to say whether there was already an existing value. If the server can determine whether the new entry replaces an existing, it should use *Good\_EntryInserted* or *Good\_EntryReplaced* to return that information.

### 5.5.4 DeleteRawModifiedDetails structure

Table 26 defines the DeleteRawModifiedDetails structure.

**Table 26 – DeleteRawModifiedDetails**

Name	Type	Description
DeleteRawModifiedDetails	structure	The details for delete raw and delete modified history updates.
isDeleteModified	Boolean	TRUE for MODIFIED, FALSE for RAW. Default value is FALSE.
nodeId	NodeId	Node id of the variable for which history values are to be deleted.
startTime	UtcTime	beginning of period to be deleted
endTime	UtcTime	end of period to be deleted

The DELETERAW parameter deletes all raw entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

The DELETEMODIFIED parameter deletes all modified entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

These functions are intended to be used to delete data that has been accidentally entered into the history database; e.g., deletion of data from a source with incorrect timestamps.

If no data is found in the time range for a particular *HistoricalDataNode*, the *StatusCode* for that item is *Bad\_NoData*.

### 5.5.5 DeleteAtTimeDetails structure

Table 27 defines the structure of the DeleteAtTimeDetails structure.

**Table 27 – DeleteAtTimeDetails**

Name	Type	Description
DeleteAtTimeDetails	Structure	The details for delete raw history updates
nodeId	NodeId	Node id of the variable for which history values are to be deleted.
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be deleted.

The DELETEATTIME parameter deletes all entries in the history database for the specified timestamps for one or more *HistoricalDataNodes*.

This parameter is intended to be used to delete specific data from the history database; e.g., lab data that is incorrect and cannot be correctly reproduced.

### 5.5.6 DeleteEventDetails structure

Table 27 defines the structure of the DeleteEventDetails structure.

**Table 28 – DeleteEventDetails**

Name	Type	Description
DeleteEventDetails	structure	The details for delete raw and delete modified history updates.
nodeId	NodeId	Node id of the variable for which history values are to be deleted.
eventId[]	ByteString	An array of <i>EventIds</i> to identify which events are to be deleted.

The DELETEEVENT parameter deletes all event entries from the history database matching the *EventId* for one or more *HistoricalEventNodes*.

If no events are found that match the specified filter for a *HistoricalEventNode*, the *StatusCodes* for that *Node* is *Bad\_NoData*.

## 5.6 Aggregate Details

### 5.6.1 General

The purpose of this section is to detail the requirements and behavior for OPC UA Server supporting Historical Access *Aggregates*. The intent is to standardize the OPC UA Server supporting Historical Access *Aggregates* such that OPC UA Server supporting Historical Access clients can reliably predict the results of an *Aggregate* computation and understand its meaning. If users require custom functionality in the *Aggregates*, those *Aggregates* should be written as custom vendor defined *Aggregates*.

The standard *Aggregates* must be as consistent as possible, meaning that each *Aggregate*'s behavior must be similar to every other *Aggregate*'s behavior where input parameters, raw data, and boundary conditions are similar. Where possible, the *Aggregates* should deal with input and preconditions in a similar manner.

This section is divided up into two parts. The first sub section deals with *Aggregate* characteristics and behavior that are common to all *Aggregates*. The remaining sub sections deal with the characteristics and behavior of *Aggregates* that are aggregate-specific.

### 5.6.2 Common characteristics

#### 5.6.2.1 Description

This subsection deals with aggregate characteristics and behavior that are common to all aggregates.

#### 5.6.2.2 Generating intervals

To read aggregates, OPC clients must specify three time parameters:

- start time (Start)
- end time (End)
- resample interval (Int)

The OPC server must use these three parameters to generate a sequence of time intervals and then calculate an aggregate for each interval. This section specifies, given the three parameters, which time intervals are generated. Table 29 outlines information on the intervals for each Start and End time combination. Range is defined to be  $|End - Start|$ .

All interval aggregates return a timestamp of the start of the interval unless otherwise noted for the particular aggregate.

**Table 29 – History Aggregate Interval Information**

Start/End Time	Resample Interval	Resulting Intervals
$Start = End$	$Int = \text{Anything}$	No intervals. Returns a <code>Bad_InvalidArgument StatusCode</code> , regardless of whether there is data at the specified time or not.
$Start < End$	$Int = 0 \text{ or } Int \geq Range$	One interval, starting at $Start$ and ending at $End$ . Includes $Start$ , excludes $End$ , i.e., $[Start, End)$ .
$Start < End$	$Int \neq 0, Int < Range, Int \text{ divides } Range \text{ evenly.}$	$\lceil Range/Int \rceil$ intervals. Intervals are $[Start, Start + Int), [Start + Int, Start + 2 * Int), \dots, [End - Int, End)$ .
$Start < End$	$Int \neq 0, Int < Range, Int \text{ does not divide } Range \text{ evenly.}$	$\lceil Range/Int \rceil$ intervals. Intervals are $[Start, Start + Int), [Start + Int, Start + 2 * Int), \dots, [Start + (\lfloor Range/Int \rfloor - 1) * Int, Start + \lfloor Range/Int \rfloor * Int), [Start + \lfloor Range/Int \rfloor * Int, End]$ . In other words, the last interval contains the “rest” that remains in the range after taking away $\lfloor Range/Int \rfloor$ intervals of size $Int$ .
$Start > End$	$Int = 0 \text{ or } Int \geq Range$	One interval, starting at $Start$ and ending at $End$ . Includes $Start$ , excludes $End$ , i.e., $[End, Start)$ .
$Start > End$	$Int \neq 0, Int < Range, Int \text{ divides } Range \text{ evenly.}$	$\lceil Range/Int \rceil$ intervals. Intervals are $[Start - Int, Start], [Start - 2 * Int, Start - Int), \dots, [End, End + Int]$ .
$Start > End$	$Int \neq 0, Int < Range, Int \text{ does not divide } Range \text{ evenly.}$	$\lceil Range/Int \rceil$ intervals. Intervals are $[Start - Int, Start], [Start - 2 * Int, Start - Int), \dots, [Start - \lfloor Range/Int \rfloor * Int, Start - (\lfloor Range/Int \rfloor - 1) * Int], [End, Start - \lfloor Range/Int \rfloor * Int]$ . In other words, the last interval contains the “rest” that remains in the range after taking away $\lfloor Range/Int \rfloor$ intervals of size $Int$ starting at $Start$ .

### 5.6.2.3 Data types

Table 8 outlines the valid data types for each aggregate. Some aggregates are intended for numeric data types – i.e. integers or real/floating point numbers. Dates, strings, arrays, etc. are not supported. Other aggregates are intended for digital data types – i.e. Boolean or enumerations. In addition some aggregates may return results with a different datatype than those used to calculate the aggregate. Table 8 also outlines the default data type returned for each aggregate.

