OPC Unified Architecture

Specification

Part 4: Services

Release 1.01

February 6, 2009

Specification Type:	Industry Standard Specification	Comments:	Report or view errata: http://www.opcfoundation.org/errata
Title:	OPC Unified Architecture Part 4 :Services	Date:	February 6, 2009
Version:	Release 1.01	Software:	MS-Word
		Source:	OPC UA Part 4 - Services 1.01 Specification.doc
Author:	OPC Foundation	Status:	Release

CONTENTS

FIGURES

TABLES

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 © 2006-2009, 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 FOUDATION 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

OPC Unified Architecture Specification

Part 4: Services

1 Scope

This specification defines the OPC Unified Architecture (OPC UA) *Services*. The *Services* described are the collection of abstract Remote Procedure Calls (RPC) that are implemented by OPC UA *Servers* and called by OPC UA *Clients*. All interactions between OPC UA *Clients* and *Servers* occur via these *Services*. The defined *Services* are considered abstract because no particular RPC mechanism for implementation is defined in this Part. Part 6 specifies one or more concrete mappings supported for implementation. For example, one mapping in Part 6 is to XML Web Services. In that case the *Services* described in this Part appear as the Web service methods in the WSDL contract.

Not all OPC UA *Servers* will need to implement all of the defined *Services*. Part 7 defines the *Profiles* that dictate which Services need to be implemented in order to be compliant with a particular *Profile*.

2 Reference documents

Part 1: OPC UA Specification: Part 1 – Concepts, Version 1.01 or later
http://www.opcfoundation.org/UA/Part1/

Part 2: OPC UA Specification: Part 2 – Security Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part2/

Part 3: OPC UA Specification: Part 3 – Address Space Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part3/

Part 5: OPC UA Specification: Part 5 – Information Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part5/

Part 6: OPC UA Specification: Part 6 – Mappings, Version 1.0 or later
http://www.opcfoundation.org/UA/Part6/

Part 7: OPC UA Specification: Part 7 – Profiles, Version 1.0 or later
http://www.opcfoundation.org/UA/Part7/

Part 8: OPC UA Specification: Part 8 – Data Access, Version 1.01 or later
http://www.opcfoundation.org/UA/Part8/

Part 11: OPC UA Specification: Part 11 – Historical Access, Version 1.01 or later
http://www.opcfoundation.org/UA/Part11/

Part 12: OPC UA Specification: Part 12 – Discovery, Version 1.0 or later
http://www.opcfoundation.org/UA/Part12/

Part 13: OPC UA Specification: Part 13 – Aggregates, Version 1.0 or later
http://www.opcfoundation.org/UA/Part13/

ISO/IEC 7498: Information Processing Systems - OSI Reference Model

http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

3 Terms, definitions, and conventions

3.1 OPC UA Part 1 terms

The following terms defined in 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 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 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 permitted range for value changes that will not trigger a data change *Notification*.

NOTE: *Deadband* can be applied as a 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. Part 8 defines an additional *Deadband* filter.

3.4.2 Endpoint

a physical address available on a network that allows *Clients* to access one or more *Services* provided by a *Server*.

NOTE: Each *Server* may have multiple *Endpoints*. The address of an *Endpoint* shall include a *HostName*.

3.4.3 Gateway Server

a *Server* that acts as an intermediary for one or more *Servers*.

NOTE: *Gateway Servers* may be deployed to limit external access, provide protocol conversion or to provide features which the underlying *Servers* do not support.

3.4.4 HostName

a unique identifier for a machine on a network.

NOTE: This identifier shall be unique within a local network, however, it may also be globally unique.

3.4.5 Security Token

an identifier for a cryptographic key set.

NOTE: All *Security Tokens* belong to a security context which is in case of OPC UA the *Secure Channel*.

3.4.6 ServerUri

a globally unique identifier for a *Server* application instance.

NOTE: The *ServerUri* may be a *GUID* generated automatically during install or it could be a unique *URL* assigned by the administrator.

3.4.7 SoftwareCertificate

A digital certificate for a software product, which can be installed on several hosts to describe the capabilities of the software product.

NOTE: 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 OPC 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
Request		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
Response		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 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. Data types that are *Service*-specific are defined in the parameter table of the *Service*.

Table 2 – Parameter Types defined in Part 3

Parameter Type
BaseDataType
NodeId
QualifiedName
LocaleId
Boolean
Byte
ByteString
Double
Guid
Int32
String
UInt16
UInt32
UInteger
UtcTime
XmlElement
Duration

The term *Services* used in this Part is consistent with the definition provided by http://www.opcfoundation.org/UA/Part13/

ISO/IEC 7498. The parameters of the Request and Indication service primitives are represented in Table 1 as Request parameters. Likewise, the parameters of the Response and Confirmation service primitives are represented in Table 1 as Response parameters. All request and response parameters are conveyed between the sender and receiver without change. Therefore, separate columns for request, indication, response, and confirmation parameter values are not needed and have been intentionally omitted to improve readability

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 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*

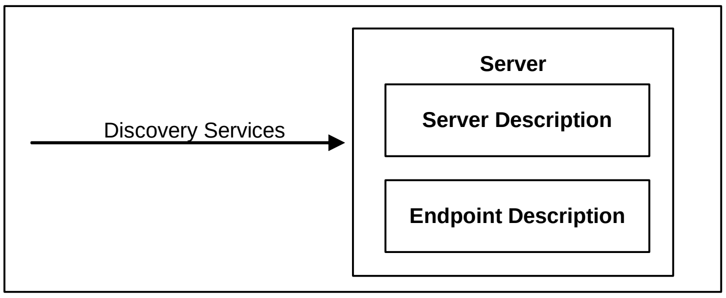

Figure 1 – Discovery Service Set

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*.

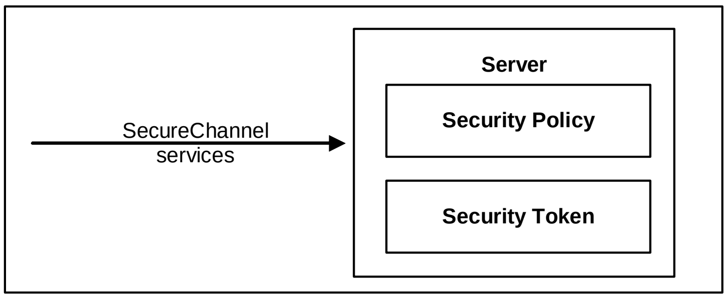

Figure 2 – SecureChannel Service Set

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*.

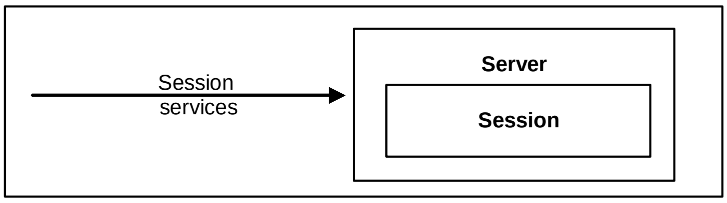

Figure 3 – Session Service Set

The *NodeManagement Service Set*, illustrated in Figure 4, defines *Services* that allow the *Client* to add, modify and delete *Nodes* in the *AddressSpace*.

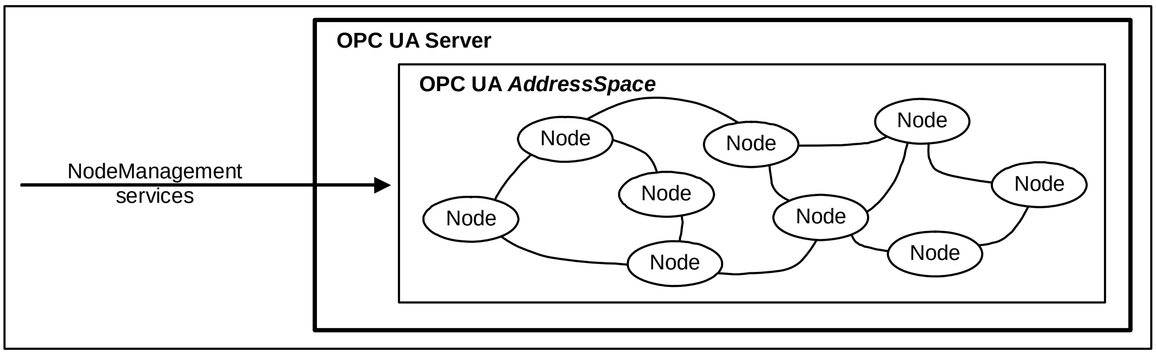

Figure 4 – NodeManagement Service Set

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 get a subset of data from the *AddressSpace* or the *View*.

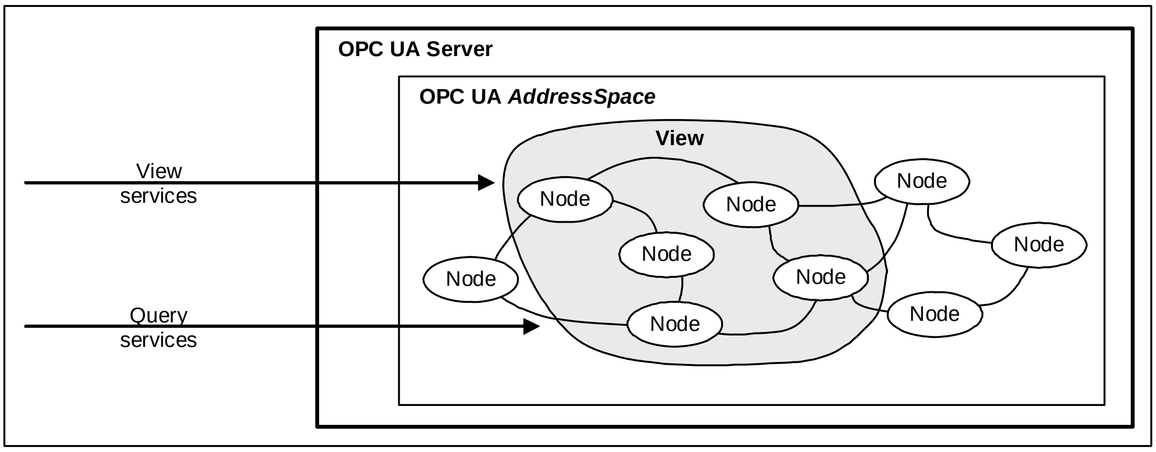

Figure 5 – View Service Set

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*.

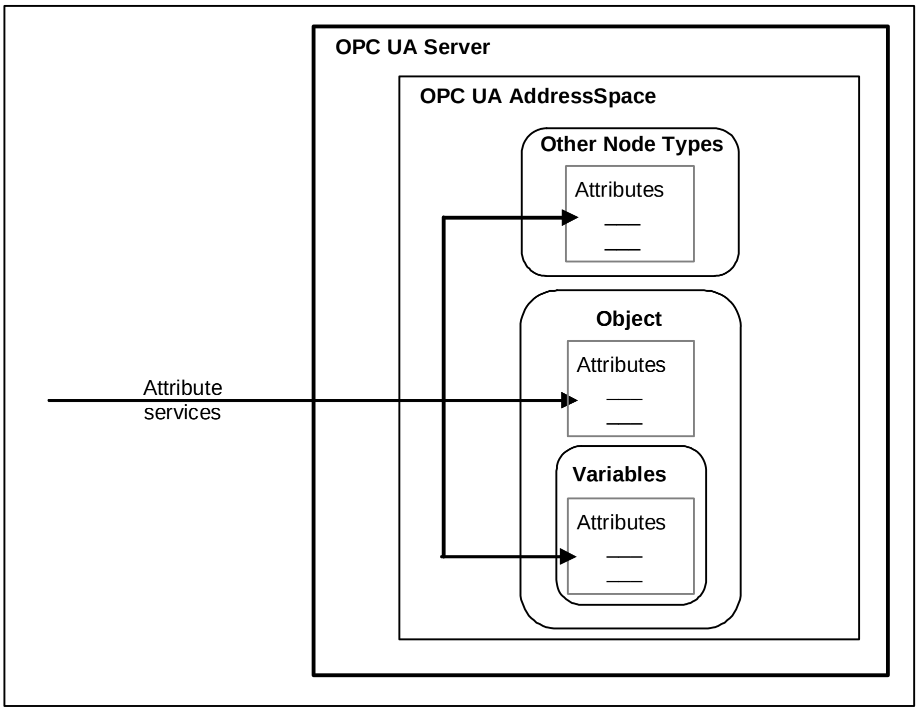

Figure 6 – Attribute Service Set

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.

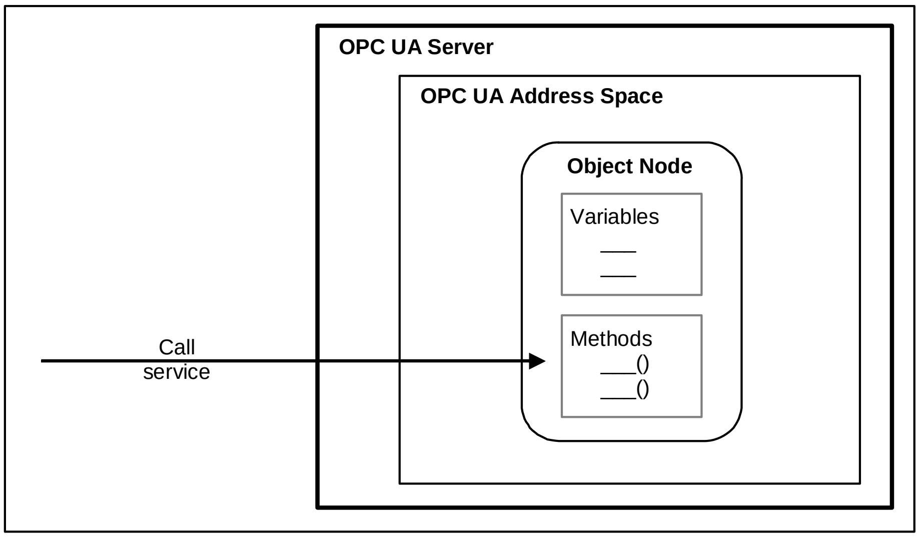

Figure 7 – Method Service Set

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* used to monitor *Attribut*es for value changes and *Objects* for *Events*.

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.

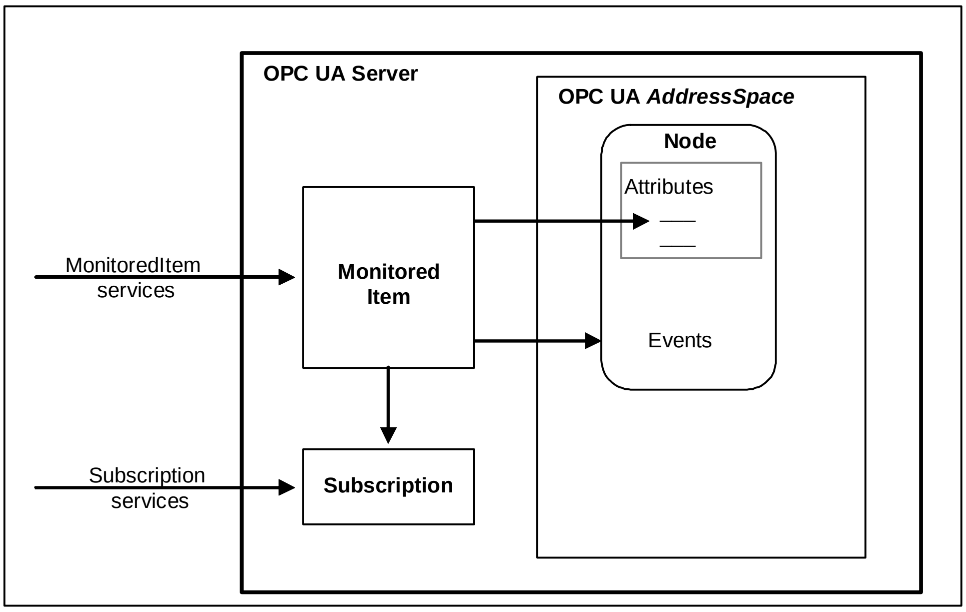

Figure 8 – MonitoredItem and Subscription Service Sets

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 for this step is specific to the communication technology used and is described in 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 shall 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 contains the definitions of common parameters used by these *Services*. 6.2 describes auditing requirements for all 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 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 7.26 and contains common request parameters such as *authenticationToken*, *timestamp* and *requestHandle*.

The *ResponseHeader* structure is defined in 7.27 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 *StatusCode* (see 7.33).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see 7.27). The mechanism for returning this parameter is specific to the communication technology used to convey the *Service* response and is defined in Part 6.

The status of individual operations in a request is represented by individual *StatusCode*s.

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 7.28.
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* shall 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 shall return a bad code in the *serviceResult* if the array is empty.

The *Services* define various specific *StatusCodes* and a *Server* shall use these specific *StatusCodes* as described in the *Service*. A *Client* should be able to handle these *Service* specific *StatusCodes*. In addition a *Client* shall expect other common *StatusCodes* defined in Table 165 and Table 166. Additional details for *Client* handling of specific *StatusCodes* may be defined in Part 7.

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* should immediately terminates all sessions and channels that use those credentials. In this case, the *Service* result code should be either *Bad_IdentityTokenRejected* or *Bad_CertificateUntrusted*.

Message parsing can fail due to syntax errors or if data contained within the message exceeds ranges supported by the receiver. When this happens messages shall be rejected by the receiver. If the receiver is a *Server* then it shall return a *ServiceFault* with result code of *Bad_DecodingError* or *Bad_EncodingLimitsExceeded*. If the receiver is the *Client* then the stack should report these errors to the *Client* application.

Many applications will place limits on the size of messages and/or data elements contained within these messages. For example, a *Server* may reject requests containing string values longer than a certain length. These limits are typically set by administrators and apply to all connections between a *Client* and a *Server*.

Clients that receive *Bad_EncodingLimitsExceeded* faults from the *Server* will likely have to reformulate their requests. The administrator may need to increase the limits for the *Client* if it receives a response from the *Server* with this fault.

In some cases, parsing errors are fatal and it is not possible to return a fault. For example, the incoming message could exceed the buffer capacity of the receiver. In these cases, these errors may be treated as a communication fault which requires the *SecureChannel* to be re-established (See 5.5).

The *Client* and *Server* reduce the chances of a fatal error by exchanging their message size limits in the *CreateSession* service. This will allow either party to avoid sending a message that causes a communication fault. The *Server* should return a *Bad_ResponseTooLarge* fault if a serialized response message exceeds the message size specified by the *Client*. Similarly, the *Client* stack should report a *Bad_RequestTooLarge* error to the application before sending a message that exceeds the *Server's* limit.

Note that the message size limits only apply to the raw message body and do not include headers or the effect of applying any security. This means that a message body that is smaller than the specified maximum could still cause a fatal error.

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*. Part 12 describes how to use the *Discovery Services* with dedicated *Discovery Servers.*

Every *Server* shall have a *Discovery Endpoint* that *Clients* can access without establishing a *Session*. This *Endpoint* may or may 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.

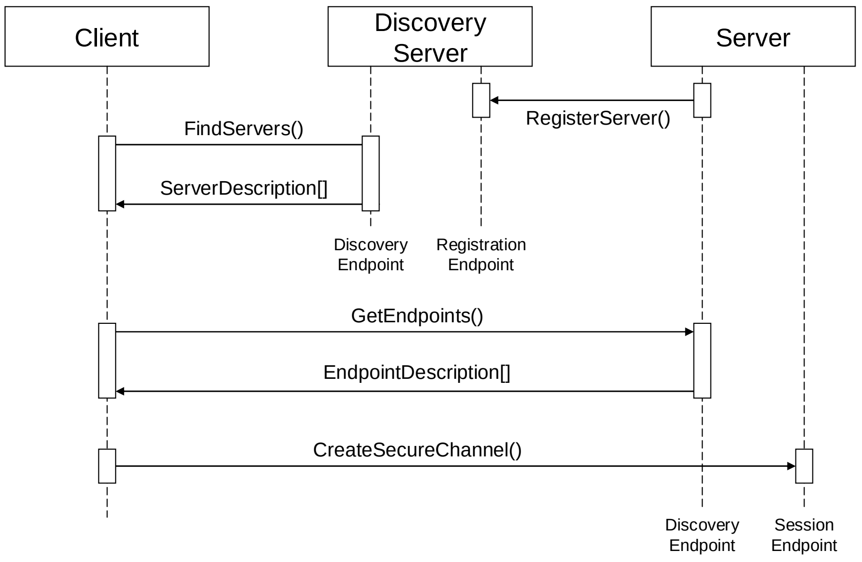

Figure 9 – The Discovery Process

The URL for a *Discovery Endpoint* shall provide all of the information that the *Client* needs to connect to the *Discovery Endpoint*.

Once a *Client* retrieves the *Endpoints*, the C*lient* 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 the *Server* configuration may have changed and the *Client* needs to go through the discovery process again.

Discovery Endpoints shall not require any message security, but it may require transport layer security. In production systems, Administrators may disable discovery for security reasons and *Clients* shall rely on cached *EndpointDescriptions*. To provide support for systems with disabled *Discovery Services Clients* shall allow *Administrators* to manually update the *EndpointDescriptions* used to connect to a *Server*. *Servers* shall allow *Administrators* to disable the *Discovery Endpoint*.

A *Client* shall be careful when using the information returned from a *Discovery Endpoint* since it has no security. A *Client* does this by comparing the information returned from the *Discovery Endpoint* to the information returned in the *CreateSession* response. A *Client* shall verify that:

1) The *HostName* specified in the *Server Certificate* is the same as the *HostName* contained in the *endpointUrl* provided in the *EndpointDescription.*

2) The *Server Certificate* returned in *CreateSession* response is the same as the *Certificate* used to create the *SecureChannel*.

3) The *EndpointDescriptions* returned from the *Discovery Endpoint* are the same as the *EndpointDescriptions* returned in the *CreateSession* response.

If the *Client* detects that one of the above requirements are not fulfilled, the *Client* shall close the *SecureChannel* and report an error.