**Table 30 – Standard History Aggregate Data Type Information**

BrowseName	Valid Data Type	Default Result Data Type
	<b>Interpolation Aggregate</b>	
Interpolative	Numeric	Double
	<b>Data Averaging Aggregates</b>	
Average	Numeric	Double
TimeAverage	Numeric	Double
Total	Numeric	Double
TotalizeAverage	Numeric	Double
	<b>Data Variation Aggregates</b>	
Minimum	Numeric	Raw data type
Maximum	Numeric	Raw data type
MinimumActualTime	Numeric	Raw data type
MaximumActualTime	Numeric	Raw data type
Range	Numeric	Raw data type
	<b>Counting Aggregates</b>	
AnnotationCount	All	Integer
Count	All	Integer
DurationInState0	Boolean	Duration
DurationInState1	Boolean	Duration
NumberOfTransitions	Boolean	Integer
	<b>Time Aggregates</b>	
Start	All	Raw data type
End	All	Raw data type
Delta	Numeric	Raw data type
	<b>Data Quality Aggregates</b>	
DurationGood	All	Duration
DurationBad	All	Duration
PercentGood	All	Double
PercentBad	All	Double
WorstQuality	Numeric	StatusCode

#### 5.6.2.4 StatusCode calculation

For Aggregate values, the *StatusCode* for each returned aggregate shall be *Good*, if the *StatusCode* for ALL values used in the aggregate was *Good*.

If the *StatusCode* of ANY value used in computing the aggregate was not *Good*, then the server must use the *TreatUncertainAsBad*, *PercentDataBad* and *PercentDataGood* parameter (see Clause 4.8.2) settings to determine the *StatusCode* of the resulting aggregate for the interval. Some aggregates may explicitly define their own method of determining quality.

If the percentage of *Good* values in an interval is greater than or equal to the *PercentDataGood*, the aggregate is considered *Good*.

If the percentage of *Bad* values in an interval is greater than or equal to the *PercentDataBad*, the aggregate is considered *Bad*.

Since a value can be either *Good* or *Bad* only (*Uncertain* is defined as *Good* or *Bad* as per *TreatUncertainAsBad* setting), percentage good = 100 – percentage bad. If a percentage good (X) is in the following range Percentage bad < X < Percentage Good then the quality of the aggregate is *Uncertain\_SubNormal*.

### 5.6.3 Aggregate specific characteristics

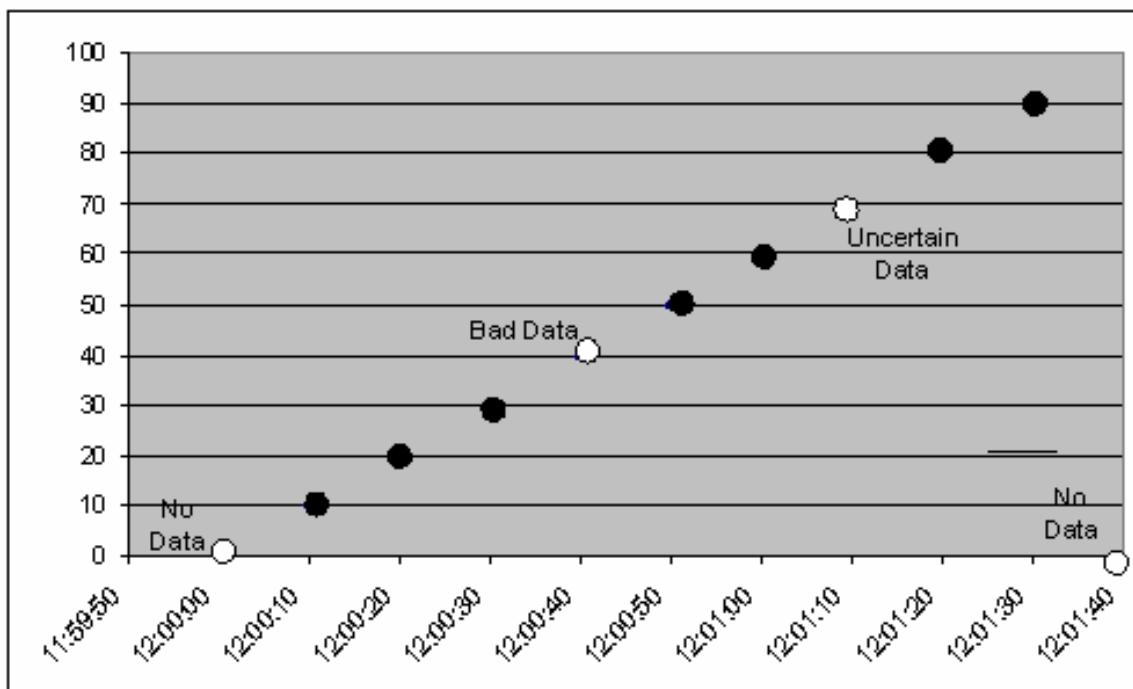
#### 5.6.3.1 Description

This sub section deals with aggregate specific characteristics and behavior that is specific to a particular aggregate.

#### 5.6.3.2 Example aggregate data – Historian 1

For the purposes of examples consider a source historian with the following data:

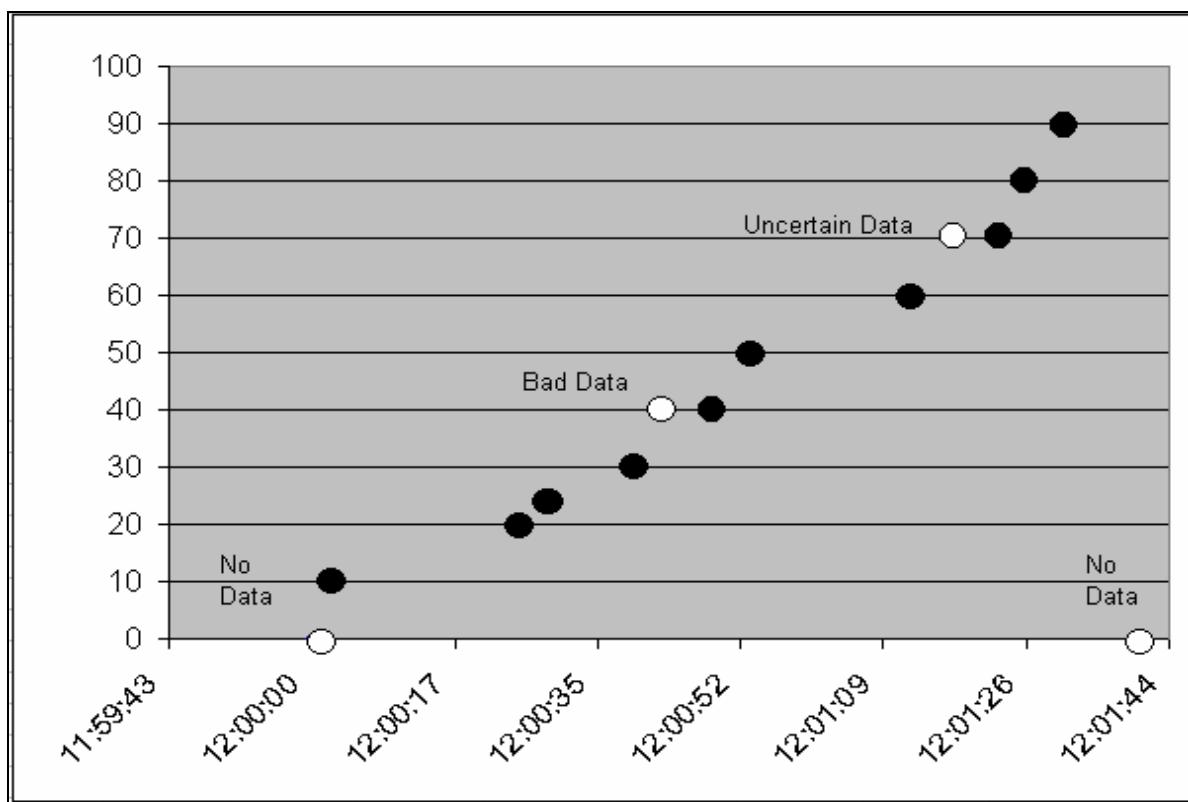
Timestamp	Value	StatusCode	Notes
Jan-01-2002 12:00:00	-	Bad_NoData	First archive entry, Point Created
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:20	20	Raw, Good	
Jan-01-2002 12:00:30	30	Raw, Good	
Jan-01-2002 12:00:40	40	Raw, Bad	Scan failed, Bad data entered
Jan-01-2002 12:00:50	50	Raw, Good	
Jan-01-2002 12:01:00	60	Raw, Good	
Jan-01-2002 12:01:10	70	Raw, Uncertain	Value is flagged as questionable
Jan-01-2002 12:01:20	80	Raw, Good	
Jan-01-2002 12:01:30	90	Raw, Good	
	NULL	No Data	No more entries, awaiting next scan.



#### 5.6.3.3 Example aggregate data – Historian 2

The following data is also included in a separate column to illustrate non-periodic data

Timestamp	Value	StatusCode	Notes
Jan-01-2002 12:00:00	-	Bad_NoData	First archive entry, Point Created
Jan-01-2002 12:00:02	10	Raw_Good	
Jan-01-2002 12:00:25	20	Raw_Good	
Jan-01-2002 12:00:28	25	Raw_Good	
Jan-01-2002 12:00:39	30	Raw_Good	
Jan-01-2002 12:00:42	40	Raw_Bad	Bad quality data received, Bad data entered
Jan-01-2002 12:00:48	40	Raw_Good	Received good StatusCode value
Jan-01-2002 12:00:52	50	Raw_Good	
Jan-01-2002 12:01:12	60	Raw_Good	
Jan-01-2002 12:01:17	70	Raw_Uncertain	Value is flagged as questionable
Jan-01-2002 12:01:23	70	Raw_Good	
Jan-01-2002 12:01:26	80	Raw_Good	
Jan-01-2002 12:01:30	90	Raw_Good	
	-	No Data	No more entries, awaiting next Value.



#### 5.6.3.4 Example Conditions

For the purposes of all examples,

##### Historian 1

1. *TreatUncertainAsBad* = False. Therefore *Uncertain* values are included in aggregate call.
2. *Stepped* attribute = False. Therefore Linear interpolation is used between data points.
3. *SteppedInterpolationMode* = True. Therefore Stepped extrapolation is used at end boundary conditions

## Historian 2

1. *TreatUncertainAsBad* = True. Therefore *Uncertain* values are treated as *Bad*, and not included in the aggregate call.
2. *Stepped* attribute = False. Therefore Linear interpolation is used between data points.
3. *SteppedInterpolationMode* = True, Therefore Stepped extrapolation is used at end boundary conditions

### 5.6.3.5 Interpolative

#### 5.6.3.5.1 Description

In order for the interpolative aggregate to return meaningful data, there must be good values at the boundary conditions. For the purposes of discussion we will use the terms good and non-good. As discussed in the *StatusCodes* section (See Clause 5.6.2.4), what is represented by non-good is Server dependant. For some Servers non-good represents only *Bad* data, for others it represents *Bad* and *Uncertain* data depending on the *TreatUncertainAsBad* setting.

When determining boundary conditions, the following rules must be followed:

- o If the value at the requested time is non-good, the aggregate looks for good bounding data within the intervals preceding and following the requested time. (In the case of Stepped interpolation a bounding value following requested time is not required). If no good data is found within the respective intervals, there is no bound and the aggregate must return *Bad\_NoData*. If no data exists within the respective intervals, the aggregate will continue expanding the respective search intervals up to a maximum equal to the requested time Range. If no data exists within the respective Ranges, there is no bound and the aggregate must return *Bad\_NoData*.
- o The method of interpolation, either interpolated (sloped Lines between point) or as Stepped (vertically-connected horizontal lines between points) is determined by the *Stepped* attribute. See Clause 4.7.1.4
- o If there is no end bound (i.e. future time), the value should be extrapolated forward in time from the previous good value. The method of extrapolation, Stepped (i.e. hold last value) or extrapolated (extend line based on preceding slope) will be server dependant. This is indicated by the *SteppedInterpolationMode* property. See Clause 4.8.2.
- o The aggregate should not extrapolate backwards in time. If there is no beginning bound, it must return *Bad\_NoData*. The trailing value should not be pulled backward in time.
- o If there happens to be a good raw value at the requested time, the raw value is returned.
- o If any non-good values are skipped in order to find the closest good value, the aggregate will be *Uncertain\_Subnormal*
- o Unless otherwise indicated, *StatusCodes* are *Good, Interpolated*.

The following examples demonstrate the various situations:

#### **5.6.3.5.2 Interpolated data with good bounding value.**

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:00:10	10	Raw, Good	13.5	Interpolated, Good	Value2 –Interpolated between values at 12:00:02 and 12:00:25
Jan-01-02 12:00:15	15	Interpolated, Good	15.7	Interpolated, Good	Value2 –Interpolated between values at 12:00:02 and 12:00:25

#### **5.6.3.5.3 Interpolated data with good bounding value with bad data in the interval.**

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:00:35	35	Interpolated, Uncertain	28.2	Interpolated, Good	Value2 –Interpolated between values at 12:00:28 and 12:00:39
Jan-01-02 12:00:40	40	Interpolated, Uncertain	31.1	Interpolated, Uncertain	Raw Value is Bad, Value2 – Interpolated between values at 12:00:39 and 12:00:48
Jan-01-02 12:00:45	45	Interpolated, Uncertain	36.7	Interpolated, Uncertain	Bounding Value Bad, Value2 – Interpolated between values at 12:00:39 and 12:00:48
Jan-01-02 12:00:50	50	Raw, Good	45	Interpolated, Good	
Jan-01-02 12:00:55	55	Interpolated, Good	51.5	Interpolated, Good	

#### **5.6.3.5.4 Interpolated data with no good end bounding value.**

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:01:20	80	Raw, Good	67.3*	Interpolated, Uncertain	Uncertain Values excluded. Value2 –Interpolated between values at 12:01:12 and 12:01:23
Jan-01-02 12:01:25	85	Interpolated, Good	76.7	Interpolated, Good	
Jan-01-02 12:01:30	90	Raw, Good	90	Raw, Good	
Jan-01-02 12:01:35	90	Interpolated, Uncertain	90	Interpolated, Uncertain	Bounding Value at 12:01:30, Extrapolated using stepped method

\* If Historian 2 had treated *Uncertain* values as *Good*. The value would be 70, interpolated between 12:00:17 and 12:00:2323 and the quality would be “*Interpolated, Good*”.