5.4.2 FindServers

5.4.2.1 Description

This *Service* returns the *Servers* known to a *Server* or *Discovery Server*. The behaviour of *Discovery Servers* is described in detail in 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 5.4.2.2.

Every *Server* shall provide a *Discovery Endpoint* that supports this *Service*, however, the *Server* shall only return a single record that describes itself. *Gateway Servers* shall return a record for each *Server* that they provide access to plus (optionally) a record that allows the *Gateway Server* to be accessed as a an ordinary OPC UA *Server*.

Every *Server* shall 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* shall be the same value that appears in index 0 of the *ServerArray* property (see Part 5).

Every *Server* shall also have a human readable identifier called the *ServerName* which is not necessarily globally unique. This identifier may be available in multiple locales.

A *Server* may have multiple *HostNames*. For this reason, the *Client* shall pass the URL it used to connect to the *Endpoint* to this *Service*. The implementation of this *Service* shall use this information to return responses that are accessible to the *Client* via the provided URL.

This *Service* shall not require any message security but it may require transport layer security.

Some *Servers* may be accessed via a *Gateway Server* and shall have a value specified for *gatewayServerUri* in their *ApplicationDescription* (See 7.1). The *discoveryUrls* provided in *ApplicationDescription* shall belong to the *Gateway Server.* Some *Discovery Servers* may return multiple records for the same *Server* if that *Server* can be accessed via multiple paths.

This *Service* can be used without security and it is therefore vulnerable to DOS attacks. A *Server* should minimize the amount of processing required to send the response for this *Service*. This can be achieved by preparing the result in advance.

5.4.2.2 Parameters

Table 3 defines the parameters for the *Service*.

Table 3 – FindServers Service Parameters

Name	**Type**	**Description**
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
endpointUrl	String	The network address that the *Client* used to access the *Discovery Endpoint*. The *Server* uses this information for diagnostics and to determine what URLs to return in the response. The *Server* should return a suitable default URL if it does not recognize the *HostName* in the URL.
localeIds []	LocaleId	List of locales to use. The server should return the *ServerName* using one of locales specified. If the server supports more than one of the requested locales then the server shall 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.
Response		
responseHeader	ResponseHeader	Common response parameters. The *ResponseHeader* type is defined in 7.27.
servers []	ApplicationDescription	List of *Servers* that meet criteria specified in the request. This list is empty if no servers meet the criteria. The *ApplicationDescription* type is defined in 7.1.

5.4.2.3 Service results

Common *StatusCodes* are defined in Table 165.

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* shall 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 5.4.3.2.

A *Server* may support multiple security configurations for the same *Endpoint*. In this situation, the *Server* shall 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:

Server Application Instance Certificate

13) Message Security Mode

14) Security Policy

15) Supported User Identity Tokens

The *ApplicationInstanceCertificate* is used to secure the *OpenSecureChannel* request (See 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 type of user credentials shall be passed to the *Server* in the *ActivateSession* request (See 5.6.3).

Each *EndpointDescription* also specifies a URI for the *Transport Profile* that the *Endpoint* supports. The *Transport Profiles* specify information such as message encoding format and protocol version and are defined in Part 7. *Clients* shall fetch the *Server's SoftwareCertificates* if they want to discover the complete list of *Profiles* supported by the *Server* (See 7.30).

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*. 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*.

A *Server* may have multiple *HostNames*. For this reason, the *Client* shall pass the URL it used to connect to the *Endpoint* to this *Service*. The implementation of this *Service* shall use this information to return responses that are accessible to the *Client* via the provided URL.

This *Service* can be used without security and it is therefore vulnerable to DOS attacks. A *Server* should minimize the amount of processing required to send the response for this *Service*. This can be achieved by preparing the result in advance.

Some of the *EndpointDescriptions* returned in a response shall specify the *Endpoint* information for a *Gateway Server* that can be used to access another *Server*. In these situations, the *gatewayServerUri* is specified in the *EndpointDescription* and all security checks used to verify *Certificates* shall use the *gatewayServerUri* (See 6.1.4) instead of the *serverUri*.

To connect to a *Server* via the gateway the *Client* shall first establish a *SecureChannel* with the *Gateway Server*. Then the *Client* shall call the *CreateSession* service and pass the *serverUri* specified in the *EndpointDescription* to the *Gateway Server*. The *Gateway Server* shall then connect to the underlying *Server* on behalf of the *Client*. The process of connecting to *Server* via a *Gateway Server* is illustrated in Figure 10.

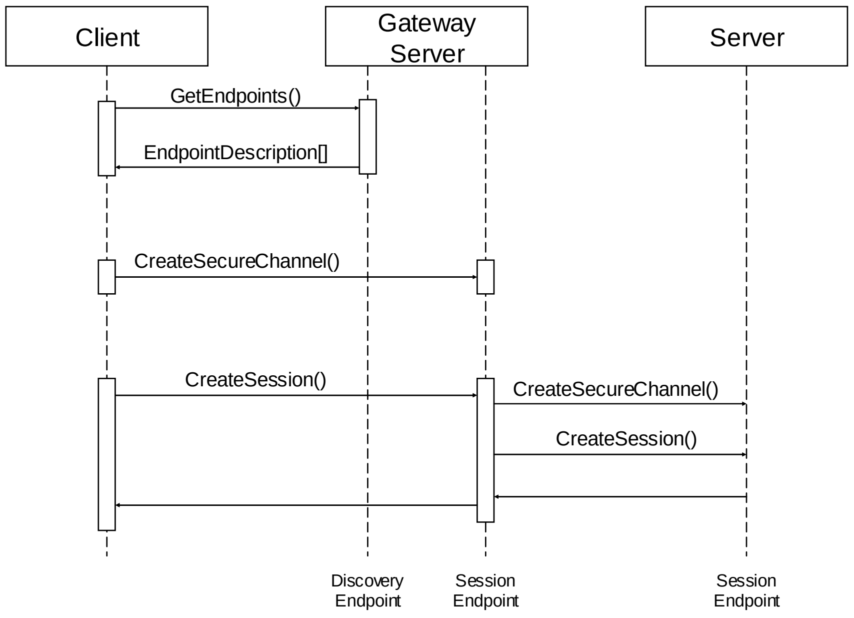

Figure 10 – Using a Gateway Server

5.4.3.2 Parameters

Table 4 defines the parameters for the *Service*.

Table 4 – GetEndpoints Service Parameters

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

5.4.3.3 Service Results

Common *StatusCodes* are defined in Table 165.

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* shall establish a *SecureChannel* with the *Discovery Server* before calling this *Service.* The *SecureChannel* is described in 5.5. The *Administrator* of the *Server* shall provide the *Server* with an *EndpointDescription* for the *Discovery Server* as part of the configuration process. *Discovery Servers* shall reject registrations if the *serverUri* provided does not match the *applicationUri* in *Server Certificate* used to create the *SecureChannel*.

A *Server* only provides its *ServerUri* and the URLs of the *Discovery Endpoints* to the *Discovery Server*. *Clients* shall use the *GetEndpoints* service to fetch the most up to date configuration information directly from the *Server*.

The *Server* shall provide a localized name for itself in all locales that it supports.

Servers shall 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 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* shall use depends on which category it falls into.

The registration process for manually launched *Servers* is illustrated in Figure 11.

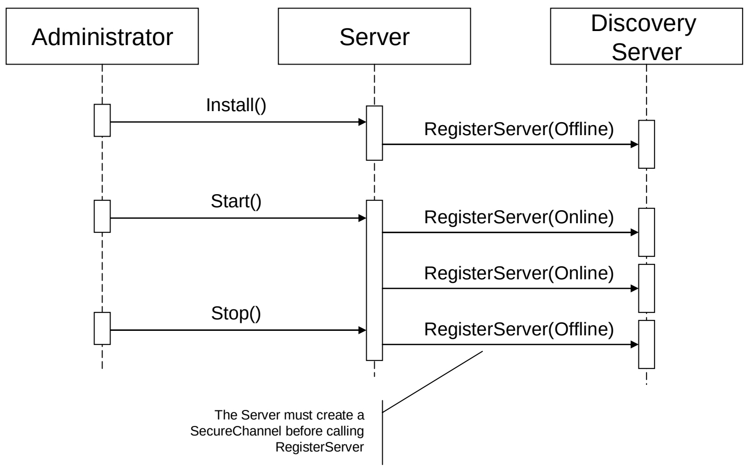

Figure 11 – The Registration Process – Manually Launched Servers

The registration process for automatically launched *Servers* is illustrated in Figure 12.

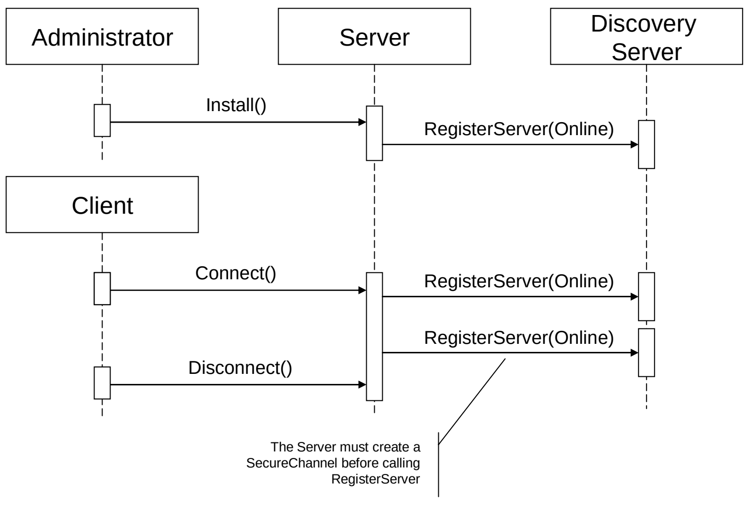

Figure 12 – The Registration Process – Automatically Launched Servers

The registration process is designed to be platform independent, robust and able to minimize errors created by misconfiguration. For that reason, *Servers* shall register themselves more than once.

Under normal conditions, *Servers* shall periodically register with the *Discovery Server* as long as they are able to receive connections from *Clients*. If a *Server* goes offline then it shall 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* shall 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* shall 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 – RegisterServer Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
server	RegisteredServer	The server to register.
serverUri	String	The globally unique identifier for the *Server* instance.
productUri	String	The globally unique identifier for the *Server* product.
serverNames []	LocalizedText	A list of localized descriptive names for the *Server*. The list shall have at least one valid entry.
serverType	Enum ApplicationType	The type of application. The enumeration values are defined in Table 103. The value "CLIENT_1" (The application is a *Client)* is not allowed.
gatewayServerUri	String	The URI of the *Gateway Server* associated with the *discoveryUrls*. This value is only specified by *Gateway Servers* that wish to register the *Servers* that they provide access to. For *Servers* that do not act as a *Gateway Server* this parameter shall be null.
discoveryUrls []	String	A list of *Discovery Endpoints* for the *Server*. The list shall have at least one valid entry.
semaphoreFilePath	String	The path to the semaphore file used to identify the server instance. The *Discovery Server* shall check that this file exists before returning the *ApplicationDescription* to the client. If the same semaphore file is used by another *Server* then that registration is deleted and replaced by the one being passed as part of this service invocation. If this value is null or empty then the *DiscoveryServer* does not attempt to verify the existence of the file.
isOnline	Boolean	True if the *Server* is currently able to accept connections from *Clients*.
Response		
ResponseHeader	ResponseHeader	Common response parameters. The type *ResponseHeader* is defined in 7.27.

5.4.4.3 Service Results

Table 6 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 6 – RegisterServer Service Result Codes

Symbolic Id	Description
Bad_ServerUriInvalid	See Table 165 for the description of this result code.
Bad_ServerNameMissing	No *ServerName* was specified.
Bad_DiscoveryUrlMissing	No *DiscoveryUrl* was specified.
Bad_SempahoreFileMissing	The semaphore file specified 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 OPC UA security are defined in Part 2.

The *SecureChannel Services* are unlike other *Services* because they are not implemented directly by the *OPC UA Application*. Instead, they are provided by the *Communication Stack* on which the *OPC UA Application* is built. For example, an OPC 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 *OPC UA Application* shall verify that the *Message* it received was in the context of a WS Secure Conversation. 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* shall use these algorithms when it creates a *SecureChannel*.

Note that some *SecurityPolicies* defined in Part 7 will turn off authentication and encryption resulting in a *SecureChannel* that provides no security.

When a *Client* and *Server* are communicating via a *SecureChannel*, they shall verify that all incoming *Messages* have been signed and encrypted according to the requirements specified in the *EndpointDescription*. An *OPC UA Application* shall not process any *Message* that does not conform to these requirements.

The relationship between the *SecureChannel* and the *OPC UA Application* depends on the implementation technology. Part 6 defines any requirements that depend on the technology used.

The correlation between the *OPC UA Application Session* and the *SecureChannel* is illustrated in Figure 13. The *Communication Stack* is used by the *OPC UA Applications* to exchange *Messages*. In a first step, the *SecureChannel Services* are used to establish a *SecureChannel* between the two *Communication Stacks* which allows the secure exchange of *Messages*. In a second step, the *OPC UA Applications* use the *Session Service Set* to establish a *OPC UA Application Session*.

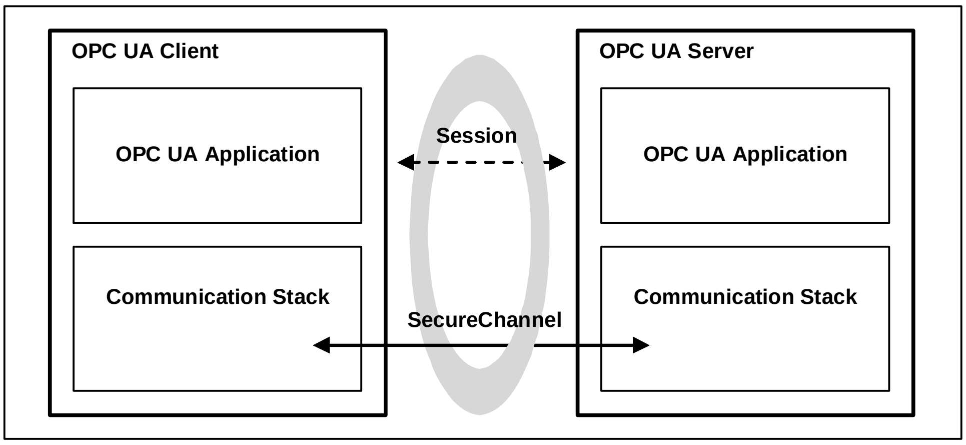

Figure 13 – SecureChannel and Session Services

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 5.6.3.

If a *Server* acts as a *Client* to other *Servers*, which is commonly referred to as *Server* chaining, then the Server shall 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. Whenever possible a *Server* should impersonate the original *Client* by passing the original *Client's* user identity to the underlying *Server* when it calls the *ActiveSession Service*. If impersonation is not an option then the *Server* shall map the original *Client's* user identity onto a new user identity which the underlying *Server* does recognize.

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 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 *SecurityTokens* 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 the *SecurityToken* expires or the *Server* receives a *Message* secured with a new *SecurityToken*. 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* shall be signed with the sender's *Certificate*. These *Messages* shall always be encrypted. If the transport layer does not provide encryption, then these *Messages* shall be encrypted with the receiver's *Certificate*.

The *Certificates* used in the *OpenSecureChannel* service shall be the application instance Certificates. *Clients* and *Servers* shall verify that the same *Certificates* were used in the *CreateSession* and *ActivateSession* services.

5.5.2.2 Parameters

Table 7 defines the parameters for the *Service*.

Table 7 – OpenSecureChannel Service Parameters

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

5.5.2.3 Service results

Table 8 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 8 – OpenSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecurityChecksFailed	See Table 165 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 165 for the description of this result code.
Bad_CertificateUriInvalid	See Table 165 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateUntrusted	See Table 165 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateRevoked	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 165 for the description of this result code.
Bad_RequestTypeInvalid	The security token request type is not valid.
Bad_SecurityModeRejected	The security mode does not meet the requirements set by the *Server*.
Bad_SecurityPolicyRejected	The security policy does not meet the requirements set by the *Server*.
Bad_SecureChannelIdInvalid	See Table 165 for the description of this result code.
Bad_NonceInvalid	See Table 165 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* shall be signed with the appropriate key associated with the current token for the *SecureChannel*.

5.5.3.2 Parameters

Table 9 defines the parameters for the *Service*.

Table 9 – CloseSecureChannel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *sessionId* is always omitted. The type *RequestHeader* is defined in 7.26.
secureChannelId	ByteString	The identifier for the *SecureChannel* to close.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).

5.5.3.3 Service results

Table 10 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 10 – CloseSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 165 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 two values which uniquely identify the *Session*. The first value is the *sessionId* which is used to identify the *Session* in the audit logs and in the *Server* address space. The second is the a*uthenticationToken* which is used to associate an incoming request with a *Session*.

Before calling this *Service*, the *Client* shall 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* shall associate with the a*uthenticationToken*. The *Server* may accept requests with the a*uthenticationToken* only if they are associated with the same *SecureChannel* that was used to create the *Session*. The *Client* may associate a new *SecureChannel* with the *Session* by calling the *ActivateSession* method.

The *SecureChannel* is always managed by the *Communication Stack* which means it shall provide APIs which the *Server* can use to find out information about the *SecureChannel* used for any given request. The *Communication Stack* shall, at a minimum, provide the *SecurityPolicy* and *SecurityMode* used by the *SecureChannel*. It shall also provide a *SecureChannelId* which uniquely identifies the *SecureChannel* or the *Client Certificate* used to establish the *SecureChannel*. The *Server* uses one of these to identify the *SecureChannel* used to send a request. 7.29 describes how to create the a*uthenticationToken* for different types of *Communication Stack*.

Depending upon on the *SecurityPolicy* and the *SecurityMode* of the *SecureChannel,* the exchange of the *Application Instance Certificates* and the *Nonces* may be optional and the signatures may be empty. See Part 7 for the definition of *SecurityPolicies* and the handling of these parameters.

The *Server* returns its *EndpointDescriptions* in the response. *Clients* use this information to determine whether the list of *EndpointDescriptions* returned from the *Discovery Endpoint* matches the *Endpoints* that the *Server* has. If there is a difference the *Client* shall close the *Session* and report an error. The *Server* returns all *EndpointDescriptions* for the *ServerUri* specified by the *Client* in the request. The *Client* only verifies *EndpointDescriptions* with a *transportProfileUri* that match the *profileUri* specified in original the *GetEndpoints* request. A *Client* may skip this check if the *EndpointDescriptions* were provided by a trusted source such as the *Administrator*.

The *Session* created with this *Service* shall 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 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 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 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* shall 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.

If a *Client* invokes the *CloseSession Service* then all *Subscriptions* associated with the *Session* are also deleted if the *deleteSubscriptions* flag is set to TRUE. If a *Server* terminates a *Session* for any other reason, *Subscriptions* associated with the *Session*, are not deleted. Each *Subscription* has its own lifetime to protect against data loss in the case of a *Session* termination. 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 OPC 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 Part 2 for additional information on auditing. A *Server* that maintains an audit log shall provide the information in the audit log entries via event *Messages* defined in this Specification. The *Server* may choose to only provide the *Audit* information via event *Messages*. The *Audit EventType* is defined in Part 3.

Second, these aggregating *Servers* may open independent *Sessions* to the underlying *Servers* for each *Client* that accesses data from them. Figure 14 illustrates this concept.

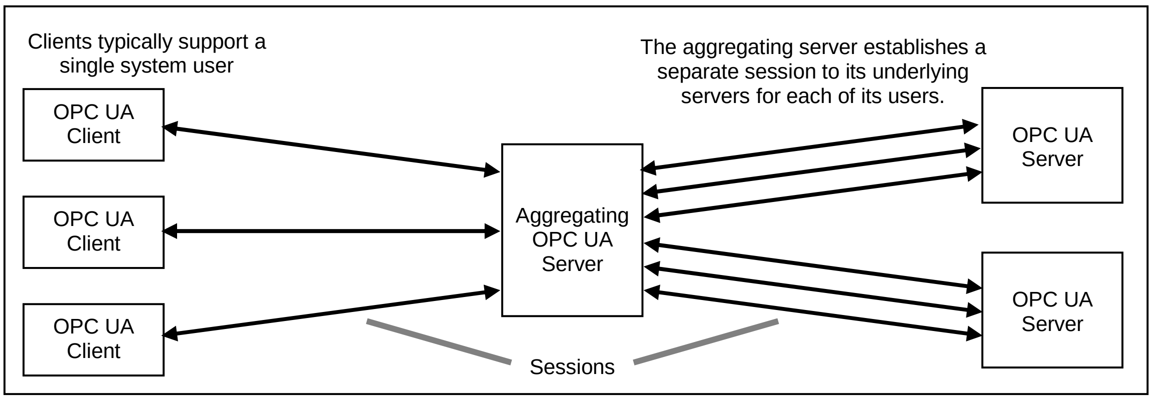
Clients typically support a single system user
The aggregating server establishes a separate session to its underlying servers for each of its users.
OPC UA Client
OPC UA Client
OPC UA Client
Aggregating OPC UA Server
OPC UA Server
OPC UA Server
Sessions

Figure 14 – 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
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
clientDescription	Application Description	Information that describes the *Client* application. The type *ApplicationDescription* is defined in 7.1.
serverUri	String	This value is only specified if the *EndpointDescription* has a *gatewayServerUri*. This value is the *applicationUri* from the *EndpointDescription* which is the *applicationUri* for the underlying *Server*. The type *EndpointDescription* is defined in 7.9.
endpointUrl	String	The network address that the *Client* used to access the *Session Endpoint*. The *HostName* portion of the URL should be one of the *HostNames* for the application that are specified in the *Server's ApplicationInstanceCertificate* (see Cause 7.2). The *Server* shall raise an *AuditUrlMismatchEventType* event if the URL does not match the Server's *HostNames*. *AuditUrlMismatchEventType* event type is defined in Part 5. The Server uses this information for diagnostics and to determine the set of *EndpointDescriptions* to return in the response.
sessionName	String	Human readable string that identifies the *Session*. The *Server* makes this name and the *sessionId* visible in its *AddressSpace* for diagnostic purposes. The *Client* should provide a name that is unique for the instance of the *Client*. If this parameter is not specified the *Server* shall assign a value.
clientNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. Profiles may increase the required length. The *Server* shall use this value to prove possession of its application instance *Certificate* in the response.
clientCertificate	ApplicationInstance Certificate	The application instance *Certificate* issued to the *Client*. The *ApplicationInstanceCertificate* type is defined in 7.2.
Requested SessionTimeout	Duration	Requested maximum number of milliseconds that a *Session* should remain open without activity. If the *Client* fails to issue a *Service* request within this interval, then the *Server* shall automatically terminate the *Client Session*.
maxResponse MessageSize	UInt32	The maximum size, in bytes, for the body of any response message. The *Server* should return a *Bad_ResponseTooLarge* service fault if a response message exceeds this limit. The value zero indicates that this parameter is not used. 5.3 provides more information on the use of this parameter.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type).
sessionId	NodeId	A unique *NodeId* assigned by the *Server* to the *Session*. This identifier is used to access the diagnostics information for the *Session* in the *Server* address space. It is also used in the audit logs and any events that report information related to the *Session*. The *Session* diagnostic information is described in Part 5. Audit logs and their related events are described in 6.2
authentication Token	Session AuthenticationToken	A unique identifier assigned by the *Server* to the *Session*. This identifier shall be passed in the *RequestHeader* of each request and is used with the *SecureChannelId* to determine whether a *Client* has access to the *Session*. This identifier shall not be reused in a way that the *Client* or the *Server* has a chance of confusing them with a previous or existing *Session*. The *SessionAuthenticationToken* type is described in 7.29.
revisedSession Timeout	Duration	Actual maximum number of milliseconds that a *Session* shall remain open without activity. The *Server* should attempt to honour the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The *Client* shall use this value to prove possession of its application instance *Certificate* in the *ActivateSession* request. This value may also be used to prove possession of the *userIdentityToken* it specified in the *ActivateSession* request.
serverCertificate	ApplicationInstance Certificate	The application instance *Certificate* issued to the *Server*. A *Server* shall prove possession by using the private key to sign the *Nonce* provided by the *Client* in the request. The *Client* shall verify that this *Certificate* is the same as the one it used to create the *SecureChannel*. The *ApplicationInstanceCertificate* type is defined in 7.2.
serverEndpoints []	Endpoint Description	List of *Endpoints* that the server supports. The *Server* shall return a set of *EndpointDescriptions* available for the *serverUri* specified in the request. The *EndpointDescription* type is defined in 7.9. The *Client* shall verify this list with the list from a *Discovery Endpoint* if it used a *Discovery Endpoint* to fetch the *EndpointDescriptions*.