#### **5.6.3.5.5 Interpolated data with no good start bounding value.**

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	0	No Data, Bad	No bounding Value, do not extrapolate
Jan-01-2002 12:00:05	-	No Data, Bad	11.3	Interpolated, Good	Value 1 - No bounding value, do not extrapolate Value2 –Interpolated between values at 12:00:02 and 12:00:25
Jan-01-2002 12:00:10	10	Raw, Good	13.5	Interpolated, Good	Value2 –Interpolated between values at 12:00:02 and 12:00:25
Jan-01-2002 12:00:15	15	Interpolated, Good	15.7	Interpolated, Good	Value2 –Interpolated between values at 12:00:02 and 12:00:25

### 5.6.3.6 Average

#### 5.6.3.6.1 Description

The average aggregate adds up the values of all good raw data for each interval, and divides the sum by the number of good values. If any non-good values are ignored in the computation, the aggregate *StatusCode* will be determined using the *StatusCode Calculation* (See Clause 5.6.2.4)

If no data exists for an interval, the *StatusCode* of the aggregate for that interval will be *Good\_NoData*.

All interval aggregates return timestamp of the start of the interval. Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*.

#### 5.6.3.6.2 Average data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	10	Calculated, Good	-	No Data, Bad	Value2-No Raw data in interval
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	No Raw data in intervals

#### 5.6.3.6.3 Average data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	30	Calculated, Good	
Jan-01-2002 12:00:40	-	No Data, Bad	-	No Data, Bad	Value1-Only Bad data in interval Value 2- No data in interval
Jan-01-2002 12:00:45	-	No Data, Bad	40	Calculated, Good	Value 1- No data in interval
Jan-01-2002 12:00:50	50	Calculated, Good	50	Calculated, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	-	No Data, Bad	No data in intervals

#### 5.6.3.6.4 Average data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:20	80	Calculated, Good	70	Calculated, Good	
Jan-01-2002 12:01:25	-	No Data, Bad	80	Calculated, Good	Value 1- No data in interval
Jan-01-2002 12:01:30	90	Calculated, Good	90	Calculated, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	-	No Data, Bad	No data in intervals

### 5.6.3.6.5 Average data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	10	Partial, Good	Value1- No data in interval Value2 - Partial interval :02:-05
Jan-01-2002 12:00:05	-	No Data, Bad	-	No Data, Bad	No data in intervals
Jan-01-2002 12:00:10	10	Calculated, Good	-	No Data, Bad	Value 2- No data in interval
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	No data in intervals

### 5.6.3.7 TimeAverage

#### 5.6.3.7.1 Description

The time weighted average aggregate uses interpolation as described in the interpolated section above to find the value of a point at the beginning and end of an interval. A straight line is drawn between each raw value in the interval. The area under the line is divided by the length of the interval to yield the average.

For Example:

**Given:**

**Start:** Jan-01-2002 12:00:10

**End:** Jan-01-2002 12:00:15

**Interval:** 00:00:05

**Then:**

Point1 = Good Raw value of 10 at 12:00:10

Point2 = interpolated value of 15 at 12:00:15, using bounding values at 12:00:10 and 12:00:20.

Area under the line is 62.5 ( $1/2 \text{ base} * \text{height} + \text{base} * \text{height}$ ). Interval is 5 seconds

TimeAverage = Area/interval = 12.5

If any of an interval's raw values are non-good, they are ignored, and the aggregate *StatusCodes* for that interval is determined using the *StatusCodes* Calculation (See Clause 5.6.2.4)

All cases use the interpolated values determined in Cases outlined in section 5.6.3.5 for the bounding values.

#### 5.6.3.7.2 TimeAverage data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	12.5	Calculated, Good	14.5	Calculated, Good	Area under the line between 12:00:10 and 12:00:15 divided by interval length of 5
Jan-01-2002 12:00:15	17.5	Calculated, Good	16.7	Calculated, Good	

#### 5.6.3.7.3 TimeAverage data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	37.5	Calculated, Uncertain	29.7	Calculated, Uncertain	Value1– Interpolate values at :35 and :40 using bounds at :30 and :50 Value2– Interpolate values at :35 and :40 using bounds at :28 and :48 Uncertain means Bad Value ignored
Jan-01-2002 12:00:40	42.5	Calculated, Uncertain	33.9	Calculated Uncertain	Value1– Interpolate values at :40 and :45 using bounds at :30 and :50 Value2– Interpolate values at :40 and :45 using bounds at :39 and :48 Uncertain means Bad Value ignored
Jan-01-2002 12:00:45	47.5	Calculated, Uncertain	40.9	Calculated Uncertain	Value1– Interpolate value at :45 using bounds at :30 and :50 Value2– Interpolate value at :45 using bounds at :39 and :48 Interpolate Value at :50 using bounds at :48 and :52 Uncertain means Bad Value ignored
Jan-01-2002 12:00:50	52.5	Calculated, Good	48.3	Calculated, Good	Value1– Interpolate value at :55 using bounds at :50 and 01:00 Value2– Interpolate value at :50 using bounds at :48 and :52 Interpolate Value at :55 using bounds at :52 and :01:12
Jan-01-2002 12:00:55	57.5	Calculated, Good	52.8	Calculated, Good	Value1– Interpolate value at :55 using bounds at :50 and 01:00 Value2– Interpolate value at :50 using bounds at :48 and :52 Interpolate Value at :55 using bounds at :52 and :01:12

#### 5.6.3.7.4 TimeAverage data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:20	82.5	Calculated, Good	72.0	Calculated Uncertain	Value1– Interpolate value at :25 using bounds at :20 and :30 Value2– Interpolate value at :20 using bounds at :16 and :23 (Uncertain value at :17 is ignored by this historian) Interpolate Value at :25 using bounds at :23 and :26
Jan-01-2002 12:01:25	87.5	Calculated, Good	83.3	Calculated, Good	Value1– Interpolate value at :25 using bounds at :20 and :30 Value2– Interpolate value at :25 using bounds at :23 and :26
Jan-01-2002 12:01:30	90*	Calculated, Uncertain	90*	Calculated, Uncertain	Extrapolate Value at :35 using value at :30
Jan-01-2002 12:01:35	90*	Calculated, Uncertain	90*	Calculated, Uncertain	Extrapolate Values at :35 and :40 using value at :30

\* Stepped extrapolation is used at the boundary. Servers may opt to extrapolate data based on the previous slope.