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

5.6.2.3 Service results

Table 12 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 12 – CreateSession Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 165 for the description of this result code.
Bad_NonceInvalid	See Table 165 for the description of this result code.
Bad_SecurityChecksFailed	See Table 165 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 165 for the description of this result code.
Bad_CertificateUriInvalid	See Table 165 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateUntrusted	See Table 165 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateRevoked	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 165 for the description of this result code.
Bad_TooManySessions	The server has reached its maximum number of sessions.
Bad_ServerUriInvalid	See Table 165 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 shall be issued by the *Client* before it issues any other *Service* request after *CreateSession*. Failure to do so shall cause the *Server* to close the *Session*.

Whenever the *Client* calls this *Service* the *Client* shall 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 *serverCertificate* 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 shall 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* shall 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 shall 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 shall 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 shall 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 an 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 *serverCertificate* specified in the *CreateSession* response. If a token includes a secret then it should be encrypted using the public key from the *serverCertificate.*

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 reestablish 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.

A *Gateway Server* is expected to impersonate the user provided by the *Client* when it connects to the underlying *Server*. This means it shall re-calculate the signatures on the *UserIdentityToken* using the nonces provided by the underlying *Server*. The *Gateway Server* will have to use its own user credentials if the *UserIdentityToken* provided by the *Client* does not support impersonation.

5.6.3.2 Parameters

Table 13 defines the parameters for the *Service*.

Table 13 – ActivateSession Service Parameters

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

5.6.3.3 Service results

Table 14 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 14 – ActivateSession Service Result Codes

Symbolic Id	Description
Bad_IdentityTokenInvalid	See Table 165 for the description of this result code.
Bad_IdentityTokenRejected	See Table 165 for the description of this result code.
Bad_UserAccessDenied	See Table 165 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*.
Bad_IdentityChangeNotSupported	The *Server* does not support changing the user identity assigned to the session.

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 *StatusCode* 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
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
deleteSubscriptions	Boolean	If the value is TRUE, the Server deletes all Subscriptions associated with the Session. If the value is FALSE, the Server keeps the Subscriptions associated with the Session until they timeout based on their own lifetime.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).

5.6.4.3 Service results

Table 16 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 16 – CloseSession Service Result Codes

Symbolic Id	Description
Bad_SessionIdInvalid	See Table 165 for the description of this result code.

5.6.5 Cancel

5.6.5.1 Description

This *Service* is used to cancel outstanding Service requests. Successfully cancelled service requests shall respond with Bad_RequestCancelledByClient.

5.6.5.2 Parameters

Table 17 defines the parameters for the *Service*.

Table 17 – Cancel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
requestHandle	IntegerId	The *requestHandle* assigned to one or more requests that should be cancelled. All outstanding requests with the matching *requestHandles* shall be cancelled.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
cancelCount	UInt32	Number of cancelled requests.

5.6.5.3 Service results

Common *StatusCodes* are defined in Table 165.

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 Part 3).

5.7.2.2 Parameters

Table 18 defines the parameters for the *Service*.

Table 18 – AddNodes Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToAdd []	AddNodesItem	List of *Nodes* to add. All *Nodes* are added as a *Reference* to an existing *Node* using a hierarchical *ReferenceType*.
parentNodeId	Expanded NodeId	*ExpandedNodeId* of the parent *Node* for the *Reference*. The *ExpandedNodeId* type is defined in 7.10.
referenceTypeId	NodeId	*NodeId* of the hierarchical *ReferenceType* to use for the *Reference* from the parent *Node* to the new *Node*.
requestedNewNodeId	Expanded NodeId	*Client* requested expanded *NodeId* of the *Node* to add. The *serverIndex* in the expanded NodeId shall be 0. If the *Server* cannot use this *NodeId*, it rejects this *Node* and returns the appropriate error code. If the *Client* does not want to request a *NodeId*, then it sets the value of this parameter to the null expanded *NodeId*. If the *Node* to add is a *ReferenceType Node*, its *NodeId* should be a numeric id. See Part 3 for a description of *ReferenceType NodeIds*.
browseName	QualifiedName	The browse name of the *Node* to add.
nodeClass	NodeClass	*NodeClass* of the *Node* to add.
nodeAttributes	Extensible Parameter NodeAttributes	The *Attributes* that are specific to the *NodeClass*. The *NodeAttributes* parameter type is an extensible parameter type specified in 7.18. A *Client* is allowed to omit values for some or all *Attributes*. If an *Attribute* value is omitted, the *Server* shall use the default values from the *TypeDefinitionNode*. If a *TypeDefinitionNode* was not provided the *Server* shall choose a suitable default value. The *Server* may still add an optional Attribute to the *Node* with an appropriate default value even if the *Client* does not specify a value.
typeDefinition	Expanded NodeId	*NodeId* of the *TypeDefinitionNode* for the *Node* to add. This parameter shall be null for all *NodeClasses* other than *Object* and *Variable* in which case it shall be provided.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	AddNodesResult	List of results for the *Nodes* to add. The size and order of the list matches the size and order of the *nodesToAdd* request parameter.
statusCode	StatusCode	*StatusCode* for the *Node* to add (see 7.33 for *StatusCode* definition).
addedNodeId	NodeId	*Server* assigned *NodeId* of the added *Node*. Null *NodeId* if the operation failed.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Nodes* to add (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *nodesToAdd* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.2.3 Service results

Table 19 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 19 – AddNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.7.2.4 StatusCodes

Table 20 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 20 – AddNodes Operation Level Result Codes

Symbolic Id	Description
Bad_ParentNodeIdInvalid	The parent node id does not to refer to a valid node.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
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	See Table 166 for the description of this result code.
Bad_BrowseNameInvalid	See Table 166 for the description of this result code.
Bad_BrowseNameDuplicated	The browse name is not unique among nodes that share the same relationship with the parent.
Bad_NodeAttributesInvalid	The node *Attributes* are not valid for the node class.
Bad_TypeDefinitionInvalid	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 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* shall 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 21 defines the parameters for the *Service*.

Table 21 – AddReferences Service Parameters

Name	**Type**	**Description**
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
referencesToAdd []	AddReferences Item	List of *Reference* instances to add to the *SourceNode*. The *targetNodeClass* of each *Reference* in the list shall match the *NodeClass* of the *TargetNode*.
sourceNodeId	NodeId	*NodeId* of the *Node* to which the *Reference* is to be added. The source *Node* shall always exist in the *Server* to add the *Reference*. The isForward parameter can be set to FALSE if the target *Node* is on the local *Server* and the source *Node* on the remote *Server*.
referenceTypeId	NodeId	*NodeId* of the *ReferenceType* that defines the *Reference*.
isForward	Boolean	If the value is TRUE, the Server creates a forward Reference. If the value is FALSE, the Server creates an inverse Reference.
targetServerUri	String	URI of the remote *Server*. If this parameter is not null, it overrides the *serverIndex* in the *targetNodeId*.
targetNodeId	Expanded NodeId	Expanded *NodeId* of the *TargetNode*. The *ExpandedNodeId* type is defined in 7.10.
targetNodeClass	NodeClass	*NodeClass* of the *TargetNode*. The *Client* shall specify this since the *TargetNode* might not be accessible directly by the *Server*.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *References* to add (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *referencesToAdd* request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the *References* to add (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *referencesToAdd* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.3.3 Service results

Table 22 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 22 – AddReferences Service Result Codes

Symbolic Id	**Description**
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.7.3.4 StatusCodes

Table 23 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 23 – AddReferences Operation Level Result Codes

Symbolic Id	Description
Bad_SourceNodeIdInvalid	See Table 166 for the description of this result code.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_ServerUriInvalid	See Table 165 for the description of this result code.
Bad_TargetNodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeClassInvalid	See Table 166 for the description of this result code.
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 *Server*.
Bad_UserAccessDenied	See Table 165 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*.

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 *deleteTargetReferences* parameter.

When any of the *Nodes* deleted by an invocation of this *Service* is being monitored, then a *Notification* containing the status code Bad_NodeIdUnknown is sent to the monitoring *Client* indicating that the *Node* has been deleted.

5.7.4.2 Parameters

Table 24 defines the parameters for the *Service*.

Table 24 – DeleteNodes Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToDelete []	DeleteNodes Item	List of *Nodes* to delete
nodeId	NodeId	*NodeId* of the *Node* to delete.
deleteTargetReferences	Boolean	A *Boolean* parameter with the following values : TRUE delete *References* in *TargetNodes* that *Reference* the *Node* to delete. FALSE delete only the *References* for which the *Node* to delete is the source. The *Server* can not guarantee that he is able to delete all target *References* if this parameter is TRUE.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *Nodes* to delete (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the list of the *nodesToDelete* request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the *Nodes* to delete (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *nodesToDelete* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.4.3 Service results

Table 25 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 25 – DeleteNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.7.4.4 StatusCodes

Table 26 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 26 – DeleteNodes Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_NoDeleteRights	See Table 166 for the description of this result code.
Uncertain_ReferenceNotDeleted	The server was not able to delete all target references.

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* are contained in a *View*, then the *ViewVersion Property* is updated if this *Property* is supported.

The deletion of a Reference shall trigger a *ModelChange Event*.

5.7.5.2 Parameters

Table 24 defines the parameters for the *Service*.

Table 27 – DeleteReferences Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
referencesToDelete []	DeleteReferences Item	List of *References* to delete.
sourceNodeId	NodeId	*NodeId* of the *Node* that contains the *Reference* to delete.
referenceTypeId	NodeId	*NodeId* of the *ReferenceType* that defines the *Reference* to delete.
isForward	Boolean	If the value is TRUE, the Server deletes a forward Reference. If the value is FALSE, the Server deletes an inverse Reference.
targetNodeId	ExpandedNodeId	*NodeId* of the *TargetNode* of the *Reference*. If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *nodeId* shall contain the absolute namespace URI. If the *TargetNode* is a local *Node* the *nodeId* shall contain the namespace index.
deleteBidirectional	Boolean	A *Boolean* parameter with the following values : TRUE delete the specified *Reference* and the opposite *Reference* from the *TargetNode*. If the *TargetNode* is located in a remote *Server*, the *Server* is permitted to delete the specified *Reference* only. FALSE delete only the specified *Reference*.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *References* to delete (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *referencesToDelete* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *References* to delete (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *referencesToDelete* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.5.3 Service results

Table 28 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 28 – DeleteReferences Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.7.5.4 StatusCodes

Table 29 defines values for the r*esults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 29 – DeleteReferences Operation Level Result Codes

Symbolic Id	Description
Bad_SourceNodeIdInvalid	See Table 166 for the description of this result code.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_ServerIndexInvalid	The server index is not valid.
Bad_TargetNodeIdInvalid	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_NoDeleteRights	See Table 166 for the description of this result code.

5.8 View Service Set

5.8.1 Overview

Clients use the browse *Services* of the *View Service Set* to navigate through the *AddressSpace* or through a *View* which is a subset of the *AddressSpace*.

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*. See Part 5 for a description of the organisation of views in the *AddressSpace*.

5.8.2 Browse

5.8.2.1 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.

5.8.2.2 Parameters

Table 30 defines the parameters for the *Service*.

Table 30 – Browse Service Parameters

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

5.8.2.3 Service results

Table 31 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 31 – Browse Service Result Codes

Symbolic Id	Description
Bad_ViewIdUnknown	See Table 165 for the description of this result code.
Bad_ViewTimestampInvalid	See Table 165 for the description of this result code.
Bad_ViewParameterMismatchInvalid	See Table 165 for the description of this result code.
Bad_ViewVersionInvalid	See Table 165 for the description of this result code.
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.8.2.4 StatusCodes

Table 32 defines values for the r*esults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 32 – Browse Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_BrowseDirectionInvalid	See Table 166 for the description of this result code.
Bad_NodeNotInView	See Table 166 for the description of this result code.
Bad_NoContinuationPoints	See Table 166 for the description of this result code.
Uncertain_NotAllNodesAvailable	Browse results may be incomplete because of the unavailability of a subsystem.

5.8.3 BrowseNext

5.8.3.1 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* shall be submitted on the same *Session* that was used to submit the Browse or *BrowseNext* that is being continued.

5.8.3.2 Parameters

Table 33 defines the parameters for the *Service*.

Table 33 – BrowseNext Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
releaseContinuationPoints	Boolean	A *Boolean* parameter with the following values : TRUE passed *continuationPoints* shall be reset to free resources in the *Server*. FALSE passed *continuationPoints* shall be used to get the next set of browse information. A *Client* shall always use the continuation point returned by a *Browse* or *BrowseNext* response to free the resources for the continuation point in the *Server*. If the *Client* does not want to get the next set of browse information, *BrowseNext* shall be called with this parameter set to TRUE.
continuationPoints []	Continuation Point	A list of *Server*-defined opaque values that represent continuation points. The value for a continuation point was returned to the *Client* in a previous *Browse* or *BrowseNext* response. These values are used to identify the previously processed *Browse* or *BrowseNext* request that is being continued and the point in the result set from which the browse response is to continue. Clients may mix continuation points from different Browse or BrowseNext responses. The *ContinuationPoint* type is described in 7.6.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	BrowseResult	A list of references the met the criteria specified in the original *Browse* request. The size and order of this list matches the size and order of the *continuationPoints* request parameter. The *BrowseResult* type is defined in 7.3.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the *results* (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *results* response parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.8.3.3 Service results

Table 34 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 34 – BrowseNext Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.8.3.4 StatusCodes

Table 35 defines values for the r*esults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 35 – BrowseNext Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_BrowseDirectionInvalid	See Table 166 for the description of this result code.
Bad_NodeNotInView	See Table 166 for the description of this result code.
Bad_ContinuationPointInvalid	See Table 166 for the description of this result code.

5.8.4 TranslateBrowsePathsToNodeIds

5.8.4.1 Description

This *Service* is used to request that the *Server* translates 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 *ReferenceTypes* and *BrowseNames*.

One purpose of this *Service* is to allow programming against type definitions. Since *BrowseNames* shall 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* shall return a list of *NodeIds*. However, since one of the main purposes 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 as the first *NodeId* in the list.

5.8.4.2 Parameters

Table 36 defines the parameters for the *Service*.

Table 36 – TranslateBrowsePathsToNodeIds Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
browsePaths []	BrowsePath	List of browse paths for which *NodeIds* are being requested.
startingNode	NodeId	*NodeId* of the starting *Node* for the browse path.
relativePath	RelativePath	The path to follow from the *startingNode*. The last element in the *relativePath* shall always have a *targetName* specified. This further restricts the definition of the RelativePath type. The *Server* shall return *Bad_BrowseNameInvalid* if the *targetName* is missing. The *RelativePath* structure is defined in Section 7.25.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	BrowsePathResult	List of results for the list of browse paths. The size and order of the list matches the size and order of the *browsePaths* request parameter.
statusCode	StatusCode	*StatusCode* for the browse path (see 7.33 for *StatusCode* definition).
targets []	BrowsePathTarget	List of targets for the *relativePath* from the *startingNode*. A *Server* may encounter a *Reference* to a *Node* in another *Server* which it can not follow while it is processing the *RelativePath*. If this happens the *Server* returns the *NodeId* of the external *Node* and sets the *remainingPathIndex* parameter to indicate which *RelativePath* elements still need to be processed. To complete the operation the *Client* shall connect to the other *Server* and call this service again using the target as the *startingNode* and the unprocessed elements as the *relativePath*.
targetId	ExpandedNodeId	The identifier for a target of the *RelativePath*.
remainingPathIndex	Index	The index of the first unprocessed element in the *RelativePath*. This value shall be equal to the maximum value of *Index* data type if all elements were processed (see 7.12 for *Index* definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the list of browse paths (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *browsePaths* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.8.4.3 Service results

Table 37 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in 7.33.

Table 37 – TranslateBrowsePathsToNodeIds Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.8.4.4 StatusCodes

Table 38 defines values for the operation level *statusCode* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 38 – TranslateBrowsePathsToNodeIds Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_NothingToDo	See Table 165 for the description of this result code. This code indicates that the relativePath contained an empty list.
Bad_BrowseNameInvalid	See Table 166 for the description of this result code. This code indicates that a TargetName was missing in a RelativePath.
Uncertain_ReferenceOutOfServer	The path element has targets which are in another server.
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.

5.8.5 RegisterNodes

A *Server* often has no direct access to the information that it manages. Variables or services might be in underlying systems and additional effort is required to establish a connection to these systems. The *RegisterNodes Service* can be used by *Clients* to register the *Nodes* that they know they will access repeatedly (e.g. Write, Call). It allows *Servers* to set up anything needed so that the access operations will be more efficient. Clients can expect performance improvements when using registered *NodeIds*, but the optimization measures are vendor-specific. For *Variable Nodes Servers* shall concentrate their optimization efforts on the *Value Attribute*.

Registered *NodeIds* are only guaranteed to be valid within the current *Session*. *Clients* shall unregister unneeded Ids immediately to free up resources.

RegisterNodes does not validate the *NodeIds* from the request. *Servers* will simply copy unknown *NodeIds* in the response. Structural *NodeId* errors (size violations, invalid id types) will cause the complete *Service* to fail.

For the purpose of *Auditing*, *Servers* shall not use the registered *NodeIds* but only the canonical *NodeId*s which is the value of the *NodeId Attribute*.

5.8.5.1 Parameters

Table 39 defines the parameters for the *Service*.

Table 39 – RegisterNodes Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToRegister []	NodeId	List of *NodeIds* to register that the client has retrieved through browsing, querying or in some other manner.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
registeredNodeIds []	NodeId	A list of *NodeIds* which the *Client* shall use for subsequent access operations. The size and order of this list matches the size and order of the *nodesToRegister* request parameter. The *Server* may return the *NodeId* from the request or a new (an alias) *NodeId*. It is recommended that the Server return a numeric *NodeIds* for aliasing. In case no optimization is supported for a *Node*, the *Server* shall return the *NodeId* from the request.

5.8.5.2 Service results

Table 40 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 40 – RegisterNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_NodeIdInvalid	See Table 166 for the description of this result code. *Servers* shall completely reject the *RegisterNodes* request if any of the *NodeIds* in the *nodesToRegister* parameter are structurally invalid.

5.8.6 UnregisterNodes

5.8.6.1 Description

This *Service* is used to unregister *NodeIds* that have been obtained via the *RegisterNodes* service.

UnregisterNodes does not validate the *NodeIds* from the request. *Servers* shall simply unregister *NodeIds* that are known as registered *NodeIds*. Any *NodeIds* that are in the list, but are not registered *NodeIds* are simply ignored.

5.8.6.2 Parameters

Table 46 defines the parameters for the *Service*.

Table 41 – UnregisterNodes Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToUnregister []	NodeId	A list of *NodeIds* that have been obtained via the *RegisterNodes* service.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).

5.8.6.3 Service results

Table 47 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 42 – UnregisterNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.9 Query Service Set

5.9.1 Overview

This *Service Set* is used to issue a *Query* to a *Server*. OPC 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 OPC UA data stores and information management systems. OPC 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 OPC UA *Query Services* as part of an initialization process or an occasional information synchronization step. For example, OPC 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*.

See Annex B for example queries.

5.9.2 Querying Views

A *View* is a subset of the *AddressSpace* available in the *Server*. See 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*. OPC 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 shall be accomplished using the user identity of the *Client* as described in 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 7.4, also see B.1 for simple examples and B.2 for more complex 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 *References* are returned as part of the *QueryDataSets*.

5.9.3.2 Parameters

Table 43 defines the request parameters and Table 44 the response parameters for the *QueryFirst Service*.