### 5.6.3.7.5 TimeAverage data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	10.7	Partial, Uncertain	Value1-No bounding value, do not extrapolate. No data in the interval Value2- Interpolate value at :05 using bounds at :02 and :25 Use partial interval :02 to :05, with interval of 3.
Jan-01-2002 12:00:05	-	No Data, Bad	12.4	Calculated, Good	Value1-No bounding value, do not extrapolate. No data in the interval Value2- Interpolate values at :05 and 10 using bounds at :02 and :25
Jan-01-2002 12:00:10	12.5	Calculated, Good	14.5	Calculated, Good	Value1– Interpolate value at :15 using bounds at :10 and :20 Value2– Interpolate values at :10 and :15 using bounds at :02 and :25
Jan-01-2002 12:00:15	17.5	Calculated, Good	16.7	Calculated, Good	Value1– Interpolate value at :15 using bounds at :10 and :20 Value2– Interpolate values at :15 and :20 using bounds at :02 and :25

### 5.6.3.8 Total

#### 5.6.3.8.1 Description

The total aggregate adds up all the values of all good raw values for each interval. If any non-good values are ignored in the computation, the aggregate *StatusCode* will be determined using the *StatusCode* Calculation (See Clause 5.6.2.4).

If no data exists for an interval, the *StatusCode* of the aggregate for that interval will be *Good\_NoData*.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

#### 5.6.3.8.2 Total data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:00:10	10	Raw, Good	10	Calculated, Good	
Jan-01-02 12:00:15	0	Calculated, Good	0	Calculated, Good	

#### 5.6.3.8.3 Total data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:00:35	0	Calculated, Good	30	Calculated, Good	
Jan-01-02 12:00:40	-	No Data, Bad	-	No Data, Bad	
Jan-01-02 12:00:45	-	No Data, Bad	40	Calculated, Good	
Jan-01-02 12:00:50	50	Calculated, Good	50	Calculated, Good	
Jan-01-02 12:00:55	0	Calculated, Good	0	Calculated, Good	

#### 5.6.3.8.4 Total data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-02 12:01:20	80	Calculated, Good	70	Calculated, Good	
Jan-01-02 12:01:25	0	Calculated, Good	80	Calculated, Good	
Jan-01-02 12:01:30	90	Calculated, Good	90	Calculated, Good	
Jan-01-02 12:01:35	-	No Data, Bad	-	No Data, Bad	

#### 5.6.3.9 TotalizeAverage

##### 5.6.3.9.1 Description

The TotalizeAverage aggregate performs the following calculation for each interval:

$$\text{TotalizeAverage} = \text{time\_weighted\_avg} * \text{interval\_length (sec)}$$

Where:

Time\_weighted\_avg is the result from the TimeAverage aggregate, using the interval supplied to the TotalizeAverage call.

Interval\_length is the interval of the aggregate.

The resulting units would be normalized to seconds, i.e. [time\_weighted\_avg Units]\*sec.

If any non-good values are ignored in the computation of an interval, the aggregate *StatusCode* will be determined using the *StatusCode Calculation* (See Clause 5.6.2.4).

All interval aggregates return timestamp of the start of the interval. Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

#### 5.6.3.9.2 TotalizeAverage data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	12.5	Calculated, Good	72.5	Calculated, Good	Area under the line between 12:00:10 and 12:00:15
Jan-01-2002 12:00:15	17.5	Calculated, Good	83.5	Calculated, Good	

#### 5.6.3.10 Minimum

##### 5.6.3.10.1 Description

The minimum aggregate is the same as the minimum actual time, except the timestamp of the aggregate will always be the start of the interval for every interval.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*.

##### 5.6.3.10.2 Minimum data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	

##### 5.6.3.10.3 Minimum data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	30	Calculated, Good	
Jan-01-2002 12:00:40	-	No Data, Bad	40	Calculated, Bad	Value1- Only Bad data in interval.
Jan-01-2002 12:00:45	-	No Data, Bad	40	Calculated, Good	
Jan-01-2002 12:00:50	50	Raw, Good	50	Calculated, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	-	No Data, Bad	

##### 5.6.3.10.4 Minimum data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:20	80	Raw, Good	70	Calculated Good	
Jan-01-2002 12:01:25	-	No Data, Bad	80	Calculated Good	
Jan-01-2002 12:01:30	90	Raw, Good	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	-	No Data, Bad	

### 5.6.3.10.5 Minimum data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	10	Calculated Good	
Jan-01-2002 12:00:05	-	No Data, Bad	-	No Data, Bad	
Jan-01-2002 12:00:10	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	

### 5.6.3.10.6 Minimum data with Partial Interval.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:05	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:21	30	Partial, Good	20	Partial, Good	

## 5.6.3.11 Maximum

### 5.6.3.11.1 Description

This aggregate is the same as the minimum, except the value is the maximum raw value within the interval [s,e).

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*.

### 5.6.3.11.2 Maximum data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	

### 5.6.3.11.3 Maximum data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	30	Calculated, Good	
Jan-01-2002 12:00:40	-	No Data, Bad	40	Calculated, Bad	Only Bad data in interval.
Jan-01-2002 12:00:45	-	No Data, Bad	40	Calculated, Good	
Jan-01-2002 12:00:50	50	Raw, Good	50	Calculated, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	-	No Data, Bad	

#### 5.6.3.11.4 Maximum data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:20	80	Raw, Good	70	Calculated Good	
Jan-01-2002 12:01:25	-	No Data, Bad	80	Calculated Good	
Jan-01-2002 12:01:30	90	Raw, Good	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	-	No Data, Bad	

#### 5.6.3.11.5 Maximum data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	10	Calculated Good	
Jan-01-2002 12:00:05	-	No Data, Bad	-	No Data, Bad	
Jan-01-2002 12:00:10	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	-	No Data, Bad	

#### 5.6.3.11.6 Maximum data with Partial Interval.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:05	10	Raw, Good	-	No Data, Bad	
Jan-01-2002 12:00:21	30	Partial, Good	25	Partial, Good	

### 5.6.3.12 MininumActualTime

#### 5.6.3.12.1 Description

The minimum actual time aggregate retrieves the minimum good raw value within the interval [s,e), and returns that value with the timestamp at which that value occurs. Note that if the same minimum exists at more than one timestamp, the oldest one is retrieved, and the *StatusCode* is set to *MultiValues*. If a non-good value is lower than the good minimum, the *StatusCode* of the aggregate will be determined using the *StatusCode* Calculation (See Clause 5.6.2.4).