Table 43 – QueryFirst Request Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
view	ViewDescription	Specifies a *View* and temporal context to a *Server* (see 7.37 for *ViewDescription* definition).
nodeTypes[]	NodeTypeDescription	This is the *Node* type description.
typeDefinitionNode	ExpandedNodeId	*NodeId* of the originating *TypeDefinitionNode* of the instances for which data is to be returned.
includeSubtypes	Boolean	A flag that indicates whether the *Server* should include instances of subtypes of the TypeDefinitionNode in the list of instances of the *Node* type.
dataToReturn[]	QueryDataDescription	Specifies an *Attribute* or *Reference* from the originating typeDefinitionNode along a given relativePath for which to return data.
relativePath	RelativePath	Browse path relative to the originating Node that identifies the Node which contains the data that is being requested, where the originating *Node* is an instance *Node* of the type defined by the type definition *Node*. The instance *Nodes* are further limited by the filter provided as part of this call. For a definition of relativePath see 7.25. This relative path could end on a *Reference*, in which case the *ReferenceDescription* of the *Reference* would be returned as its value.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* Id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in Part 6. If the *RelativePath* ended in a *Reference* 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 *NumericRange* type is defined in 7.21. This parameter is null if the specified *Attribute* is not an array or a structure. However, if the specified *Attribute* is an array or a structure, and this parameter is null, then all elements are to be included in the range.
filter	ContentFilter	Resulting *Nodes* shall be limited to the *Nodes* matching the criteria defined by the filter. ContentFilter is discussed in 7.4. If an empty filter is provided then the entire address space shall be examined and all *Nodes* that contain a matching requested *Attribute* or *Reference* are returned.
maxDataSetsToReturn	Counter	The number of *QueryDataSets* that the *Client* wants the *Server* to return in the response and on each subsequent continuation call response. The Server is allowed to further limit the response, but shall not exceed this limit. A value of 0 indicates that the *Client* is imposing no limitation.
maxReferencesToReturn	Counter	The number of *References* that the *Client* wants the *Server* to return in the response for each *QueryDataSet* and on each subsequent continuation call response. The Server is allowed to further limit the response, but shall not exceed this limit. A value of 0 indicates that the *Client* is imposing no limitation. For example a result where 4 *Nodes* are being returned, but each has 100 *References*, if this limit were set to 50 then only the first 50 *References* for each *Node* would be returned on the initial call and a continuation point would be set indicating additional data.

Table 44 – QueryFirst Response Parameters

Name	Type	Description
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
queryDataSets []	QueryDataSet	The array of *QueryDataSet*. This array is empty if no *Nodes* or *References* met the *nodeTypes* criteria. In this case the continuationPoint parameter shall be empty. The *QueryDataSet* type is defined in 7.22.
continuationPoint	ContinuationPoint	Server-defined opaque value that identifies the continuation point. The continuation point is used only when the *Query* 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 *QueryDataSets* to return exceeds the maximum number of *QueryDataSets* to return that was specified by the *Client* in the request. The continuation point is used in the *QueryNext Service*. When not used, the value of this parameter is null. If a continuation point is returned, the *Client* shall call *QueryNext* to get the next set of *QueryDataSets* or to free the resources for the continuation point in the *Server*. A continuation point shall remain active until the *Client* passes the continuation point to *QueryNext* or the session is closed. If the max continuation points have been reached the oldest continuation point shall be reset. The *ContinuationPoint* type is described in 7.6.
parsingResults[]	ParsingResult	List of parsing results for *QueryFirst*. The size and order of the list matches the size and order of the *NodeTypes* request parameter. This list is populated with any status codes that are related to the processing of the node types that are part of the query. The array can be empty if no errors where encountered. If any node type encountered an error all node types shall have an associated status code.
statusCode	StatusCode	Parsing result for the requested *NodeTypeDescription*.
dataStatusCodes []	StatusCode	List of results for *dataToReturn*. The size and order of the list matches the size and order of the *dataToReturn* request parameter. The array can be empty if no errors where encountered.
dataDiagnosticInfos []	DiagnosticInfo	List of diagnostic information *dataToReturn* (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *dataToReturn* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the query request.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the requested *NodeTypeDescription*. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the query request.
filterResult	ContentFilter Result	A structure that contains any errors associated with the filter. This structure shall be empty if no errors occurred. The *ContentFilterResult* type is defined in 7.4.2.

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, filterResult, parsingResults and optional *DiagnosticInfo* is returned. Table 45 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 45 – QueryFirst Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_ContentFilterInvalid	See Table 166 for the description of this result code.
Bad_ViewIdUnknown	See Table 165 for the description of this result code.
Bad_ViewTimestampInvalid	See Table 165 for the description of this result code.
Bad_ViewParameterMismatchInvalid	See Table 165 for the description of this result code.
Bad_ViewVersionInvalid	See Table 165 for the description of this result code.
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.

5.9.3.4 StatusCodes

Table 46 defines values for the parsingResults *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 46 – QueryFirst Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_NotTypeDefinition	The provided Nodeid was not a type definition nodeid.
Bad_AttributeIdInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.

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* shall 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 47 defines the parameters for the *Service*.

Table 47 – QueryNext Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
releaseContinuationPoint	Boolean	A *Boolean* parameter with the following values : TRUE passed *continuationPoint* shall be reset to free resources for the continuation point in the *Server*. FALSE passed *continuationPoint* shall be used to get the next set of *QueryDataSets*. A *Client* shall always use the continuation point returned by a *QueryFirst* or *QueryNext* response to free the resources for the continuation point in the *Server*. If the *Client* does not want to get the next set of *Query* information, *QueryNext* shall be called with this parameter set to TRUE. If the parameter is set to TRUE all array parameters in the response shall contain empty arrays.
continuationPoint	ContinuationPoint	Server defined opaque value that represents the continuation point. The value of the continuation point was returned to the *Client* in a previous *QueryFirst* or *QueryNext* response. This value is used to identify the previously processed *QueryFirst* or *QueryNext* request that is being continued, and the point in the result set from which the browse response is to continue. The *ContinuationPoint* type is described in 7.6.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
queryDataSets []	QueryDataSet	The array of *QueryDataSets*. The *QueryDataSet* type is defined in 7.22.
revisedContinuationPoint	ContinuationPoint	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 *releaseContinuationPoint* is set, the value of this parameter is null. The *ContinuationPoint* type is described in 7.6.

5.9.4.3 Service results

Table 48 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 48 – QueryNext Service Result Codes

Symbolic Id	**Description**
Bad_ContinuationPointInvalid	See Table 166 for the description of this result code.

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 that which 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 49 defines the parameters for the *Service*.

Table 49 – Read Service Parameters

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

5.10.2.3 Service results

Table 50 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 50 – Read Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_MaxAgeInvalid	The max age parameter is invalid.
Bad_TimestampsToReturnInvalid	See Table 165 for the description of this result code.

5.10.2.4 StatusCodes

Table 51 defines values for the operation level *statusCode* contained in the *DataValue* structure of each *values* element. Common *StatusCodes* are defined in Table 166.

Table 51 – Read Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_AttributeIdInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeNoData	See Table 166 for the description of this result code.
Bad_DataEncodingInvalid	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.
Bad_NotReadable	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 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 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 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 Part 11.

5.10.3.2 Parameters

Table 52 defines the parameters for the *Service*.

Table 52 – HistoryRead ServiceParameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
historyReadDetails	Extensible Parameter HistoryReadDetails	The details define the types of history reads that can be performed. The *HistoryReadDetails* parameter type is an extensible parameter type formally defined in Part 11. The *ExtensibleParameter* type is defined in 7.11.
timestampsToReturn	enum TimestampsTo Return	An enumeration that specifies the timestamps to be returned for each requested *Variable Value Attribute*. The *TimestampsToReturn* enumeration is defined in 7.34. Specifying a *TimestampsToReturn* of NEITHER is not valid. A *Server* shall return a *Bad_InvalidTimestampArgument StatusCode* in this case.
releaseContinuation Points	Boolean	A *Boolean* parameter with the following values : TRUE passed *continuationPoints* shall be reset to free resources in the *Server*. FALSE passed *continuationPoints* shall be used to get the next set of historical information. A *Client* shall always use the continuation point returned by a *HistoryRead* response to free the resources for the continuation point in the *Server*. If the *Client* does not want to get the next set of historical information, *HistoryRead* shall 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 *HistoryReadDetails* is RAW, PROCESSED, MODIFIED or ATTIME: The *nodeId* of the *Nodes* whose historical values are to be read. The value returned shall always include a timestamp. If the *HistoryReadDetails* is EVENTS: The *NodeId* of the *Node* whose *Event* history is to be read. If the *Node* does not support the requested access for historical values or historical *Events* the appropriate error response for the given *Node* shall be generated.
indexRange	NumericRange	This parameter is used to identify a single element of 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 *NumericRange* type is defined in 7.21. This parameter is null if the value is not an array. However, if the value is an array, and this parameter is null, then all elements are to be included in the range.
dataEncoding	QualifiedName	A *QualifiedName* that specifies the data encoding to be returned for the *Value* to be read (see 7.23 for definition how to specify the data encoding). The parameter is ignored when reading history of *Events*.
continuationPoint	ByteString	For each *NodeToRead* this parameter specifies a continuation point returned from a previous *HistoryRead* call, allowing the *Client* to continue that read from the last value received. The *HistoryRead* is used to select an ordered sequence of historical values or events. A continuation point marks a point in that ordered sequence, such that the *Server* 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 Part 11.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type).
results []	HistoryReadResult	List of read results. The size and order of the list matches the size and order of the *nodesToRead* request parameter.
statusCode	StatusCode	*StatusCode* for the *NodeToRead* (see 7.33 for *StatusCode* 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. When this parameter is not used, its value is null. *Servers* shall support at least one continuation point per *Session*. *Servers* specify a max history continuation points per *Session* in the *Server* capabilities *Object* defined in Part 5. A continuation point shall remain active until the *Client* passes the continuation point to *HistoryRead* or the *Session* is closed. If the max continuation points have been reached the oldest continuation point shall be reset.
historyData	Extensible Parameter HistoryData	The history data returned for the *Node*. The *HistoryData* parameter type is an extensible parameter type formally defined in Part 11. It specifies the types of history data that can be returned. The *ExtensibleParameter* base type is defined in 7.11.
diagnosticInfos []	Diagnostic Info	List of diagnostic information. The size and order of the list matches the size and order of the *nodesToRead* request parameter. There is one entry in this list for each *Node* contained in the *nodesToRead* parameter. This list is empty if

		diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.3.3 Service results

Table 53 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 53 – HistoryRead Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 165 for the description of this result code.
Bad_HistoryOperationInvalid	See Table 166 for the description of this result code.
Bad_HistoryOperationUnsupported	See Table 166 for the description of this result code.

5.10.3.4 StatusCodes

Table 54 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166. History access specific *StatusCodes* are defined in Part 11.

Table 54 – HistoryRead Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_DataEncodingInvalid	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_ContinuationPointInvalid	See Table 165 for the description of this result code.
Bad_InvalidTimestampArgument	The defined timestamp to return was invalid.

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 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* should 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*.

If a *Server* allows writing of *Attributes* with the *DataType* LocalizedText, the Client can add or overwrite the text for a locale by writing the text with the associated *LocaleId*. Writing a null *String* for a locale shall delete the *String* for that locale. Writing a null *String* for the text and a null *String* for the *LocaleId* shall delete the entries for all locales.

5.10.4.2 Parameters

Table 55 defines the parameters for the *Service*.

Table 55 – Write Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToWrite []	WriteValue	List of *Nodes* and their *Attributes* to write.
nodeId	NodeId	*NodeId* of the *Node* that contains the *Attributes*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in Part 6.
indexRange	NumericRange	This parameter is used to identify a single element of an array, or a single range of indexes for arrays. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is not used if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array 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 *Node*'s *Attribute* value (see 7.7 for *DataValue* definition). If the *indexRange* parameter is specified then the *Value* shall be an array even if only one element is being written. If the *SourceTimestamp* or the *ServerTimestamp* is specified, the *Server* shall use these values. The *Server* returns a Bad_WriteNotSupported error if it does not support writing of timestamps. A Server shall return a Bad_TypeMismatch error if the data type of value is not the same as or a subtype of datatype for the *Attribute*. The *Server* returns a Bad_DataEncodingUnsupported error if it does not support writing of the passed data encoding.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of results for the *Nodes* to write (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *nodesToWrite* request parameter. There is one entry in this list for each *Node* contained in the *nodesToWrite* parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Nodes* to write (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *nodesToWrite* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.4.3 Service results

Table 56 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 56 – Write Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.10.4.4 StatusCodes

Table 57 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 57 – Write Operation Level Result Codes

Symbolic Id	Description
Good_CompletesAsynchronously	See Table 165 for the description of this result code. The value was successfully written to an intermediate system but the *Server* does not know if the data source was updated properly.
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_AttributeIdInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeNoData	See Table 166 for the description of this result code.
Bad_WriteNotSupported	The requested write operation is not supported. If a *Client* attempts to write any value, quality, timestamp combination and the *Server* does not support the requested combination (which could be a single quantity such as just timestamp), then the *Server* shall not perform any write on this *Node* and shall return this *StatusCode* for this *Node*.
Bad_NotWritable	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code. The current user does not have permission to write the attribute.
Bad_OutOfRange	See Table 166 for the description of this result code.
Bad_TypeMismatch	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.

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 58 defines the parameters for the *Service*.

Table 58 – HistoryUpdate Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
historyUpdateDetails []	Extensible Parameter HistoryUpdate Details	The details defined for this update. The *HistoryUpdateDetails* parameter type is an extensible parameter type formally defined in Part 11. It specifies the types of history updates that can be performed. The *ExtensibleParameter* type is defined in 7.11.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* 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 *historyUpdateDetails* parameter specified in the request.
statusCode	StatusCode	*StatusCode* for the update of the *Node* (see 7.33 for *StatusCode* definition).
operationResults []	StatusCode	List of *StatusCodes* for the operations to be performed on a *Node*. The size and order of the list matches the size and order of any list defined by the details element being reported by this *updateResults* entry.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the operations to be performed on a *Node* (see 7.8 for *DiagnosticInfo* 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 *updateResults* entry. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.
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 *historyUpdateDetails* parameter specified in the request. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.5.3 Service results

Table 59 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 59 – HistoryUpdate Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.10.5.4 StatusCodes

Table 60 defines values for the *statusCode* and *operationResult* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166. History access specific *StatusCodes* are defined in Part 11.

Table 60 – HistoryUpdate Operation Level Result Codes

Symbolic Id	Description
Bad_NotWritable	See Table 166 for the description of this result code.
Bad_HistoryOperationInvalid	See Table 166 for the description of this result code.
Bad_HistoryOperationUnsupported	See Table 166 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 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* shall 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.

5.11.2 Call

5.11.2.1 Description

This *Service* is used to call (invoke) a list of *Methods*. 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.

5.11.2.2 Parameters

Table 61 defines the parameters for the *Service*.

Table 61 – Call Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
methodsToCall []	CallMetthodRequest	List of *Methods* to call.
objectId	NodeId	The *NodeId* shall be that of the *Object* or *ObjectType* that is the source of a *HasComponent Reference* (or subtype of *HasComponent Reference*) to this *Method*. See Part 3 for a description of *Objects* and their *Methods*.
methodId	NodeId	*NodeId* of the *Method* 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 *InputArguments Property* of the *Method*. The name, a description and the data type of each argument are defined by the *Argument* structure in each element of the method's *InputArguments Property*.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	CallMethodResult	Result for the *Method* calls.
statusCode	StatusCode	*StatusCode* of the *Method* executed in the server. This *StatusCode* is set to the Bad_InvalidArgument *StatusCode* if at least one input argument broke a constraint (e.g. wrong data type, value out of range). This *StatusCode* is set to a bad *StatusCode* if the *Method* execution failed in the server, e.g. based on an exception or an HRESULT.
inputArgumentResults []	StatusCode	List of *StatusCodes* for each *inputArgument*.
inputArgumentDiagnosticInfos []	DiagnosticInfo	List of diagnostic information for each *inputArgument*. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.
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 *OutputArguments Property* of the *Method*. The name, a description and the data type of each argument are defined by the *Argument* structure in each element of the methods *OutputArguments Property*.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *StatusCode* of the *callResult*. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.11.2.3 Service results

Table 62 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 62 – Call Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.11.2.4 StatusCodes

Table 60 defines values for the *statusCode* and *operationResult* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 63 – Call Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code. Used to indicate that the specified object is not valid.
Bad_NodeIdUnknown	See Table 166 for the description of this result code. Used to indicate that the specified object is not valid.
Bad_ArgumentsMissing	The client did not specify all of the input arguments for the method.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object.
Bad_OutOfRange	See Table 166 for the description of this result code. Used to indicate that an input argument is outside the acceptable range.
Bad_TypeMismatch	See Table 166 for the description of this result code. Used to indicate that an input argument does not have the correct data type.

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 any *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 parameters are defined for *MonitoredItems* that tell the *Server* how the item is to be sampled, evaluated and reported. These parameters are the sampling interval, the monitoring mode, the filter and the queue parameter. Figure 15 illustrates these concepts.

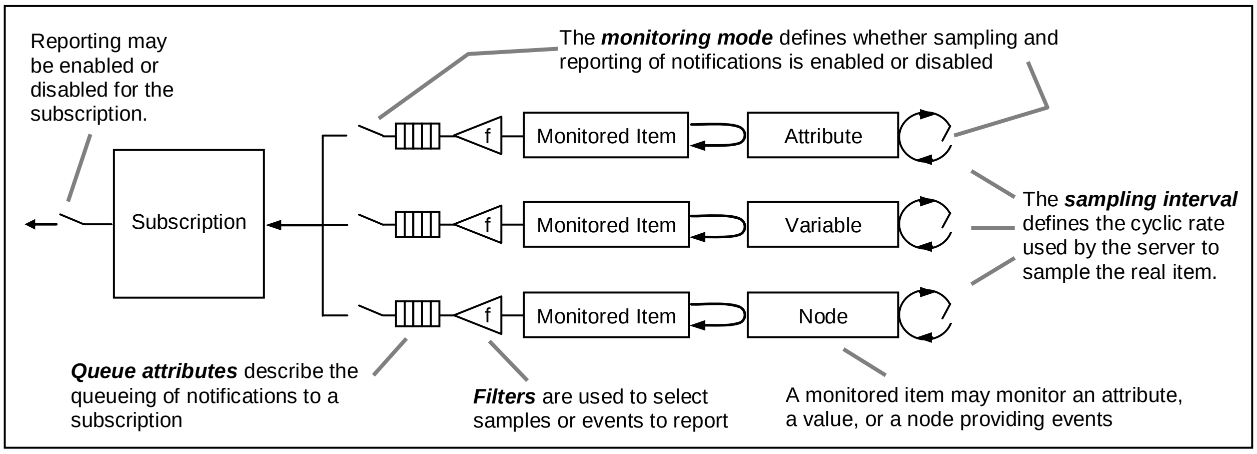
Reporting may be enabled or disabled for the subscription.
The monitoring mode defines whether sampling and reporting of notifications is enabled or disabled
Subscription
f
Monitored Item
Attribute
Variable
Node
The sampling interval defines the cyclic rate used by the server to sample the real item.
Queue attributes describe the queueing of notifications to a subscription
Filters are used to select samples or events to report
A monitored item may monitor an attribute, a value, or a node providing events

Figure 15 – MonitoredItem Model

Attributes, other than the *Value Attribute*, are only 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 7.16.2) and in 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 7.16.3) is used to determine if an *Event* received from the *Node* is sent to the *Client*. The filter also allows selecting fields of the *EventType* that will be contained in the *Event* such as *EventId*, *EventType*, *SourceNode*, *Time* and *Description*.

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 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.

A *Client* shall define a sampling interval of 0 if it subscibes for *Events*.

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. However the time between values returned to the *Client* shall be greater or equal to the sampling interval.

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 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 does not require sampling and 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 OPC UA *Server* provides access to a decoupled system and therefore has no knowledge of the data update logic. In this case, even though the OPC 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 *MinimumSamplingInterval Attribute* defined in Part 3. If the *Server* specifies a value for the

MinimumSamplingInterval Attribute it shall always return a *revisedSamplingInterval* that is equal or higher than the *MinimumSamplingInterval* if the *Client* subscribes to the *Value Attribute*.

Clients should also be aware that the sampling by the OPC 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 16.

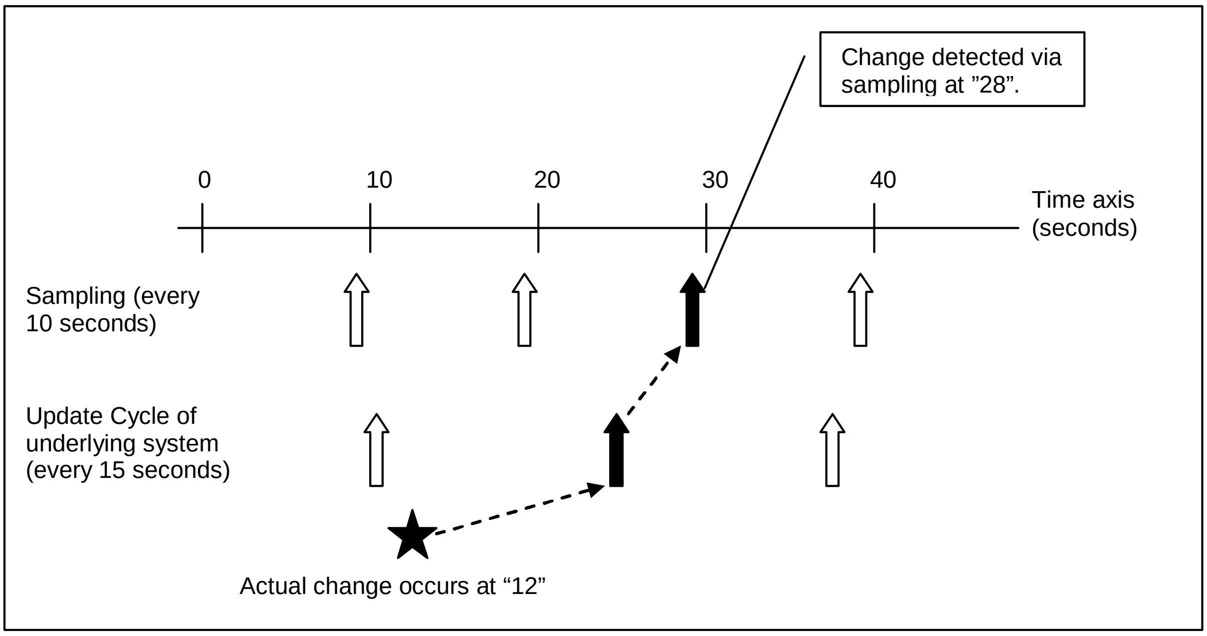
Change detected via sampling at ”28”.
0
10
20
30
40
Time axis (seconds)
Sampling (every 10 seconds)
Update Cycle of underlying system (every 15 seconds)
Actual change occurs at “12”

Figure 16 – Typical delay in change detection.