Unless otherwise indicated, *StatusCodes* are *Good, Raw*. If no values are in the interval no data is returned with a timestamp of the start of the interval. If only bad quality values are

available then the status is returned as Bad, Raw, The value is indeterminate, since some system may save bad values, but other may not.

#### **5.6.3.12.2 MinimumActualTime data with good bounding value.**

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:15	-	No Data, Bad	No raw data in interval, do not interpolate

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:15	-	No Data, Bad	No raw data in interval, do not interpolate

#### **5.6.3.12.3 MinimumActualTime data with good bounding value with bad data in the interval.**

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:40	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:45	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:50	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	No raw data in interval, do not interpolate

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:39	30	Raw, Good	
Jan-01-2002 12:00:42	-	Raw, Bad	Only Bad data in interval
Jan-01-2002 12:00:48	40	Raw, Good	
Jan-01-2002 12:00:52	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	No raw data in interval, do not interpolate

#### **5.6.3.12.4 MinimumActualTime data with no good end bounding value.**

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:01:20	80	Raw, Good	
Jan-01-2002 12:01:25	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:01:30	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:01:23	70	Raw, Good	
Jan-01-2002 12:01:26	80	Raw, Good	
Jan-01-2002 12:01:30	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	

### 5.6.3.12.5 MinimumActualTime data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:15	-	No Data, Bad	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:02	10	Raw, Good	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:10	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	

### 5.6.3.12.6 MinimumActualTime with Partial Interval.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:30	30	Partial, Good	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:25	20	Partial, Good	

### 5.6.3.13 MaximumActualTime

#### 5.6.3.13.1 Description

This is the same as the minimum actual time aggregate, except that the value is the maximum raw value within the interval [s,e). Note that if the same maximum exists at more than one timestamp, the oldest one is retrieved, and the *StatusCode* is set to *MultiValues*

Unless otherwise indicated, *StatusCodes* are *Good*, *Raw*.

#### 5.6.3.13.2 MaximumActualTime data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:15	-	No Data, Bad	No raw data in interval, do not interpolate

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:15	-	No Data, Bad	No raw data in interval, do not interpolate

#### 5.6.3.13.3 MaximumActualTime data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:40	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:45	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:00:50	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	No raw data in interval, do not interpolate

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:39	30	Raw, Good	
Jan-01-2002 12:00:42	-	No Data, Bad	Only Bad data in interval
Jan-01-2002 12:00:48	40	Raw, Good	
Jan-01-2002 12:00:52	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	No raw data in interval, do not interpolate

#### 5.6.3.13.4 MaximumActualTime data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:01:20	80	Raw, Good	
Jan-01-2002 12:01:25	-	No Data, Bad	No raw data in interval, do not interpolate
Jan-01-2002 12:01:30	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:01:23	70	Raw, Good	
Jan-01-2002 12:01:26	80	Raw, Good	
Jan-01-2002 12:01:30	90	Raw, Good	
Jan-01-2002 12:01:35	-	No Data, Bad	

### 5.6.3.13.5 MaximumActualTime data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:00	-	No Data, Bad	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:15	-	No Data, Bad	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:02	10	Raw, Good	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:10	-	No Data, Bad	
Jan-01-2002 12:00:15	-	No Data, Bad	

### 5.6.3.13.6 MaximumActualTime with Partial Interval.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:20	20	Raw, Good	
Jan-01-2002 12:00:30	30	Partial, Good	

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:05	-	No Data, Bad	
Jan-01-2002 12:00:28	25	Partial, Good	

### 5.6.3.14 Range

#### 5.6.3.14.1 Description

The range aggregate finds the difference between the raw maximum and raw minimum values in the interval. If only one value exists in the interval, the range is zero. Note that the range is always zero or positive.

If there are any non-good raw values in the interval, they are ignored, and the aggregate *StatusCodes* will be *Uncertain\_Subnormal*.

All interval aggregates are returned with timestamp of the start of the interval. Unless otherwise indicated, *StatusCodes* are *Good, Calculated*.

### 5.6.3.15 AnnotationCount

#### 5.6.3.15.1 Description

This aggregate returns a count of all annotations.

### 5.6.3.16 Count

#### 5.6.3.16.1 Description

This aggregate retrieves a count of all the raw values within an interval. If one or more raw values are non-good, they are not included in the count, and the aggregate *StatusCode* is determined using the *StatusCode Calculation* (See Clause 5.6.2.4). If no good data exists for an interval, the count is zero.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

#### 5.6.3.16.2 Count data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	1	Calculated, Good	0	Calculated, Good	
Jan-01-2002 12:00:15	0	Calculated, Good	0	Calculated, Good	

#### 5.6.3.16.3 Count data with uncertain data in the interval.

**Start:** Jan-01-2002 12:00:50 **End:** Jan-01-2002 12:01:30 **Interval:** 00:00:00

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:50	4*	Calculated, Good	4	Calculated, Uncertain	Value2 treats uncertain as bad, which also changes StatusCode

\* For servers with *TreatUncertainAsBad* = True then the result would be 3.

### 5.6.3.17 DurationInState0

#### 5.6.3.17.1 Description

This aggregate returns the time duration during the resample interval that the variable was in the zero state. If one or more raw values are non-good, they are not included in the duration, and the aggregate *StatusCode* is determined using the *StatusCode Calculation* (See Clause 5.6.2.4). If no good data exists for an interval, the duration is 0.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

### 5.6.3.18 DurationInState1

#### 5.6.3.18.1 Description

This aggregate returns the time duration during the resample interval that the variable was in the one state. If one or more raw values are non-good, they are not included in the duration, and the aggregate *StatusCode* is determined using the *StatusCode Calculation* (See Clause 5.6.2.4). If no good data exists for an interval, the duration is 0.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

### **5.6.3.19 NumberOfTransitions**

#### **5.6.3.19.1 Description**

This aggregate returns a count of the number of transition the variable had during the resample interval. If one or more raw values are non-good, they are not included in the duration, and the aggregate *StatusCode* is determined using the *StatusCode Calculation* (See Clause 5.6.2.4). If no good data exists for an interval, the number of transitions is 0.

Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*

### **5.6.3.20 Start**

#### **5.6.3.20.1 Description**

The start aggregate retrieves the first raw value within the interval [s,e), and returns that value with the timestamp at which that value occurs. If the value is non-good , than the *StatusCode* of the aggregate will be *Uncertain\_Subnormal*. Unless otherwise indicated, *StatusCodes* are *Good*, *Raw*.