5.12.1.3 Monitoring mode

The monitoring mode parameter 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* parameter, such as its sampling interval.

When a *MonitoredItem* is enabled (i.e. when the *MonitoringMode* is changed from *DISABLED* to *SAMPLING* or *REPORTING*) or it is created in the enabled state, the *Server* shall report the first sample as soon as possible and the time of this sample becomes the starting point for the next 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 parameter 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 *DataChangeFilter* and the *AggregateFilter* are used when monitoring *Variable Values* 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 parameters

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* shall return *Notifications* for any particular item in the same order they are in the queue.

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 *DataChangeFilter* 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 may 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*. If *Events* are lost, an *Event* of the type *EventQueueOverflowEventType* is generated. This Event is generated when the first Event has to be discarded on a MonitoredItem subscribing for Events. It is put into the Queue of the MonitoredItem in addition to the size of the Queue defined for this MonitoredItem without discarding any other Event. If discardOldest is set to TRUE it is put at the beginning of the queue, otherwise at the end. An aggregating *Server* shall not pass on such an *Event*. It shall be handled like other connection error scenarios.

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 17 illustrates this concept.

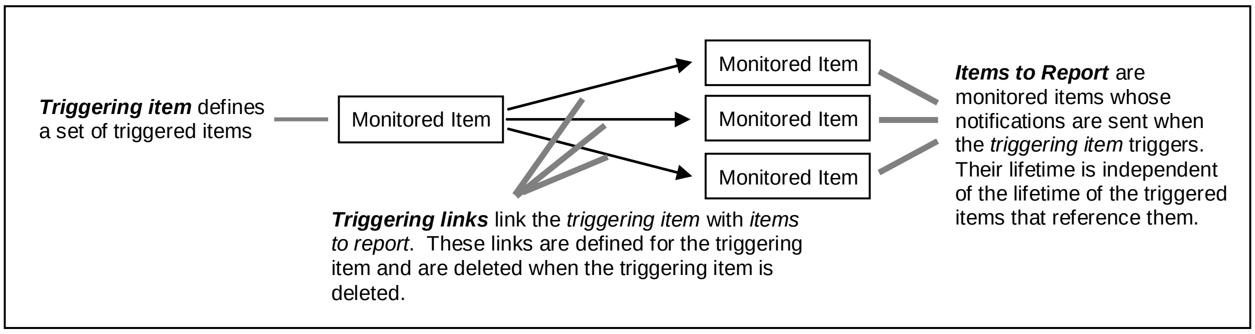
Triggering item defines a set of triggered items
Monitored Item
Monitored Item
Monitored Item
Monitored Item
Items to Report are monitored items whose notifications are sent when the *triggering item* triggers. Their lifetime is independent of the lifetime of the triggered items that reference them.
Triggering links link the *triggering item* with *items to report*. These links are defined for the triggering item and are deleted when the triggering item is deleted.

Figure 17 – 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:

d) 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.

e) 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.

f) 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.

g) 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*. 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 Type Specification Parts of this multi-part specification, such as Part 8. See Clause 4 of 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 shall succeed and the bad status Bad_NotReadable or Bad_UserAccessDenied shall 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* shall start sending data for the *MonitoredItem*.

The return diagnostic info setting in the request header of the *CreateMonitoredItems* or the last *ModifyMonitoredItems Service* is applied to the *Monitored Items* and is used as the diagnostic information settings when sending Notifications in the *Publish* response.

5.12.2.2 Parameters

Table 64 defines the parameters for the *Service*.

Table 64 – CreateMonitoredItems Service Parameters

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

5.12.2.3 Service results

Table 65 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 65 – CreateMonitoredItems Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 165 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.

5.12.2.4 StatusCodes

Table 66 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 66 – CreateMonitoredItems Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoringModeInvalid	See Table 166 for the description of this result code.
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_AttributeIdInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeNoData	See Table 166 for the description of this result code.
Bad_DataEncodingInvalid	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_MonitoredItemFilterInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterUnsupported	See Table 166 for the description of this result code.
Bad_FilterNotAllowed	See Table 165 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).

The return diagnostic info setting in the request header of the *CreateMonitoredItems* or the last *ModifyMonitoredItems Service* is applied to the *Monitored Items* and is used as the diagnostic information settings when sending Notifications in the *Publish* response.

5.12.3.2 Parameters

Table 67 defines the parameters for the *Service*.

Table 67 – ModifyMonitoredItems Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* used to qualify the *monitoredItemId* (see 7.13 for *IntegerId* definition).
timestampsToReturn	enum Timestamps ToReturn	An enumeration that specifies the timestamp *Attributes* to be transmitted for each *MonitoredItem* to be modified. The *TimestampsToReturn* enumeration is defined in 7.34. When monitoring *Events*, this applies only to *Event* fields that are of type *DataValue*.
itemsToModify []	MonitoredItemModifyRequest	The list of *MonitoredItems* to modify.
monitoredItemId	IntegerId	*Server*-assigned id for the *MonitoredItem*.
requestedParameters	Monitoring Parameters	The requested values for the monitoring parameters. The *MonitoringParameters* type is defined in 7.15.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	MonitoredItemModifyResult	List of results for the *MonitoredItems* to modify. The size and order of the list matches the size and order of the *itemsToModify* request parameter.
statusCode	StatusCode	*StatusCode* for the *MonitoredItem* to be modified (see 7.33 for *StatusCode* definition).
revisedSampling Interval	Duration	The actual sampling interval that the *Server* will use. 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. The Server shall always return a *revisedSamplingInterval* that is equal or higher than the *requestedSamplingInterval*. If the *requestedSamplingInterval* is higher than the maximum sampling interval supported by the *Server*, the maximum sampling interval is returned.
revisedQueueSize	Counter	The actual queue size that the *Server* will use.
filterResult	Extensible Parameter MonitoringFilterResult	Contains any revised parameter values or error results associated with the *MonitoringFilter* specified in the request. This parameter may be omitted if no errors occurred. The *MonitoringFilterResult* parameter type is an extensible parameter type specified in 7.16.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *MonitoredItems* to modify (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *itemsToModify* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.12.3.3 Service results

Table 68 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 68 – ModifyMonitoredItems Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 165 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.

5.12.3.4 StatusCodes

Table 69 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 69 – ModifyMonitoredItems Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterUnsupported	See Table 166 for the description of this result code.
Bad_FilterNotAllowed	See Table 165 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 causes all queued *Notifications* to be deleted.

5.12.4.2 Parameters

Table 70 defines the parameters for the *Service*.

Table 70 – SetMonitoringMode Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* used to qualify the *monitoredItemIds* (see 7.13 for *IntegerId* definition).
monitoringMode	enum MonitoringMode	The monitoring mode to be set for the *MonitoredItems*. The *MonitoringMode* enumeration is defined in 7.17.
monitoredItemIds []	IntegerId	List of *Server*-assigned ids for the *MonitoredItems*.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *MonitoredItems* to enable/disable (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *monitoredItemIds* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *MonitoredItems* to enable/disable (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *monitoredItemIds* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.12.4.3 Service results

Table 71 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 71 – SetMonitoringMode Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.
Bad_MonitoringModeInvalid	See Table 166 for the description of this result code.

5.12.4.4 StatusCodes

Table 72 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 72 – SetMonitoringMode Operation Level Result Codes

Value	Description
Bad_MonitoredItemIdInvalid	See Table 166 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 shall 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 73 defines the parameters for the *Service*.

Table 73 – SetTriggering Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* that contains the triggering item and the items to report (see 7.13 for *IntegerId* definition).
triggeringItemId	IntegerId	*Server*-assigned id for the *MonitoredItem* 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.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
addResults []	StatusCode	List of *StatusCodes* for the items to add (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *linksToAdd* parameter specified in the request.
addDiagnosticInfos []	Diagnostic Info	List of diagnostic information for the links to add (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *linksToAdd* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.
removeResults []	StatusCode	List of *StatusCodes* for the items to delete. The size and order of the list matches the size and order of the *linksToDelete* 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 *linksToDelete* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.12.5.3 Service results

Table 74 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in 7.33.

Table 74 – SetTriggering Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.
Bad_MonitoredItemIdInvalid	See Table 166 for the description of this result code.

5.12.5.4 StatusCodes

Table 75 defines values for the results parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 75 – SetTriggering Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 166 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 76 defines the parameters for the *Service*.

Table 76 – DeleteMonitoredItems Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* that contains the *MonitoredItems* to be deleted (see 7.13 for *IntegerId* definition).
monitoredItemIds []	IntegerId	List of *Server*-assigned ids for the *MonitoredItems* to be deleted.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *MonitoredItems* to delete (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *monitoredItemIds* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *MonitoredItems* to delete (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *monitoredItemIds* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.12.6.3 Service results

Table 77 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 77 – DeleteMonitoredItems Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.

5.12.6.4 StatusCodes

Table 78 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 78 – DeleteMonitoredItems Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 166 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 5.13.1.2.

h) *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 5.12.1 for a description of *MonitoredItems*).

i) *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*.

j) *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*.

k) 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.

l) *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*.

m) *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.

n) 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*.

o) *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* (5.13.2), the *Subscription* is closed. Closing the *Subscription* causes its *MonitoredItems* to be deleted. In addition the *Server* shall issue a *StatusChangeNotification notificationMessage* with the status code Bad_Timeout. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4.

p) *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 shall 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:

q) 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.

r) 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 expires 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 79.

Table 79 – Subscription States

State	Description
CLOSED	The *Subscription* has not yet been created or has terminated
CREATING	The *Subscription* is being created.
NORMAL	The *Subscription* is cyclically checking for *Notifications* from its *MonitoredItems*. The keep-alive counter is not used in this state.
LATE	The publishing timer has expired and there are *Notifications* available or a keep-alive *Message* is ready to be sent, but there are no *Publish* requests queued. When in this state, the next *Publish* request is processed when it is received. The keep-alive counter is not used in this state.
KEEPALIVE	The *Subscription* is cyclically checking for *Notifications* from its *MonitoredItems* or for the keep-alive counter to count down to 0 from its maximum.

The state table is described in Table 80. The following rules and conventions apply:

s) *Events* represent the receipt of *Service* requests and the occurrence internal *Events*, such as timer expirations.

t) *Service* requests *Events* may be accompanied by conditions that test *Service* parameter values. Parameter names begin with a lower case letter.

u) Internal *Events* may be accompanied by conditions that test state *Variable* values. State *Variables* are defined in 5.13.1.3. They begin with an upper case letter.

v) *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 5.13.1.4.

w) 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.

x) Actions are described by functions and state *Variable* manipulations.

y) The LifetimeTimerExpires *Event* is triggered when its corresponding counter reaches zero.

Table 80 – 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 *Publish* Request
		&&
		(
		PublishingEnabled == FALSE
		||
		(PublishingEnabled == TRUE
		&& MoreNotifications == FALSE)
		)
			ResetLifetimeCounter()
			DeleteAckedNotificationMsgs()
			EnqueuePublishingReq()
				NORMAL

5	NORMAL	Receive *Publish* 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 *Publish* Request
		&& PublishingEnabled == TRUE
		&& (NotificationsAvailable == TRUE
		|| MoreNotifications == TRUE)
			ResetLifetimeCounter()
			DeleteAckedNotificationMsgs()
			ReturnNotifications()
			MessageSent = TRUE
				NORMAL

11	LATE	Receive *Publish* 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 *Publish* Request	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	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()
			IssueStatusChangeNotification()
				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()
			IssueStatusChangeNotification()
				CLOSED

5.13.1.3 State Variables and parameters

The state *Variables* are defined alphabetically in Table 81.

Table 81 – State variables and parameters

State Variable	Description
MoreNotifications	A boolean value that is set to TRUE only by the CreateNotificationMsg() when there were too many *Notifications* for a single *NotificationMessage*.
LatePublishRequest	A boolean value that is set to TRUE to reflect that, the last time the publishing timer expired, there were no *Publish* requests queued.
LifetimeCounter	A value that contains the number of consecutive publishing timer expirations without *Client* activity before the *Subscription* is terminated.
MessageSent	A boolean value that is set to TRUE to mean that either a *NotificationMessage* or a keep-alive *Message* has been sent on the *Subscription*. It is a flag that is used to ensure that either a *NotificationMessage* or a keep-alive *Message* is sent out the first time the publishing timer expires.
NotificationsAvailable	A boolean value that is set to TRUE only when there is at least one *MonitoredItem* that is in the reporting mode and that has a *Notification* queued or there is at least one item to report whose triggering item has triggered and that has a *Notification* queued. The transition of this state *Variable* from FALSE to TRUE creates the "New *Notification* Queued" *Event* in the state table.
PublishingEnabled	The parameter that requests publishing to be enabled or disabled.
PublishingReqQueued	A boolean value that is set to TRUE only when there is a *Publish* request *Message* enqueued to the *Subscription*.
RequestedMessageFound	A boolean value that is set to TRUE only when the *Message* requested to be retransmitted was found in the retransmission queue.
SeqNum	The value that records the value of the sequence number used in *NotificationMessages*
SubscriptionAssignedToClient	A boolean value that is set to TRUE only when the *Subscription* requested to be deleted is assigned to the *Client* that issued the request. A *Subscription* is assigned to the *Client* 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 82.

Table 82 – Functions

State	Description
BindSession()	Bind the *Client Session* associated with the *Subscription* to the *Client Session* used to send the *Service* being processed. If this was the last *Subscription* bound to the previous *Client*, clear the *Publish* request queue of all *Publish* requests sent by the previous *Client* and return negative responses for each.
ClientValidated()	A boolean function that returns TRUE only when the *Client* that is submitting a TransferSubscriptions request is operating on behalf of the same user and supports the same *Profiles* as the *Client* of the previous *Session*.
CreateNotificationMsg()	Increment the SeqNum and create a *NotificationMessage* from the *MonitoredItems* assigned to the *Subscription*. Save the newly-created *NotificationMessage* in the retransmission queue. If all available *Notifications* can be sent in the *Publish* response, the MoreNotifications state *Variable* is set to FALSE. Otherwise, it is set to TRUE.
CreateSubscription()	Attempt to create the *Subscription*.
DeleteAckedNotificationMsgs()	Delete the *NotificationMessages* from the retransmission queue that were acknowledged by the request.
DeleteClientPublReqQueue()	Clear the *Publish* request queue for the *Client* that is sending the DeleteSubscriptions request, if there are no more *Subscriptions* assigned to that *Client*.
DeleteMonitoredItems()	Delete all *MonitoredItems* assigned to the *Subscription*
DequeuePublishReq()	Dequeue a publishing request in first-in first-out order. Validate if the publish request is still valid by checking the timeoutHint in the RequestHeader. If the request timed out, send a *Bad_Timeout* service result for the request and dequeue another publish request.
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()

IssueStatusChangeNotification()	Issue a *StatusChangeNotification notificationMessage* with a status code for the status change of the Subscription. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4. Bad_Timeout status code is used if the lifetime expires and Good_SubscriptionTransferred is used if the Subscriptions was transferred to another Session.
ResetKeepAliveCounter()	Reset the keep-alive counter to the maximum keep-alive count of the *Subscription*. The maximum keep-alive count is set by the *Client* when the *Subscription* is created and may be modified using the ModifySubscription *Service*.
ResetLifetimeCounter()	Reset the LifetimeCounter *Variable* to the value specified for the lifetime of a *Subscription* in the CreateSubscription *Service* (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 *Subscription* 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 83 defines the parameters for the *Service*.

Table 83 – CreateSubscription Service Parameters

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

5.13.2.3 Service results

Table 84 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 84 – 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 85 defines the parameters for the *Service*. Changes to the publishing interval become effective the next time the publishing timer expires.

Table 85 – ModifySubscription Service Parameters

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

5.13.3.3 Service results

Table 86 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 86 – ModifySubscription Service Result Codes

Symbolic Id	**Description**
Bad_SubscriptionIdInvalid	See Table 165 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 87 defines the parameters for the *Service*.

Table 87 – SetPublishingMode Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
publishingEnabled	Boolean	A *Boolean* parameter with the following values : TRUE publishing of *NotificationMessages* is enabled for the *Subscription*. FALSE publishing of *NotificationMessages* is disabled for the *Subscription*. The value of this parameter does not affect the value of the monitoring mode *Attribute* of *MonitoredItems*. Setting this value to FALSE does not discontinue the sending of keep-alive *Messages*.
subscriptionIds []	IntegerId	List of *Server*-assigned identifiers for the *Subscriptions* to enable or disable (see 7.13 for *IntegerId* definition).
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *Subscriptions* to enable/disable (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *subscriptionIds* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Subscriptions* to enable/disable (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *subscriptionIds* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.13.4.3 Service results

Table 88 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 88 – SetPublishingMode Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.13.4.4 StatusCodes

Table 89 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 89 – SetPublishingMode Operation Level Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 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*. 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 shall accept more queued *Publish* requests than created Subscriptions. It is expected that a *Server* supports several *Publish* requests per *Subscription*. When a *Server* receives a new *Publish* request that exceeds its limit it shall dequeue the oldest *Publish* request and return a response with the result set to *Bad_TooManyPublishRequests*. If a *Client* receives this *Service* result for a *Publish* request it shall not issue another *Publish* request before one of its outstanding *Publish* requests is returned from the Server.

Clients can limit the size of *Publish* reponses with the *maxNotificationsPerPublish* parameter passed to the *CreateSubscription Service*. However, this could still result in a message that is too large for the *Client* or *Server* to process. In this situation, the *Client* will find that either the *SecureChannel* goes into a fault state and needs to be re-established or the *Publish* response returns an error and calling the *Republish Service* also returns an error. If either situation occurs then the *Client* will have to adjust its message processing limits or the parameters for the *Subscription and/or MonitoredItems*.

The return diagnostic info setting in the request header of the *CreateMonitoredItems* or the last *ModifyMonitoredItems Service* is applied to the *Monitored Items* and is used as the diagnostic information settings when sending Notifications in the *Publish* response.

5.13.5.2 Parameters

Table 90 defines the parameters for the *Service*.

Table 90 – Publish Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscription Acknowledgements []	Subscription Acknowledgement	The list of acknowledgements for one or more *Subscriptions*. This list may contain multiple acknowledgements for the same *Subscription* (multiple entries with the same *subscriptionId*).
subscriptionId	IntegerId	The *Server* assigned identifier for a *Subscription* (see 7.13 for *IntegerId* definition).
sequenceNumber	Counter	The sequence number being acknowledged (see 7.5 for *Counter* definition). The *Server* may delete the *Message* with this sequence number from its retransmission queue.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* for which *Notifications* are being returned (see 7.13 for *IntegerId* definition). The value 0 is used to indicate that there were no *Subscriptions* defined for which a response could be sent.
availableSequence Numbers []	Counter	A list of sequence number ranges that identify unacknowledged *NotificationMessages* that are available for retransmission from the *Subscription*'s retransmission queue. This list is prepared after processing the acknowledgements in the request (see 7.5 for *Counter* definition).
moreNotifications	Boolean	A *Boolean* parameter with the following values : TRUE the number of *Notifications* that were ready to be sent could not be sent in a single response. FALSE all *Notifications* that were ready are included in the response.
notificationMessage	Notification Message	The *NotificationMessage* that contains the list of *Notifications*. The *NotificationMessage* parameter type is specified in 7.20.
results []	StatusCode	List of results for the acknowledgements (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *subscriptionAcknowledgements* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the acknowledgements (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *subscriptionAcknowledgements* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.13.5.3 Service results

Table 91 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 91 – 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 92 defines values for the *acknowledgeResults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 92 – Publish Operation Level Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.
Bad_SequenceNumberUnknown	The sequence number is unknown to the server.

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 5.13.1.2 for the detail description of the behaviour of this *Service*.

5.13.6.2 Parameters

Table 93 defines the parameters for the *Service*.

Table 93 – Republish Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server* assigned identifier for the *Subscription* to be republished (see 7.13 for *IntegerId* definition).
retransmitSequence Number	Counter	The sequence number of a specific *NotificationMessage* to be republished (see 7.5 for *Counter* definition).
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
notificationMessage	Notification Message	The requested *NotificationMessage*. The *NotificationMessage* parameter type is specified in 7.18.

5.13.6.3 Service results

Table 94 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 94 – Republish Service Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 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 *authenticationToken* contained in the request header identifies the *Session* to which the *Subscription* and *MonitoredItems* shall be transferred. The *Server* shall 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* shall issue a *StatusChangeNotification notificationMessage* with the status code Good_SubscriptionTransferred. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4.

5.13.7.2 Parameters

Table 95 defines the parameters for the *Service*.