#### **5.6.3.20.2 Start data with good bounding value.**

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	
Jan-01-2002 12:00:15	-	No Data, Bad	Return Timestamp of the interval

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	-	No Data, Bad	Return Timestamp of the interval
Jan-01-2002 12:00:15	-	No Data, Bad	Return Timestamp of the interval

#### **5.6.3.20.3 Start data with good bounding value with bad data in the interval.**

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	
Jan-01-2002 12:00:40	40	Raw, Bad	Raw Value (If Bad values are stored)
Jan-01-2002 12:00:45	-	No Data, Bad	
Jan-01-2002 12:00:50	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:39	30	Raw, Good	First raw in :35-:40 at :39
Jan-01-2002 12:00:42	40	Raw, Bad	Raw Value (If Bad values are stored)
Jan-01-2002 12:00:48	40	Raw, Good	First raw in :45-:50 at :48
Jan-01-2002 12:00:52	50	Raw, Good	First raw in :50-:55 at :52
Jan-01-2002 12:00:55	-	No Data, Bad	

#### 5.6.3.20.4 Start data with partial intervals.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	First raw in :05-:21 at :10
Jan-01-2002 12:00:30	30	Partial, Good	First raw in :21-:35 at :30

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:25	-	No Data, Bad	No raw data in :05-:21 at :10
Jan-01-2002 12:00:25	20	Raw, Good	First raw in :21-:35 at :25

#### 5.6.3.21 End

##### 5.6.3.21.1 Description

The end aggregate retrieves the last raw value within the interval [s,e), and returns that value with the timestamp at which that value occurs. If the value is non-good , than the *StatusCode* of the aggregate will be *Uncertain\_Subnormal*.

Unless otherwise indicated, *StatusCodes* are *Good*, *Raw*.

##### 5.6.3.21.2 End data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	Last raw in :10-:15 at :10
Jan-01-2002 12:00:15	-	No Data, Bad	Return Timestamp of the interval.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	-	No Data, Bad	Return Timestamp of the interval
Jan-01-2002 12:00:15	-	No Data, Bad	Return Timestamp of the interval

### 5.6.3.21.3 End data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:35	-	No Data, Bad	
Jan-01-2002 12:00:40	40	Raw, Bad	Raw Value (If Bad values are stored)
Jan-01-2002 12:00:45	-	No Data, Bad	
Jan-01-2002 12:00:50	50	Raw, Good	
Jan-01-2002 12:00:55	-	No Data, Bad	

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:39	30	Raw, Good	Last raw in :35-:40 at :39
Jan-01-2002 12:00:40	40	Raw, Bad	Raw Value (If Bad values are stored)
Jan-01-2002 12:00:48	40	Raw, Good	Last raw in :45-:50 at :48
Jan-01-2002 12:00:52	50	Raw, Good	Last raw in :50-:55 at :52
Jan-01-2002 12:00:55	-	No Data, Bad	

### 5.6.3.21.4 End data with partial intervals.

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 1		Notes
	Value	StatusCode	
Jan-01-2002 12:00:10	10	Raw, Good	Last raw in :05-:21 at :10
Jan-01-2002 12:00:30	30	Partial, Good	Last raw in :21-:35 at :30

**Start:** Jan-01-2002 12:00:05 **End:** Jan-01-2002 12:00:35 **Interval:** 00:00:16

Timestamp	Historian 2		Notes
	Value	StatusCode	
Jan-01-2002 12:00:25	-	No Data, Bad	No raw data in :05-:21 at :10
Jan-01-2002 12:00:28	25	Raw, Good	Last raw in :21-:35 at :28

## 5.6.3.22 Delta

### 5.6.3.22.1 Description

The delta aggregate retrieves the difference between the earliest and latest good raw values in an interval. If the last value is less than the first value, the result will be negative. If the last value is the same as the first value, or if the last value is also the first value at the same timestamp, the result will be zero. If the last value is greater than the first value, the result will be positive.

If any non-good values exist earlier or later than the earliest and latest good values, respectively, the aggregate is *Uncertain\_Subnormal*.

All interval aggregates are returned with timestamp of the start of the interval. Unless otherwise indicated, *StatusCodes* are *Good*, *Calculated*.

### 5.6.3.23 DurationGood

#### 5.6.3.23.1 Description

The duration good aggregate looks at the *StatusCode* of a bounding value of the interval to determine what the *StatusCode* is at the beginning of the interval. If no bounding value exists, the *StatusCode* is assumed to be bad at the start of the interval. This aggregate only considers truly Good values. Uncertain values are not considered Good for purposes of calculating this aggregate.

Whenever a raw value  $x$  with quality  $q$  is encountered from beginning to end within an interval, the quality is considered to be  $q$  until the next value,  $y$ , is encountered, at which point the quality becomes that of  $y$ , and so on.

The time is returned in seconds. No returned value will ever be *Uncertain\_Subnormal*.

Each interval's aggregate is returned with timestamp of the start of the interval. *StatusCodes* are *Good*, *Calculated*

#### 5.6.3.23.2 DurationGood data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	5	Calculated, Good	5	Calculated, Good	
Jan-01-2002 12:00:15	5	Calculated, Good	5	Calculated, Good	

#### 5.6.3.23.3 DurationGood data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	5	Calculated, Good	5	Calculated, Good	Value2-Good from :35 to :39. Good :39 to :40
Jan-01-2002 12:00:40	0	Calculated, Good	2	Calculated, Good	Value2-Good from :40 to :42. Bad :42 to :45
Jan-01-2002 12:00:45	0	Calculated, Good	2	Calculated, Good	Value2-Bad from :45 to :48. Good :48 to :50
Jan-01-2002 12:00:50	5	Calculated, Good	5	Calculated, Good	
Jan-01-2002 12:00:55	5	Calculated, Good	5	Calculated, Good	

#### 5.6.3.23.4 DurationGood data with no good end bounding value.

**Start:** Jan-01-2002 12:01:20 **End:** Jan-01-2002 12:01:40 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:20	5	Calculated, Good	2	Calculated, Good	Value2-Uncertain from :20 to :23. Good :23 to :25
Jan-01-2002 12:01:25	5	Calculated, Good	5	Calculated, Good	
Jan-01-2002 12:01:30	5	Calculated, Good	5	Calculated, Good	
Jan-01-2002 12:01:35	5	Calculated, Good	5	Calculated, Good	

### 5.6.3.23.5 DurationGood data with no good start bounding value.

**Start:** Jan-01-2002 12:00:00 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:00	0	Calculated, Good	3	Calculated, Good	Value1-No bound, Bad from :00 to :05 Value2-Bad from :00 to :02. Good :02 to :05
Jan-01-2002 12:00:05	0	Calculated, Good	5	Calculated, Good	Value1-No bound, Bad from :05 to :10
Jan-01-2002 12:00:10	5	Calculated, Good	5	Calculated, Good	
Jan-01-2002 12:00:15	5	Calculated, Good	5	Calculated, Good	

### 5.6.3.23.6 DurationGood data with uncertain data in the interval.

**Start:** Jan-01-2002 12:01:00 **End:** Jan-01-2002 12:01:30 **Interval:** 00:00:00

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:00	20*	Calculated, Good	25*	Calculated, Good	Value1-Uncertain from :10 to :20 Value1-Uncertain from :12 to :17

\* Uncertain data should not be counted as good.

## 5.6.3.24 DurationBad

### 5.6.3.24.1 Description

The duration bad aggregate looks at the quality of a bounding value of the interval to determine what the quality is at the beginning of the interval. If no bounding value exists, the quality is assumed to be bad at the start of the interval. This aggregate only considers truly Bad values. Uncertain values are not considered bad for purposes of calculating this aggregate.

Whenever a raw value  $x$  with quality  $q$  is encountered from beginning to end within an interval, the quality is considered to be  $q$  until the next value,  $y$ , is encountered, at which point the quality becomes that of  $y$ , and so on.

The time is returned in seconds. No returned value will ever be uncertain or subnormal.

Each interval's aggregate is returned with timestamp of the start of the interval. *StatusCodes* are *Good*, *Calculated*.

Duration Bad is not simply the interval minus duration good, since the interval uncertain data.

### 5.6.3.24.2 DurationBad data with good bounding value.

**Start:** Jan-01-2002 12:00:10 **End:** Jan-01-2002 12:00:20 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:10	0	Calculated, Good	0	Calculated, Good	
Jan-01-2002 12:00:15	0	Calculated, Good	0	Calculated, Good	

#### 5.6.3.24.3 DurationBad data with good bounding value with bad data in the interval.

**Start:** Jan-01-2002 12:00:35 **End:** Jan-01-2002 12:01:00 **Interval:** 00:00:05

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:00:35	0	Calculated, Good	0	Calculated, Good	
Jan-01-2002 12:00:40	5	Calculated, Good	3	Calculated, Good	
Jan-01-2002 12:00:45	5	Calculated, Good	3	Calculated, Good	
Jan-01-2002 12:00:50	0	Calculated, Good	0	Calculated, Good	
Jan-01-2002 12:00:55	0	Calculated, Good	0	Calculated, Good	

#### 5.6.3.24.4 DurationBad data with uncertain data in the interval.

**Start:** Jan-01-2002 12:01:00 **End:** Jan-01-2002 12:01:30 **Interval:** 00:00:00

Timestamp	Historian 1		Historian 2		Notes
	Value	StatusCode	Value	StatusCode	
Jan-01-2002 12:01:00	0	Calculated, Good	0	Calculated, Good	

### 5.6.3.25 PercentGood

#### 5.6.3.25.1 Description

This aggregate performs the following calculation:

`percent_good = duration_good / interval_length * 100`

Where:

`duration_good` is the result from the DURATIONGOOD aggregate, calculated using the interval supplied to PERCENTGOOD call.

`interval_length` is the interval of the aggregates.

No returned value will ever be uncertain or subnormal.

Each interval's aggregate is returned with timestamp of the start of the interval. *StatusCodes* are *Good*, *Calculated*.

The `interval_length` is the entire sample interval, regardless of quality.

### 5.6.3.26 PercentBad

#### 5.6.3.26.1 Description

This aggregate performs the following calculation:

`percent_bad = duration_bad / interval_length * 100`

Where:

`duration_good` is the result from the DURATIONBAD aggregate, calculated using the interval supplied to PERCENTBAD call.

`Interval_length` is the interval of the aggregates.

No returned value will ever be uncertain or subnormal.

Each interval's aggregate is returned with timestamp of the start of the interval. *StatusCodes* are *Good*, *Calculated*.

The `interval_length` is the entire sample interval, regardless of quality.

### 5.6.3.27 WorstQuality

#### 5.6.3.27.1 Description

This aggregate returns the worst quality of the raw values in the interval. That is, *Bad* status are worse than *Uncertain*, which are worse than *Good*. No distinction is made between the specific reasons for the status.

This aggregate returns the worst *StatusCode* as the value of the aggregate.

The timestamp is always the start of the interval. The *StatusCodes* are *Good*, *Calculated*.

## 6 Client conventions

### 6.1 How clients may request timestamps

The OPC HDA COM based specifications allowed clients to programmatically request historical time periods as absolute time (Jan 01, 2006 12:15:45) or a string representation of relative time (NOW -5M). The OPC UA specification does not allow for using a string representation to pass date/time information using the standard services.

OPC UA client applications that wish to visually represent date/time in a relative string format must convert this string format to UTC DateTime values before sending requests to the UA server. It is recommended that all OPC UA clients use the syntax defined in this section to represent relative times in their user interfaces.

The time is considered to be a relative time local to the server. This means that all times are given in UTC time, computed from the current time on the server's local clock. The format for the relative time is:

keyword+/-offset+/-offset...

where keyword and offset are as specified in the table below. Whitespace is ignored. The time string must begin with a keyword. Each offset must be preceded by a signed integer that specifies the number and direction of the offset. If the integer preceding the offset is unsigned, the value of the preceding sign is assumed (beginning default sign is positive). The keyword refers to the beginning of the specified time period. DAY means the timestamp at the beginning of the current day (00:00 hours, midnight), MONTH means the timestamp at the beginning of the current month, etc.

For example, "DAY -1D+7H30M" could represent the start time for data requested for a daily report beginning at 7:30 in the morning of the previous day (DAY = the first timestamp for today, -1D would make it the first timestamp for yesterday, +7H would take it to 7 a.m. yesterday, +30M would make it 7:30 a.m. yesterday (the + on the last term is carried over from the last term)).

Similarly, "MONTH-1D+5H" would be 5 a.m. on the last day of the previous month, "NOW-1H15M" would be an hour and fifteen minutes ago, and "YEAR+3MO" would be the first timestamp of April 1 this year.

Resolving relative timestamps is based upon what Microsoft has done with Excel, thus for various questionable time strings, we have these results:

10-Jan-2001 + 1 MO = 10-Feb-2001

29-Jan-1999 + 1 MO = 28-Feb-1999

31-Mar-2002 + 2 MO = 30-May-2002

29-Feb-2000 + 1 Y = 28-Feb-2001

In handling a gap in the calendar (due to different numbers of days in the month, or in the year), when one is adding or subtracting months or years:

Month: if the answer falls in the gap, it is backed up to the same time of day on the last day of the month.

Year: if the answer falls in the gap (February 29), it is backed up to the same time of day on February 28.

Note that the above does not hold for cases where one is adding or subtracting weeks or days, but only when adding or subtracting months or years, which may have different numbers of days in them.

Note that all keywords and offsets are specified in uppercase.

**Table 31 –Time Keyword Definitions**

Keyword	Description
NOW	The current UTC time as calculated on the server.
SECOND	The start of the current second.
MINUTE	The start of the current minute.
HOUR	The start of the current hour.
DAY	The start of the current day.
WEEK	The start of the current week.
MONTH	The start of the current month.
YEAR	The start of the current year.

**Table 32 –Time Offset Definitions**

Offset	Description
S	Offset from time in seconds.
M	Offset from time in minutes.
H	Offset from time in hours.
D	Offset from time in days.
W	Offset from time in weeks.
MO	Offset from time in months.
Y	Offset from time in years.