Table 95 – TransferSubscriptions Service Parameters

Name	**Type**	**Description**
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionIds []	IntegerId	List of identifiers for the *Subscriptions* to be transferred to the new *Client* (see 7.13 for *IntegerId* definition). These identifiers are transferred from the primary *Client* to a backup *Client* via external mechanisms.
sendInitialValues	Boolean	A *Boolean* parameter with the following values : TRUE the first Publish response after the TransferSubscriptions call shall contain the current values of all Monitored Items in the Subscription where the Monitoring Mode is set to Reporting. FALSE the first Publish response after the TransferSubscriptions call shall contain only the value changes since the last Publish response was sent. This parameter only applies to MonitoredItems used for monitoring Attribute changes.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	TransferResult	List of results for the *Subscriptions* to transfer. The size and order of the list matches the size and order of the *subscriptionIds* request parameter.
statusCode	StatusCode	*StatusCode* for each *Subscription* to be transferred (see 7.33 for *StatusCode* definition).
availableSequence Numbers []	Counter	A list of sequence number ranges that identify *NotificationMessages* that are in the *Subscription*'s retransmission queue. This parameter is null if the transfer of the *Subscription* failed. The *Counter* type is defined in 7.5.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Subscriptions* to transfer (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *subscriptionIds* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.13.7.3 Service results

Table 96 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 96 – TransferSubscriptions Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code. The *Client* of the current *Session* is not operating on behalf of the same user as the *Session* that owns the *Subscription*.
Bad_InsufficientClientProfile	The *Client* of the current *Session* does not support one or more *Profiles* that are necessary for the *Subscription*.

5.13.7.4 StatusCodes

Table 97 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 97 – TransferSubscriptions Operation Level Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 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 98 defines the parameters for the *Service*.

Table 98 – DeleteSubscriptions Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionIds []	IntegerId	The *Server*-assigned identifier for the *Subscription* (see 7.13 for *IntegerId* definition).
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *Subscriptions* to delete (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *subscriptionIds* request parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Subscriptions* to delete (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *subscriptionIds* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.13.8.3 Service results

Table 99 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 99 – DeleteSubscriptions Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.

5.13.8.4 StatusCodes

Table 100 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 100 – DeleteSubscriptions Operation Level Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.

6 Service behaviours

6.1 Security

6.1.1 Overview

The OPC UA *Services* define a number of mechanisms to meet the security requirements outlined in Part 2. This section describes a number of important security-related procedures that *OPC UA Applications* shall follow.

6.1.2 Obtaining and Installing an Application Instance Certificate

All *OPC UA Applications* require an application instance certificate which shall 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 URI of the application instance;
- 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 shall 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 shall 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 18.

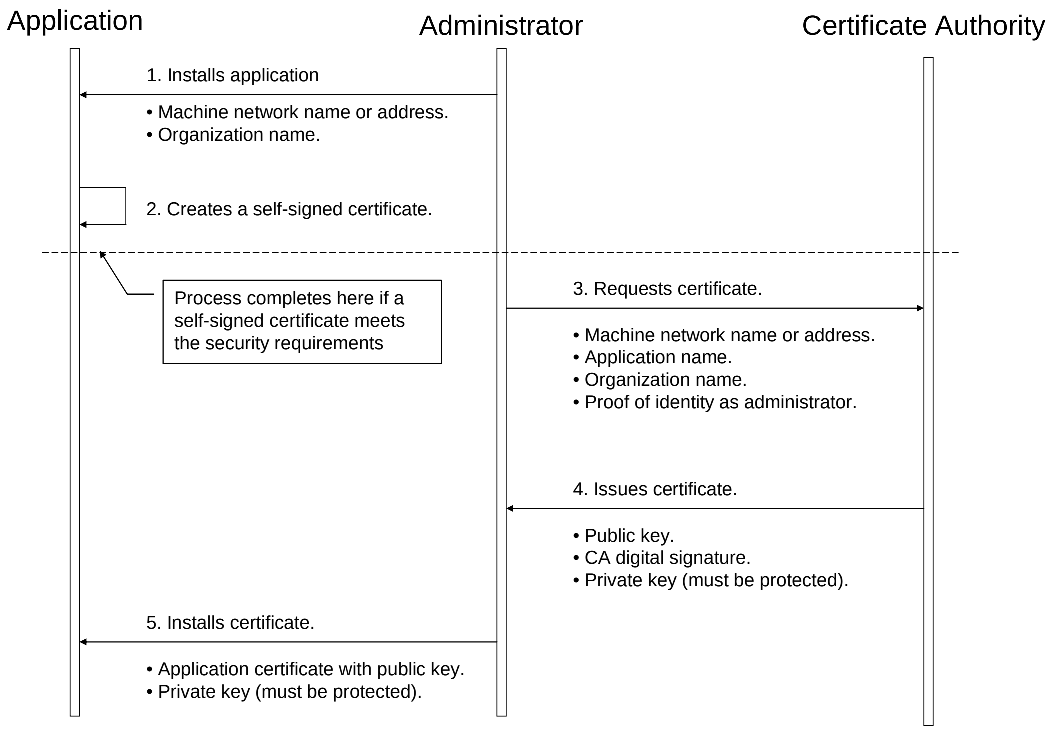
Application
Administrator
Certificate Authority
1. Installs application
• Machine network name or address.
• Organization name.
2. Creates a self-signed certificate.
Process completes here if a self-signed certificate meets the security requirements
3. Requests certificate.
• Machine network name or address.
• Application name.
• Organization name.
• Proof of identity as administrator.
4. Issues certificate.
• Public key.
• CA digital signature.
• Private key (must be protected).
5. Installs certificate.
• Application certificate with public key.
• Private key (must be protected).

Figure 18 – Obtaining and Installing an Application Instance Certificate

The figure above illustrates the interactions between the *Application*, the *Administrator* and the *CertificateAuthority*. The *Application* is as *OPC UA Application* installed on a single machine. The *Administrator* is the person responsible for managing the machine and the *OPC UA Application*. The *CertificateAuthority* is an entity that can issue digital certificates that meet the requirements of the organisation deploying the *OPC 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 shall always create a default self-signed certificate during the installation process. Every *OPC UA Application* shall 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 installation program for the application would do them automatically and only prompt the *Administrator* to provide information about the application instance being installed.

6.1.3 Obtaining and Installing a Software Certificate

All *OPC UA Applications* may have one or more software certificates that are issued by certification authorities and specify the profiles that the application supports. These software certificates contain the following information:

- The name of the vendor responsible for the product;
- The name of the product;
- The URI for the product;
- The software version and build number;
- The vendor product certificate;
- 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 certifying authority;
- A digital signature created by the certifying authority.

The product vendor is responsible for completing the certification process and requesting software certificates from the certification authorities. The vendor shall 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 certificates use by unauthorized applications.

Vendors may choose to assign their own *Certificate* to the application (called the Vendor Product Certificate). This Certificate would then be incorporated in the *Software Certificate*. The private key for the vendor product certificate is managed by the Vendor.

The steps involved in acquiring and installing a software certificate from a certificate authority are shown in Figure 19.

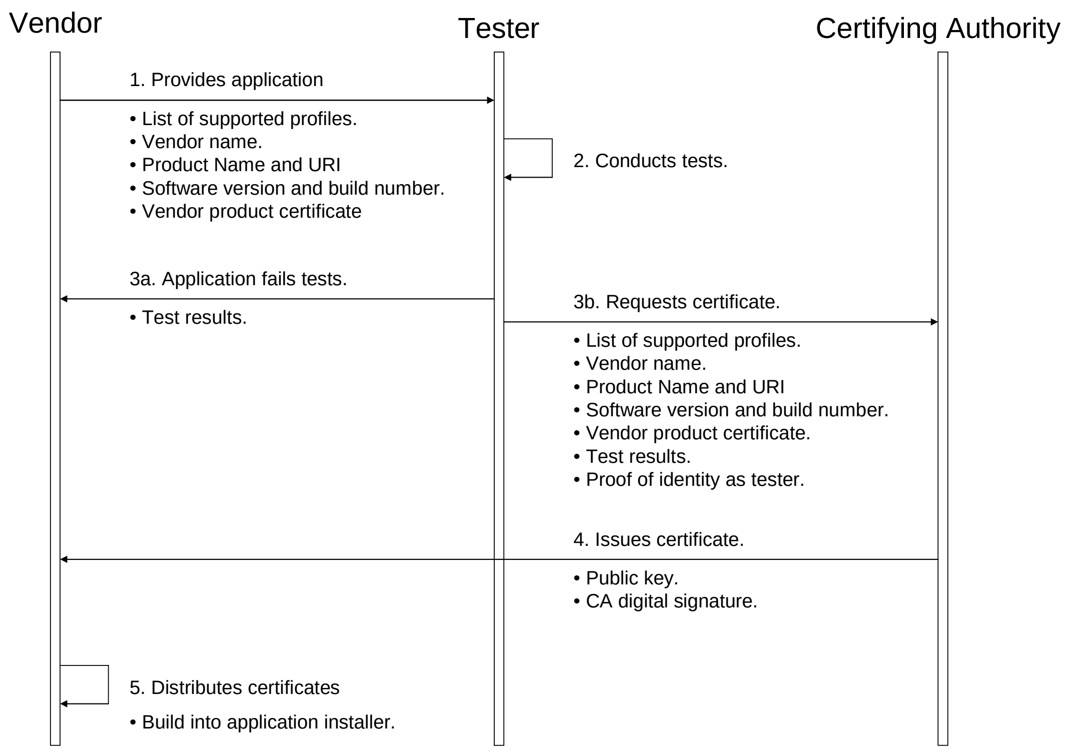
Vendor
Tester
Certifying Authority
1. Provides application
• List of supported profiles.
• Vendor name.
• Product Name and URI
• Software version and build number.
• Vendor product certificate
2. Conducts tests.
3a. Application fails tests.
• Test results.
3b. Requests certificate.
• List of supported profiles.
• Vendor name.
• Product Name and URI
• Software version and build number.
• Vendor product certificate.
• Test results.
• Proof of identity as tester.
4. Issues certificate.
• Public key.
• CA digital signature.
5. Distributes certificates
• Build into application installer.

Figure 19 – Obtaining and Installing a Software Certificate

The figure above illustrates the interactions between the *Vendor*, the *Tester* and the *CertifyingAuthority*. The *Vendor* is the organisation responsible for the *OPC UA Application*. The *Tester* may be the vendor (for self-certification) or it may be a third-party testing facility. The *CertifyingAuthority* is a PKI certificate authority managed by the organisation that created the *OPC UA Application* profiles and certification programmes.

The *CertifyingAuthority* will issue certificates only to people it trusts. For that reason, the *Tester* shall provide proof of identity before the *CertifyingAuthority* will issue a certificate.

6.1.4 Determining if a Certificate is Trusted

Applications shall never communicate with another application that they do not trust. An *Application* decides if another application is trusted by checking whether the application instance *Certificate* for the other application is trusted. Applications shall rely on lists of Certificates provided by the Administrator to determine trust. There are two separate lists: a list of trusted Applications and a list of trusted Certificate Authorities (CAs). If an application is not directly trusted (i.e. its Certificate is not in the list of trusted Applications) then the Application shall build a chain of Certificates back to a trusted CA.

When building a chain each Certificate in the chain shall be validated. If any validation error occurs then the trust check fails. Some validation errors are non-critical which means they can be suppressed by a user of an Application with the appropriate privileges. Suppressed validation errors are always reported via auditing (i.e. an appropriate Audit event is raised).

Building a trust chain requires access to all Certificates in the chain. These Certificates may be stored locally or they may be provided with the application Certificate. Processing stops if a CA certificate cannot be found or if it is in the list of trusted CAs.

Table 101 specifies the steps used to validate a Certificate in the order that they should be followed. These steps are repeated for each certificate in the chain until a trusted certificate is found. Each validation step has a unique error status and audit event type that shall be reported if the check fails. The audit event is in addition to any audit event that was generated for the particular *Service* that was invoked. The *Service* audit event in its message text shall include the audit eventID of the *AuditCertifcateEvent* (for more details see 6.2). Processing halts if an error occurs unless it is non-critical and it has been suppressed.

Application instance certificates shall not be used in a *Client* or *Server* until they have been evaluated and marked as trusted. This can happen automatically by a PKI trust chain or in an offline manner where the *Certificate* is marked as trusted by an administrator after evaluation.

Table 101 – Certificate Validation Steps

Step	Error/AuditEvent	Description
Certificate Structure	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	The certificate structure is verified. This error may not be suppressed.
Validity Period	Bad_CertificateTimeInvalid Bad_CertificateIssuerTimeInvalid AuditCertificateExpiredEventType	The current time shall be after the start of the validity period and before the end. This error may be suppressed.
Host Name	Bad_CertificateHostNameInvalid AuditCertificateDataMismatchEventType	The HostName in the URL used to connect to the Server shall be the same as one of the HostNames specified in the Certificate. This error may be suppressed.
URI	Bad_CertificateUriInvalid AuditCertificateDataMismatchEventType	Application and Software Certificates contain an application or product URI that shall match the URI specified in the *ApplicationDescription* provided with the Certificate. This check is skipped for CA Certificates. This error may not be suppressed. The *gatewayServerUri* is used to validate an *Application Certificate* when connecting to a *Gateway Server* (See 7.1).
Certificate Usage	Bad_CertificateUseNotAllowed Bad_CertificateIssuerUseNotAllowed AuditCertificateMismatchEventType	Each Certificate has a set of uses for the Certificate. These uses shall match use requested for the Certificate (i.e. Application, Software or CA), This error may be suppressed unless the Certificate indicates that the usage is mandatory.
Trust List Check	None	No further checks are required if the Certificate is in the Trust List. The Administrator shall completely validate any Certificate before placing it in the Trust List.
Find Issuer Certificate	Bad_CertificateUntrusted AuditCertificateUntrustedEventType	A Certificate cannot be trusted if the Issuer Certificate is unknown. A self-signed Certificate is its own issuer.
Signature	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	A Certificate with an invalid signature shall always be rejected.
Find Revocation List	Bad_CertificateRevocationUnknown Bad_CertificateIssuerRevocationUnknown AuditCertificateRevokedEventType	Each CA Certificate may have a revocation list. This check fails if this list is not available (i.e. a network interruption prevents the Application from accessing the list). No error is reported if the Administrator disables revocation checks for a CA Certificate. This error may be suppressed.
Revocation Check	Bad_CertificateRevoked Bad_CertificateIssuerRevoked AuditCertificateRevokedEventType	The Certificate has been revoked and may not be used. This error may not be suppressed.

Certificates are usually placed in a central location called a CertificateStore. Figure 20 illustrates the interactions between the *Application*, the *Administrator* and the *CertificateStore*. The *CertificateStore* could be on the local machine or in some central server. The exact mechanisms used to access the *CertificateStore* depend on the application and PKI environment set up by the *Administrator*.

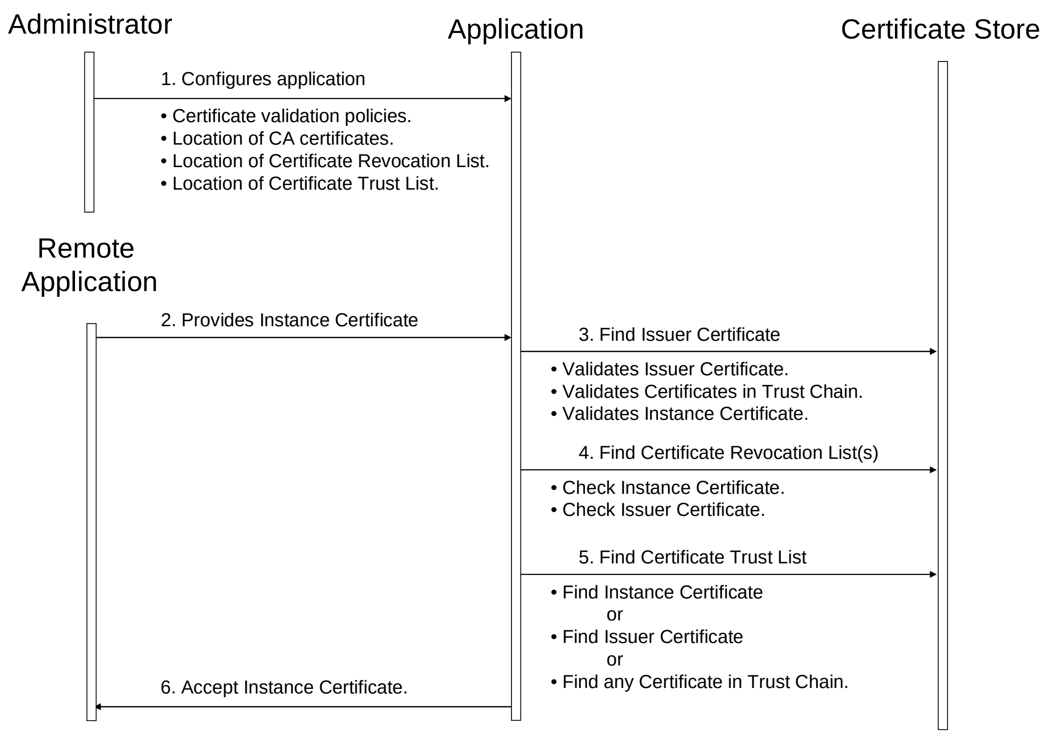
Administrator
Application
Certificate Store
1. Configures application
• Certificate validation policies.
• Location of CA certificates.
• Location of Certificate Revocation List.
• Location of Certificate Trust List.
Remote Application
2. Provides Instance Certificate
3. Find Issuer Certificate
• Validates Issuer Certificate.
• Validates Certificates in Trust Chain.
• Validates Instance Certificate.
4. Find Certificate Revocation List(s)
• Check Instance Certificate.
• Check Issuer Certificate.
5. Find Certificate Trust List
• Find Instance Certificate
or
• Find Issuer Certificate
or
• Find any Certificate in Trust Chain.
6. Accept Instance Certificate.

Figure 20 – Determining if a Application Instance Certificate is Trusted

6.1.5 Validating a Software Certificate

OPC UA Applications shall validate the *SoftwareCertificates* provided by the applications that they communicate with.

A *SoftwareCertificate* is valid if:

- The signature on the *SoftwareCertificate* is valid;
- The *SoftwareCertificate* has passed its issue date and it has not expired;
- The *SoftwareCertificate* has not been revoked by the issuer;
- The issuer *Certificate* is valid and has not been revoked by the CA that issued it;

The steps used to validate *SoftwareCertificates* are almost the same as the steps described for application instance *Certificates* in 6.1.4. The only difference is *SoftwareCertificates* are not checked against the *TrustList* for the *Application.*

SoftwareCertificates always contain an application *Certificate* which is owned by the vendor that distributes the application. The application *Certificate* may also be in the *Certificate* chain used to issue application instance *Certificates*. For this reason, *OPC UA Applications* may reject *SoftwareCertificates* provided by applications if the application *Certificate* is not part of the *Certificate* chain for the application instance *Certificate*. *OPC UA Applications* should allow *Administrators* to require this behaviour. Profiles defined in Part 7 may further specify expected *Certificate* handling behaviors.

6.1.6 Creating a SecureChannel

All *OPC UA Applications* shall 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 6.1.2 may be used for this purpose.

The steps involved in establishing a *SecureChannel* are shown in Figure 21.

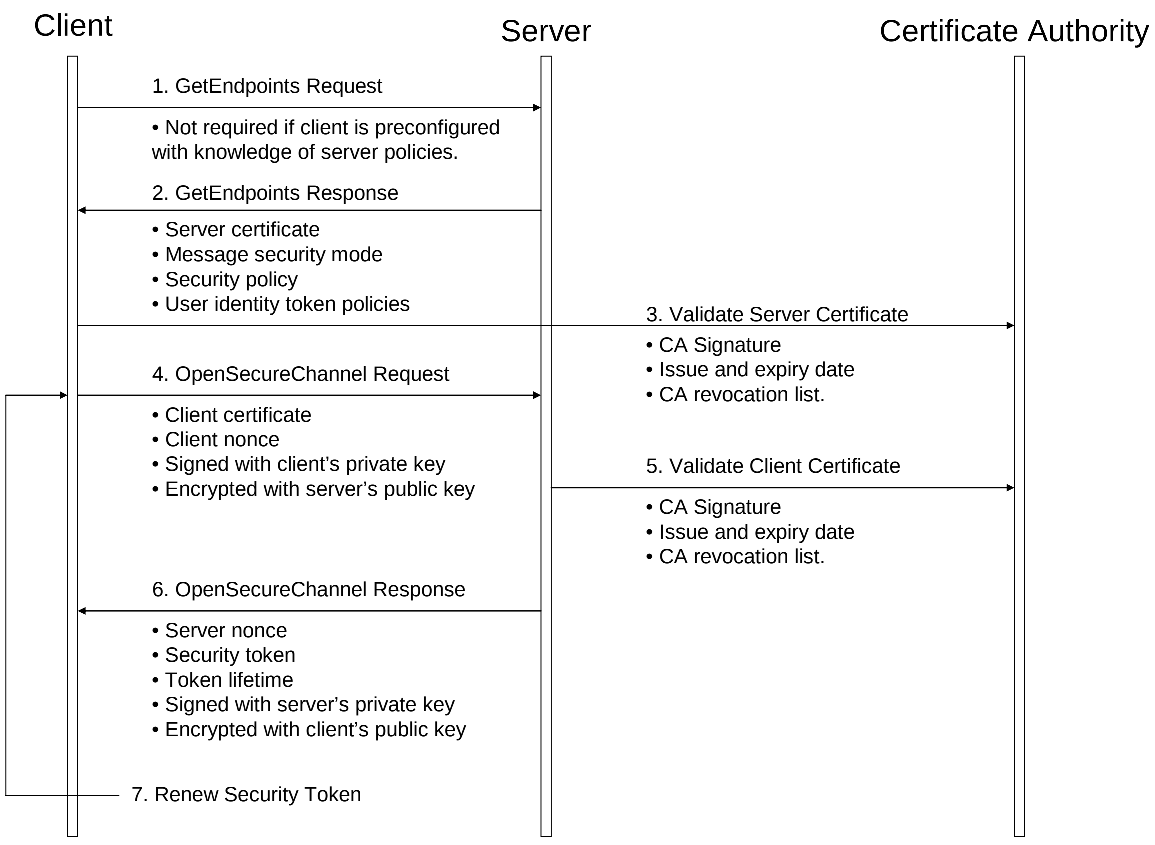
Client
Server
Certificate Authority
1. GetEndpoints Request
• Not required if client is preconfigured with knowledge of server policies.
2. GetEndpoints Response
• Server certificate
• Message security mode
• Security policy
• User identity token policies
3. Validate Server Certificate
• CA Signature
• Issue and expiry date
• CA revocation list.
4. OpenSecureChannel Request
• Client certificate
• Client nonce
• Signed with client's private key
• Encrypted with server's public key
5. Validate Client Certificate
• CA Signature
• Issue and expiry date
• CA revocation list.
6. OpenSecureChannel Response
• Server nonce
• Security token
• Token lifetime
• Signed with server's private key
• Encrypted with client's public key
7. Renew Security Token

Figure 21 – 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* shall 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. Any certificate shall be rejected if it is not in a trust list provided by the administrator.

Both the *Client* and *Server* shall 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 *OPC UA Application* instance certificates. *OPC UA Applications* shall 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 the CA can add the certificate to the Certificate Revocation List (CRL) for the issuer *Certificate*. Administrators may save a local copy of the CRL for each issuer *Certificate* 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 are two security risks which a *Client* shall be aware of when using the *GetEndpoints Service*. The first could come from a rogue *Discovery Server* that tries to direct the *Client* to a rogue *Server*. For this reason the *Client* shall verify that the *ServerCertificate* in the *EndpointDescription* is a trusted *Certificate* before it calls *CreateSession*.

The second security risk comes from third party that alters the contents of the *EndpointDescriptions* as they are transferred over the network back to the *Client*. The Client protects against this by comparing the list of *EndpointDescriptions* returned from the *GetEndpoints Service* with list returned in the *CreateSession* response.

The exact mechanisms for using the security token to sign and encrypt *Messages* exchanged over the *SecureChannel* are described in Part 6. The process for renewing tokens is also described in detail in 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, *OPC UA Applications* will need to exchange their application instance certificates when creating a *Session*.

6.1.7 Creating a Session

Once an OPC UA *Client* has established a *SecureChannel* with a *Server* it can create an OPC UA *Session*.

The steps involved in establishing a *Session* are shown in Figure 22.

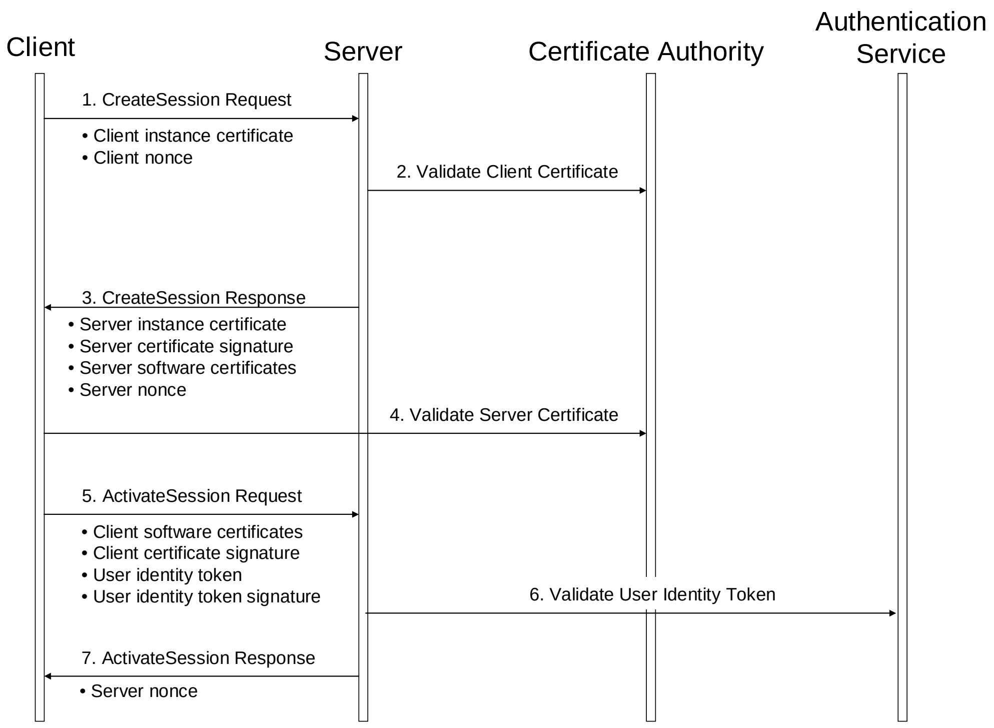
Client
Server
Certificate Authority
Authentication Service
1. CreateSession Request
• Client instance certificate
• Client nonce
2. Validate Client Certificate
3. CreateSession Response
• Server instance certificate
• Server certificate signature
• Server software certificates
• Server nonce
4. Validate Server Certificate
5. ActivateSession Request
• Client software certificates
• Client certificate signature
• User identity token
• User identity token signature
6. Validate User Identity Token
7. ActivateSession Response
• Server nonce

Figure 22 – 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 shall validate the application certificates using the CA public key that the administrator shall 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* shall 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 5.6.2. Similarly, the *Client* shall prove possession of some types of user identity tokens by creating signatures with the secret associated with the token.

6.1.8 Impersonating a User

Once an OPC 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 23.

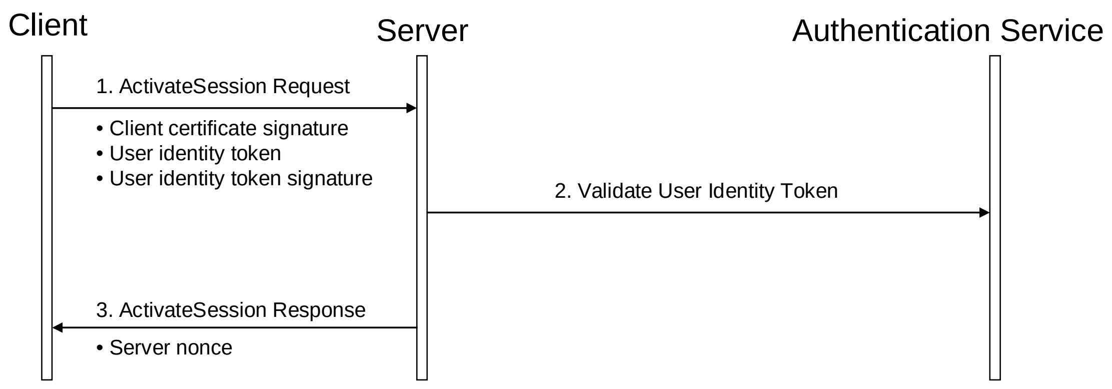
Client
Server
Authentication Service
1. ActivateSession Request
• Client certificate signature
• User identity token
• User identity token signature
2. Validate User Identity Token
3. ActivateSession Response
• Server nonce

Figure 23 – Impersonating a User

6.2 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 Part 2. This section describes what is expected of an OPC UA *Server* and *Client* with respect to auditing and it details the audit requirements for each service set. Auditing can be accomplished using one or both of the following methods:

1. The *OPC UA Application* that generates the audit event can log the audit entry in a log file or other storage location.

2. The OPC UA *Server* that generates the audit event can publish the audit event using the OPC UA event mechanism. This allows an external OPC UA *Client* to subscribe to and log the audit entries to a log file or other storage location.

6.2.2 General audit logs

Each OPC 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 shall provide the audit log entries via *Event Messages*. The *AuditEventType* and its sub-types are defined in Part 3. An audit *Event Message* also includes the audit record Id. The details of the *AuditEventType* and its sub types are defined in Part 5. A *Server* that is an aggregating *Server* that supports auditing shall 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 Discovery Service Set

This *Service Set* can be separated into two groups: Services that are called by OPC UA *Clients* and *Services* that are invoked by OPC UA *Servers*. The *FindServers* and *GetEndpoints Services* that are called by OPC UA *Clients* may generate audit entries for failed *Service* invocations. The *RegisterServer* Service that is invoked by OPC UA *Servers* shall generate audit entries for all new registrations and for failed *Service* invocations. These audit entries shall include the *Server* URI, *Server* names, *Discovery* URIs and isOnline status. Audit entries should not be generated for *RegisterServer* invocation that do not cause changes to the registered *Servers*.

6.2.5 Auditing for SecureChannel Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for failed service invocations and for successful invocation of the *OpenSecureChannel* and *CloseSecureChannel* Services. The *Client* generated audit entries should be setup prior to the actual call, allowing the correct audit record Id to be provided. The *OpenSecureChannel* Service shall generate an audit *Event* of type *AuditOpenSecureChannelEventType* or a subtype of it. The *CloseSecureChannel* service shall generate an audit *Event* of type *AuditCloseSecureChannelEventType* or a subtype of it. Both of these *Event* types are sub-types of the *AuditChannelEventType*. See Part 5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure cases the *Message* for *Events* of this type should include a description of why the service failed. This description should be more detailed then what was returned to the client. From a security point of view a client only needs to know that it failed, but from an *Auditing* point of view the exact details of the failure need to be known. In the case of *Certificate* validation errors the description should include the *AuditEventId* of the specific *AuditCertificateEvent* that was generated to report the certificate error. The *AuditCertifcateEvent* shall also contain the detailed certificate validation error. 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 shall be made available to the *Server* and the *Server* shall report them.

6.2.6 Auditing for Session Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditSessionEventType* or a sub-type of it. In particular, they shall generate the base *EventType* or the appropriate sub-type, depending on the service that was invoked. The *CreateSession* service shall generate *AuditCreateSessionEventType* events or sub-types of it. The *ActivateSession* service shall generate *AuditActivateSessionType* events or sub-types of it. When the *ActivateSession Service* is called to change the user identity then the server shall generate *AuditImpersonateUserEventType* events or sub-types of it. The CloseSession service shall generate the base *EventType* of *AuditSessionEventType* or sub-types of it. See 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.

This *Service Set* shall also generate additional audit events in the cases when *Certificate* validation errors occur. These audit events are generated in addition to the *AuditSessionEvents*.

For *Clients*, that support auditing, accessing the services in the *Session Service Set* shall 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.

6.2.7 Auditing for NodeManagement Service Set

All Services in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditNodeManagementEventType* or sub-types of it. See 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* shall 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.

6.2.8 Auditing for Attribute Service Set

The *Write* or *HistoryUpdate* Services in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditUpdateEventType* or sub-types of it. In particular the *Write Service* shall generate an audit event of type *AuditWriteUpdateEventType* or a sub-type of it. The *HistoryUpdate Service* shall generate an audit *Event* of type *AuditHistoryUpdateEvent* or a subtype of it. Three subtypes of *AuditHistoryUpdateEvent* are defined as *AuditHistoryEventUpdateEventType*, *AuditHistoryValueUpdateEventType* and *AuditHistoryDeleteEventType*. The subtype depends on the type of operation being performed, historical event update, historical data value update or a historical delete. See 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* or a subtype of it. See Part 5 for the detailed assignment of the *SourceNode*, *SourceName* and additional parameters. 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* shall 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.

6.2.9 Auditing for Method Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate 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). These method calls shall generate an audit *Event* of type *AuditUpdateMethodEventType* or sub-types of it. Methods that do not modify the address space, write values or modify the state of the system may generate events. See Part 5 for the detailed assignment of the *SourceNode*, *SourceName* and additional parameters.

For *Clients* that support auditing, accessing the *Method Service Set* shall 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.

6.2.10 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* shall generate an audit *Event* of type *AuditSessionEventType* or sub-types of it for both successful and failed *Service* invocations. See 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* shall 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 standardized 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 not 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 shall 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 another *Server* will take over the *Session* and *Subscriptions* of the *Client* in the case of failover. Failover may require a transport layer reconnect of the *Client* but the *Endpoint* URL of the *Server* shall not change.

Figure 24 shows a typical transparent redundancy setup.

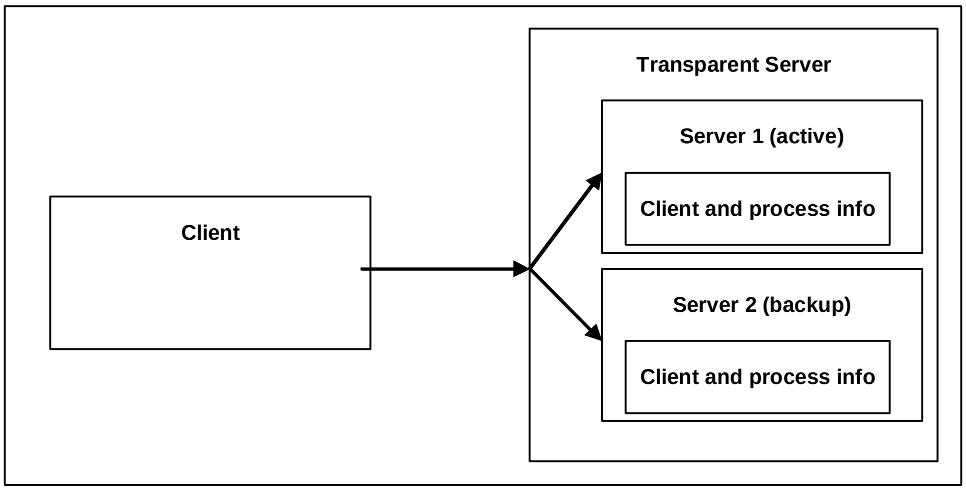

Figure 24 – Transparent Redundancy setup

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 25 shows a typical non-transparent redundancy setup.

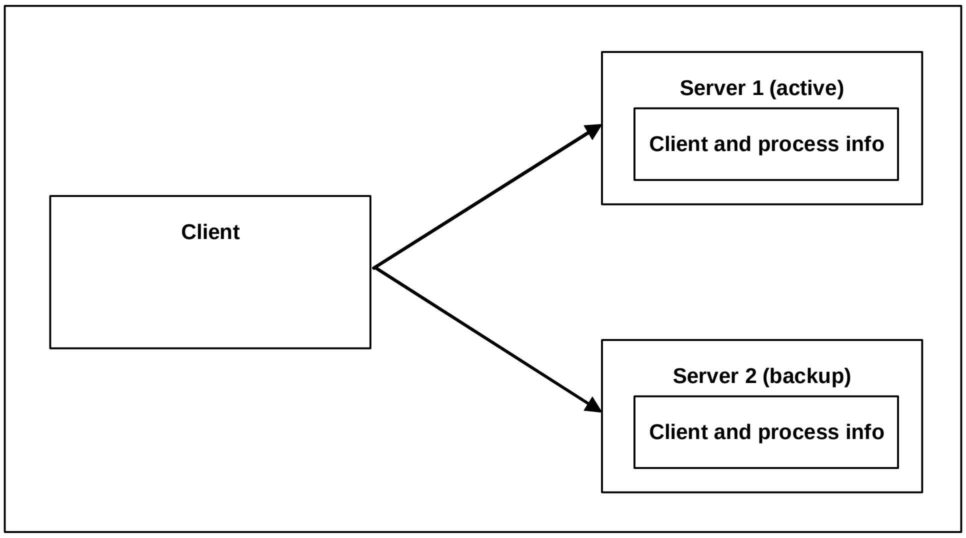

Figure 25 – Non-Transparent Redundancy setup

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 102 defines the list of failover actions.

Table 102 – Redundancy failover actions

Failover mode	**COLD**	**WARM**	**HOT**
On initial connection:			
Connect to more than one OPC UA *Server*.		X	X
Creating *Subscriptions* and adding monitored items to them.		X	X
Activating sampling on the *Subscriptions*			X
At Failover:			
Connect to backup OPC UA *Server*	X		
Creating *Subscriptions* and adding monitored items.	X		
Activating sampling on the *Subscriptions*.	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 shall 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 26 shows the difference between *Client* proxy and *Server* proxy redundancy.

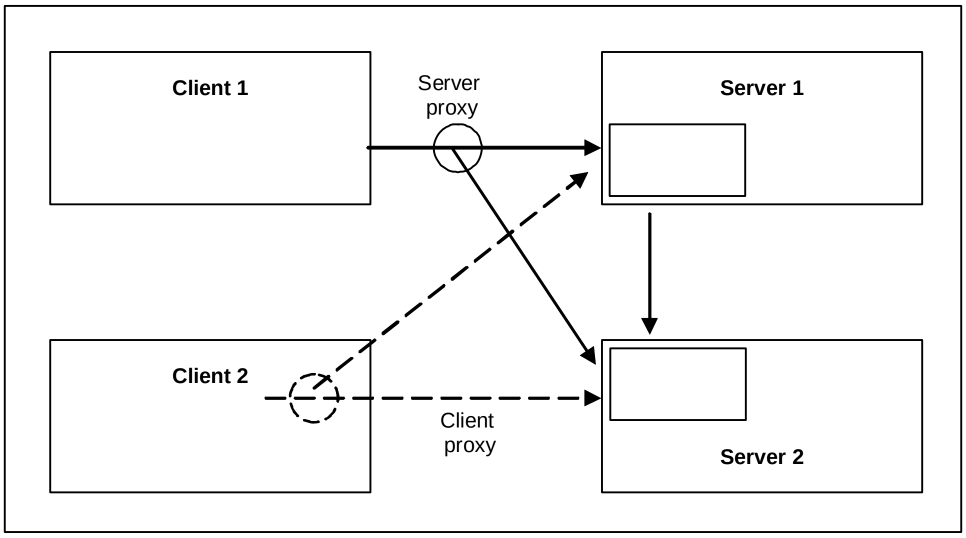

Figure 26 – Redundancy mode

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* shall 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 ApplicationDescription

The components of this parameter are defined in Table 103.

Table 103 – ApplicationDescription

Name	Type	Description
ApplicationDescription	structure	Specifies an application that is available.
applicationUri	String	The globally unique identifier for the application instance.
productUri	String	The globally unique identifier for the product.
applicationName	LocalizedText	A localized descriptive name for the application.
applicationType	Enum ApplicationType	The type of application. This value is an enumeration with one of the following values: SERVER_0 The application is a *Server*. CLIENT_1 The application is a *Client*. CLIENTANDSERVER_2 The application is a *Client* and a *Server*. DISCOVERYSERVER_3 The application is a *DiscoveryServer.*
gatewayServerUri	String	A URI that identifies the *Gateway Server* associated with the *discoveryUrls*. This value is not specified if the *Server* can be accessed directly. This field is not used if the *applicationType* is CLIENT_1.
discoveryProfileUri	String	A URI that identifies the discovery profile supported by the URLs provided. This field is not used if the *applicationType* is CLIENT_1. If this value is not specified then the Endpoints shall support the Discovery Services defined in 5.4. Alternate discovery profiles are defined in Part 7.
discoveryUrls[]	String	A list of URLs for the discovery *Endpoints* provided by the application. If the *applicationType* is CLIENT_1, this field shall contain an empty list.

7.2 ApplicationInstanceCertificate

An *ApplicationInstanceCertificate* is a *ByteString* containing an encoded *Certificate.* The encoding of an *ApplicationInstanceCertificate* depends on the security technology mapping and is defined completely in Part 6. Table 104 specifies the information that shall be contained in an *ApplicationInstanceCertificate*.

Table 104 – ApplicationInstanceCertificate

Name	Type	Description
ApplicationInstanceCertificate	structure	*ApplicationInstanceCertificate* with signature created by a *Certificate Authority*.
version	String	An identifier for the version of the *Certificate* encoding.
serialNumber	ByteString	A unique identifier for the *Certificate* assigned by the Issuer.
signatureAlgorithm	String	The algorithm used to sign the *Certificate*. The syntax of this field depends on the *Certificate* encoding.
signature	ByteString	The signature created by the Issuer.
issuer	Structure	A name that identifies the Issuer *Certificate* used to create the signature.
validFrom	UtcTime	When the *Certificate* becomes valid.
validTo	UtcTime	When the *Certificate* expires.
subject	Structure	A name that identifies the application instance that the *Certificate* describes. This field shall contain the *productName* and the name of the organization responsible for the application instance.
applicationUri	String	The *applicationUri* specified in the *ApplicationDescription*. The *ApplicationDescription* is described in 7.1.
hostnames []	String	The name of the machine where the application instance runs. A machine may have multiple names if is accessible via multiple networks. The hostname may be a numeric network address or a descriptive name. *Server Certificates* shall have at least one hostname defined.
publicKey	ByteString	The public key associated with the *Certificate*.
keyUsage []	String	Specifies how the *Certificate* key may be used. *ApplicationInstanceCertificates* shall support Digital Signature, Non-Repudiation Key Encryption, Data Encryption and Client/Server Authorization. The contents of this field depend on the *Certificate* encoding.

7.3 BrowseResult

The components of this parameter are defined in Table 105.

Table 105 – BrowseResult

Name	Type	Description
BrowseResult	structure	The results of a Browse operation.
statusCode	StatusCode	The status for the *BrowseDescription.* This value is set to *Good_MoreReferencesExist* if there are still references to return for the *BrowseDescription*.
continuationPoint	ContinuationPoint	A *Server* defined opaque value that identifies the continuation point. The *ContinuationPoint* type is defined in 7.6.
References []	ReferenceDescription	The set of references that meet the criteria specified in the *BrowseDescription*. Empty, if no *References* met the criteria. The Reference Description type is defined in 7.24.

7.4 ContentFilter

7.4.1 ContentFilter structure

The *ContentFilter* structure defines a collection of elements that make up 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 110. 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. If an element cannot be traced back to the starting element it is ignored. Extra operands for any operator shall result in an error. Annex B provides examples using the *ContentFilter* structure.

Table 106 defines the ContentFilter structure.

Table 106 – 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 *FilterOperator* enumeration is defined in Table 110.
filterOperands []	Extensible Parameter FilterOperand	Operands used by the selected operator. The number and use depend on the operators defined in Table 110. This array needs at least one entry. This extensible parameter type is the *FilterOperand* parameter type specified in 7.4.4. It specifies the list of valid *FilterOperand* values.

7.4.2 ContentFilterResult

The components of this data type are defined in Table 107.

Table 107 – ContentFilterResult Structure

Name	Type	Description
ContentFilterResult	structure	A structure that contains any errors associated with the filter.
elementResults	ContentFilter ElementResult	A list of results for individual elements in the filter. The size and order of the list matches the size and order of the elements in the *ContentFilter* parameter.
statusCode	StatusCode	The status code for a single element.
operandStatusCodes []	StatusCode	A list of status codes for the operands in an element. The size and order of the list matches the size and order of the operands in the *ContentFilterElement.* This list is empty if no operand errors occurred.
operandDiagnosticInfos []	DiagnosticInfo	A list of diagnostic information for the operands in an element. The size and order of the list matches the size and order of the operands in the *ContentFilterElement.* This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the operands.
elementDiagnosticInfos []	DiagnosticInfo	A list of diagnostic information for individual elements in the filter. The size and order of the list matches the size and order of the elements in the *filter* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the elements.

Table 108 defines values for the *statusCode* parameter that are specific to this structure. Common *StatusCodes* are defined in Table 166.

Table 108 – ContentFilterResult Result Codes

Symbolic Id	Description
Bad_FilterOperandCountMismatch	The number of operands provided for the filter operator was less then expected for the operand provided
Bad_FilterOperatorInvalid	An unregognized operator was provided in a filter
Bad_FilterOperatorUnsupported	A valid operator was provided, but the server does not provide support for this filter operator.

Table 109 defines values for the operand*StatusCode* parameter that are specific to this structure. Common *StatusCodes* are defined in Table 166.

Table 109 – ContentFilterResult Operand Result Codes

Symbolic Id	Description
Bad_FilterOperandInvalid	See Table 166 for the description of this result code.
Bad_FilterElementInvalid	The referenced element is not a valid element in the content filter
Bad_FilterLiteralInvalid	The referenced literal is not a valid basedatatype
Bad_AttributeIdInvalid	The attribute id is not a valid attribute id in the system
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_NotTypeDefinition	The provided Nodeid was not a type definition nodeid.

7.4.3 FilterOperator

Table 110 defines the basic operators that can be used in a ContentFilter. See Table 111 for a description of advanced operators. See section 7.4.4 for a definition of operands.

Table 110 – Basic FilterOperator Definition

Operator	Number of Operands	Description
Equals_0	2	TRUE if operand[0] is equal to operand[1]. If the operands are of different types, the system shall perform any implicit conversion to a common type. This operator resolves to FALSE if no implicit conversion is available and the operands are of different types. This operator returns FALSE if the implicit conversion fails. See the discussion on data type precedence in Table 114 for more information how to convert operands of different types.
IsNull_1	1	TRUE if operand[0] is a null value.
GreaterThan_2	2	TRUE if operand[0] is greater than operand[1]. The following restrictions apply to the operands: [0]: Any operand that resolves to an ordered value. [1]: Any operand that resolves to an ordered value. The same conversion rules as defined for *Equals* apply.
LessThan_3	2	TRUE if operand[0] is less than operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
GreaterThanOrEqual_4	2	TRUE if operand[0] is greater than or equal to operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
LessThanOrEqual_5	2	TRUE if operand[0] is less than or equal to operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
Like_6	2	TRUE if operand[0] matches a pattern defined by operand[1]. See Table 112 for the definition of the pattern syntax. The following restructions apply to the operands: [0]: Any operand that resolves to a String. [1]: Any operand that resolves to a String. This operator resolves to FALSE if any operand can not be resolved to a string.
Not_7	1	TRUE if operand[0] is FALSE. The following restructions apply to the operands: [0]: Any operand that resolves to a Boolean. If the operand can not be resolved to a Boolean, the result is a NULL. See below for a discussion on the handling of NULL.
Between_8	3	TRUE if operand[0] is greater or equal to operand[1] and less than or equal to operand[2]. The following restructions apply to the operands: [0]: Any operand that resolves to an ordered value. [1]: Any operand that resolves to an ordered value. [2]: Any operand that resolves to an ordered value. If the operands are of different types, the system shall perform any implicit conversion to match all operands to a common type. If no implicit conversion is available and the operands are of different types, the particular result is FALSE. See the discussion on data type precedence in Table 114 for more information how to convert operands of different types.
InList_9	2..n	TRUE if operand[0] is equal to one or more of the remaining operands. The Equals Operator is evaluated for operand[0] and each remaining operand in the list. If any Equals evaluation is TRUE, InList returns TRUE.
And_10	2	TRUE if operand[0] and operand[1] are TRUE. The following restructions apply to the operands: [0]: Any operand that resolves to a Boolean. [1]: Any operand that resolves to a Boolean. If any operand can not be resolved to a Boolean it is considered a NULL. See below for a discussion on the handling of NULL.
Or_11	2	TRUE if operand[0] or operand[1] are TRUE. The following restructions apply to the operands: [0]: Any operand that resolves to a Boolean. [1]: Any operand that resolves to a Boolean. If any operand can not be resolved to a Boolean it is considered a NULL. See below for a discussion on the handling of NULL.
Cast_12	2	Converts operand[0] to a value with a data type with a NodeId identified by operand[1]. The following restructions apply to the operands: [0]: Any operand. [1]: Any operand that resolves to a NodeId or ExpandedNodeId where the *Node* is of the *NodeClass DataType*. If there is any error in conversion or in any of the parameters then the Cast Operation evaluates to a NULL. See below for a discussion on the handling of NULL.
BitwiseAnd_16	2	The result is an integer which matches the size of the largest operand and contains a bitwise And operation of the two operands where both have been converted to the lsame size (largest of the two operands) The following restructions apply to the operands:

		[0]: Any operand that resolves to a integer. [1]: Any operand that resolves to a integer. If any operand can not be resolved to a integer it is considered a NULL. See below for a discussion on the handling of NULL.
BitwiseOr_17	2	The result is an integer which matches the size of the largest operand and contains a bitwise Or operation of the two operands where both have been converted to the lsame size (largest of the two operands) The following restructions apply to the operands: [0]: Any operand that resolves to a Integer. [1]: Any operand that resolves to a Integer. If any operand can not be resolved to a Integer it is considered a NULL. See below for a discussion on the handling of NULL.

Many operands have restrictions on their type. This requires that the operand be evaluated to determine what the type is. In some cases, the type specified in the operand (i.e. a LiteralOperand). In other cases the type requires that the value of an attribute be read. ElementOperands are Boolean values unless the operator is Cast or a nested *RelatedTo* operator.

Table 111 defines complex operators that require a target node (i.e. row) to evaluate. These operators shall be re-evaluated for each possible target node in the result set.

Table 111 – Complex FilterOperator Definition

Operator	**Number of Operands**	**Description**
InView_13	1	TRUE if the target *Node* is contained in the *View* defined by operand[0]. The following restrictions apply to the operands: [0]: Any operand that resolves to a NodeId that identifies a View Node. If operand[0] does not resolve to a NodeId that identifies a View Node, this operation shall always be False.
OfType_14	1	TRUE if the target *Node* is of type operand[0] or of a subtype of operand[0]. The following restrictions apply to the operands: [0]: Any operand that resolves to a NodeId that identifies an ObjectType or VariableType Node. If operand[0] does not resolve to a NodeId that identifies an ObjectType or VariableType Node, this operation shall always be False.
RelatedTo_15	6	TRUE if the target *Node* is of type Operand[0] and is related to a *NodeId* 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 chaining of relationships (e.g. A is related to B is related to C). In this case, the referred to element returns a list of *NodeIds* instead of TRUE or FALSE. In this case if any errors occur or any of the operands can not be resolved to an appropriate value, the result of the chained relationship is an empty list of nodes. Operand[3] defines the number of hops the relationship should be followed. If Operand[3] is 1, then objects shall be directly related. If a hop is greater than 1, then a *NodeId* 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 *NodeId* is included in the sub-list. Operand [4] defines if Operands [0] and [1] should include support for subtypes of the types defined by these operands. A TRUE indicates support for subtypes Operand [5] defines if Operands [3] should include support for subtypes the reference type defined by Operand[3]. A TRUE indicates support for subtypes. The following restrictions apply to the operands: [0]: Any operand that resolves to a NodeId or ExpandedNodeId that identifies an ObjectType or VariableType Node or a reference to another element which is a RelatedTo operator. [1]: Any operand that resolves to a NodeId or ExpandedNodeId that identifies an ObjectType or VariableType Node or a reference to another element which is a RelatedTo operator. [2]: Any operand that resolves to a NodeId that identifies a ReferenceType Node. [3]: Any operand that resolves to a value implicitly convertible to Int32. [4]: Any operand that resolves to a value implicitly convertible to a boolean; if this operand does not resolve to a Boolean, then a value of FALSE is used. [5]: Any operand that resolves to a value implicitly convertible to a boolean; if this operand does not resolve to a Boolean, then a value of FALSE is used. If any of the operands [0],[1],[2],[3] do not resolve to an appropriate value then the result of this operation shall always be False (or an Empty set in the case of a nested relatedTo operand). See examples for RelatedTo in 7.4.4.

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.

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 112. The wildcard characters can be combined in a single string (i.e. 'Th[ia][ts]%' would match 'That is fine', 'This is fine', 'That as one', 'This it is', 'Then at any' etc.).

Table 112 – Wildcard characters

Special Character	Description
%	Match any string of zero or more characters (i.e. 'main%' would match any string that starts with 'main', '%en%' would match any string that contains the letters 'en' such as 'entail', 'green' and 'content'.) If a '%' sign is intend in a string the list operand can be used (i.e. 5[%] would match '5%').
_	Match any single character (i.e. '_ould' would match 'would', 'could'). If the '_' is indended in a string then the list operand can be used (i.e. 5[_] would match '5_').
\	Escape character allows literal interpretation (i.e. \\ is \, \% is %, _ is _)
[]	Match any single character in a list (i.e. 'abc[13-68] would match 'abc1','abc3','abc4','abc5','abc6', and 'abc8'. 'xyz[c-f]' would match 'xyzc', 'xyzd', 'xyze', 'xyzf').
[^]	Not Matching any single character in a list. The ^ shall be the first charcter inside on the []. (i.e. 'ABC[^13-5]' would NOT match 'ABC1', 'ABC3', 'ABC4', and 'ABC5'. xyz[^dgh] would NOT match 'xyzd', 'xyzg', 'xyzh'.)

Table 113 defines the conversion rules for the operand values. The types are automatically converted if an implicit conversion exists (I). If an explicit conversion exists (E) then type can be converted with the cast operator. If no conversion exists (X) the then types cannot be converted, however, some servers may support application specific explicit conversions. The types used in the table are defined in Part 3. A data type that is not in the table does not have any defined conversions.

Table 113 – Conversion Rules

Target Type (To) / Source Type (From)	Boolean	Byte	ByteString	DateTime	Double	ExpandedNodeId	Float	Guid	Int16	Int32	Int64	NodeId	SByte	StatusCode	String	LocalizedText	QualifiedName	UInt16	UInt32	UInt64	XmlElement
Boolean	-	I	X	X	I	X	I	X	I	I	I	X	I	X	E	X	X	I	I	I	X
Byte	E	-	X	X	I	X	I	X	I	I	I	X	I	X	E	X	X	I	I	I	X
ByteString	X	X	-	X	X	X	X	E	X	X	X	X	X	X	X	X	X	X	X	X	X
DateTime	X	X	X	-	X	X	X	X	X	X	X	X	X	X	E	X	X	X	X	X	X
Double	E	E	X	X	-	X	E	X	E	E	E	X	E	X	E	X	X	E	E	E	X
ExpandedNodeId	X	X	X	X	X	-	X	X	X	X	X	E	X	X	I	X	X	X	X	X	X
Float	E	E	X	X	I	X	-	X	E	E	E	X	E	X	E	X	X	E	E	E	X
Guid	X	X	E	X	X	X	X	-	X	X	X	X	X	X	E	X	X	X	X	X	X
Int16	E	E	X	X	I	X	I	X	-	I	I	X	E	X	E	X	X	E	I	I	X
Int32	E	E	X	X	I	X	I	X	E	-	I	X	E	E	E	X	X	E	E	I	X
Int64	E	E	X	X	I	X	I	X	E	E	-	X	E	E	E	X	X	E	E	E	X
NodeId	X	X	X	X	X	I	X	X	X	X	X	-	X	X	I	X	X	X	X	X	X
SByte	E	E	X	X	I	X	I	X	I	I	I	X	-	X	E	X	X	I	I	I	X
StatusCode	X	X	X	X	X	X	X	X	X	I	I	X	X	-	X	X	X	E	I	I	X
String	I	I	X	E	I	E	I	I	I	I	I	E	I	X	-	E	E	I	I	I	X
LocalizedText	X	X	X	X	X	X	X	X	X	X	X	X	X	X	I	-	X	X	X	X	X
QualifiedName	X	X	X	X	X	X	X	X	X	X	X	X	X	X	I	I	-	X	X	X	X
UInt16	E	E	X	X	I	X	I	X	I	I	I	X	E	I	E	X	X	-	I	I	X
UInt32	E	E	X	X	I	X	I	X	E	I	I	X	E	E	E	X	X	E	-	I	X
UInt64	E	E	X	X	I	X	I	X	E	E	I	X	E	E	E	X	X	E	E	-	X
XmlElement	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-

Arrays of a source type can be converted to arrays of the target type by converting each element. A conversion error for any element causes the entire conversion to fail.

Arrays of length 1 can be implicitly converted to a scalar value of the same type.

Guid, NodeIds and ExpandedNodeIds are converted to and from Strings using the syntax defined in Part 6.

Floating point values are rounded by adding 0.5 and truncating when they are converted to integer values.

ByteStrings are converted to Strings by formatting the bytes as a sequence of hexadecimal digits.

LocalizedText values are converted to Strings by dropping the Locale. Strings are converted to LocalizedText values by setting the Locale to “”.

QualifiedName values are converted to Strings by dropping the NamespaceIndex. Strings are converted to QualifiedName values by setting the NamespaceIndex to 0.

A StatusCode can be converted to and from a UInt32 and Int32 by copying the bits. Only the top 16-bits if the StatusCode are copied when it is converted to and from a UInt16 or Int16 value.

Boolean values are converted to ‘1’ when true and ‘0’ when false. Non zero numeric values are converted to true Boolean values. Numeric values of 0 are converted to false Boolean values. String values containing “true”, “false”, “1” or “0” can be converted to Boolean values. Other string values cause a conversion error. In this case strings are case-insensitive.

It is sometimes possible to use implicit casts when operands with different data types are used in an operation. In this situation the precedence rules defined in Table 114 are used to determine which implicit conversion to use. The first data type in the list (top down) has the most precedence. If a data type is not in this table then it cannot be converted implicitly while evaluating an operation.

For example, assume that A = 1.1 (Float) and B = 1 (Int32) and that these values are used with an Equals operator. This operation would be evaluated by casting the Int32 value to a Float since the Float data type has more precedence.

Table 114 – Data Precedence Rules

Rank	Data Type
1	Double
2	Float
3	Int64
4	UInt64
5	Int32
6	UInt32
7	StatusCode
8	Int16
9	UInt16
10	SByte
11	Byte
12	Boolean
13	Guid
14	String
15	ExpandedNodeId
16	NodeId
17	LocalizedText
18	QualifiedName

Operands may contain null values (i.e. values which do not exist). When this happens the element always evaluates to NULL (unless the IsNull operator has been specified), Table 115 defines how to combine elements that evaluate to NULL with other elements in a logical AND operation.

Table 115 – Logical AND Truth Table

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Table 116 defines how to combine elements that evaluate to NULL with other elements in a logical OR operation.

Table 116 – Logical OR Truth Table

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

The NOT operator always evaluates to NULL if applied to a NULL operand.

A ContentFilter which evaluates to NULL after all elements are evaluated is evaluated as false.

7.4.4 FilterOperand parameters

7.4.4.1 Overview

The *ContentFilter* structure specified in 7.4 defines a collection of elements that makes up filter criteria and contains different types of *FilterOperands*. The *FilterOperand* parameter is an extensible parameter. This parameter is defined in Table 117. The *ExtensibleParamter* type is defined in 7.11.

Table 117 – FilterOperand parameterTypeIds

Symbolic Id	Description
Element	Specifies 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	Specifies a literal value.
Attribute	Specifies any *Attribute* of an *Object* or *Variable Node* using a *Node* in the type system and relative path constructed from *ReferenceTypes* and *BrowseNames*.
SimpleAttribute	Specifies any *Attribute* of an *Object* or *Variable Node* using a *TypeDefinition* and a relative path constructed from *BrowseNames*.

7.4.4.2 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* shall 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 118 defines the *ElementOperand* type.

Table 118 – ElementOperand

Name	Type	Description
ElementOperand	structure	ElementOperand value.
index	UInt32	Index into the element array.

7.4.4.3 LiteralOperand

Table 119 defines the *LiteralOperand* type.

Table 119 – LiteralOperand

Name	Type	Description
LiteralOperand	structure	LiteralOperand value.
value	BaseDataType	A literal value.

7.4.4.4 AttributeOperand

Table 120 defines the *AttributeOperand* type.

Table 120 – AttributeOperand

Name	Type	Description
AttributeOperand	structure	Attribute of a *Node* in the address space.
nodeId	NodeId	*NodeId* of a *Node* 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.
browsePath	RelativePath	Browse path relative to the *Node* identified by the *nodeId* parameter. See 7.25 for the definition of *RelativePath*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in Part 6.
indexRange	NumericRange	This parameter is used to identify a single element of an array or a single range of indexes for an array. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is not used if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used. **Editor note: Support for structure and bit mask needs to be added.**

7.4.4.5 SimpleAttributeOperand

The *SimpleAttributeOperand* is a simplified form of the *AttributeOperand* and all of the rules that apply to the *AttributeOperand* also apply to the *SimpleAttributeOperand*. The examples provided in B.1 only use *AttributeOperands*, however, the *AttributeOperand* can be replaced by a *SimpleAttributeOperand* whenever all *ReferenceTypes* in the *RelativePath* are subtypes of *HierarchialReferences* and the targets are *Object* or *Variable Nodes and an Alias is not required*.

Table 121 defines the *SimpleAttributeOperand* type.

Table 121 – SimpleAttributeOperand

Name	Type	Description
SimpleAttributeOperand	structure	Attribute of a *Node* in the address space.
typeId	NodeId	*NodeId* of a *TypeDefinitionNode*. This parameter restricts the operand to instances of the *TypeDefinitionNode* or one of its subtypes.
browsePath[]	QualifiedName	A relative path to a *Node*. This parameter specifies a relative path using a list of *BrowseNames* instead of the *RelativePath* structure used in the *AttributeOperand*. The list of *BrowseNames* is equivalent to a *RelativePath* that specifies forward references which are subtypes of the *HierarchicalReferences ReferenceType*. All *Nodes* followed by the *browsePath* shall be of the *NodeClass Object* or *Variable*. If this list is empty the *Node* is the instance of the *TypeDefinition*.
attributeId	IntegerId	Id of the *Attribute*. The *IntegerId* is defined in 7.13. The *Value Attribute* shall be supported by all *Servers*. The support of other *Attributes* depends on requirements set in Profiles or other parts of this specification.
indexRange	NumericRange	This parameter is used to identify a single element of an array, or a single range of indexes for an array. The first element is identified by index 0 (zero). This parameter is ignored if the selected Node is not a Variable or the Value of a Variable is not an array. The parameter is null if not specifed. All values in the array are used if this parameter is not specified. The *NumericRange* type is defined in 7.21.

7.5 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)mod(max value + 1)

For example:

max value + 1 = 0

max value + 2 = 1

7.6 ContinuationPoint

A *ContinuationPoint* is used to pause a *Browse or QueryFirst* operation and allow it to be restarted later by calling *BrowseNext* or *QueryNext.* Operations are paused when the number of results found exceeds the limits set by either the *Client* or the *Server*.

The *Client* specifies the maximum number of results per operation in the request message. A *Server* shall not return more than this number of results but it may return fewer results. The *Server* allocates a *ContinuationPoint* if there are more results to return. The *Server* shall always return at least one result if it returns a *ContinuationPoint*.

Servers shall support at least one *ContinuationPoint* per *Session*. *Servers* specify a maximum number of *ContinuationPoints* per *Session* in the *ServerCapabilities Object* defined in Part 5. *ContinuationPoints* remain active until the *Client* retrieves the remaining results, the *Client* releases the *ContinuationPoint* or the *Session* is closed. A *Server* shall automatically free *ContinuationPoints* from prior requests if they are needed to process a new request. The *Server* returns a *Bad_ContinuationPointInvalid* error if a *Client* tries to use a *ContinuationPoint* that has been released.

Requests will often specify multiple operations that may or may not require a *ContinuationPoint*. A *Server* shall process the operations until it runs out of *ContinuationPoints*. Once that happens the *Server* shall return a *Bad_NoContinuationPoints* error for any remaining operations.

A *Client* restarts an operation by passing the *ContinuationPoint* back to the *Server*. *Server* should always be able to reuse the *ContinuationPoint* provided so *Servers* shall never return *Bad_NoContinuationPoints* error when continuing a previously halted operation.

A *ContinuationPoint* is a subtype of the *ByteString* data type.

7.7 DataValue

The components of this parameter are defined in Table 122.

Table 122 – DataValue

Name	Type	Description
DataValue	structure	The value and associated information.
value	BaseDataType	The data value.
statusCode	StatusCode	The *StatusCode* that defines with the *Server*'s ability to access/provide the value. The *StatusCode* type is defined in 7.33
sourceTimestamp	UtcTime	The source timestamp for the value.
sourcePicoSeconds	UInteger	Specifies the number of 10 picoseconds (1.0 e-11 seconds) intervals which shall be added to the sourceTimestamp.
serverTimestamp	UtcTime	The *Server* timestamp for the value.
serverPicoSeconds	UInteger	Specifies the number of 10 picoseconds (1.0 e-11 seconds) intervals which shall be added to the serverTimestamp.

PicoSeconds

Some applications require high resolution timestamps. The *PicoSeconds* fields allow applications to specify timestamps with 10 picosecond resolution. The actual size of the *PicoSeconds* field depends on the resolution of the *UtcTime DataType*. For example, if the *UtcTime DataType* has a resolution of 100 nanoseconds then the *PicoSeconds* field would have to store values up to 10000 in order to provide the 10 picosecond resolution. The resolution of the *UtcTime DataType* depends on the *Mappings* defined in Part 6.

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* shall 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* shall 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 shall 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 7.33.

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, shall 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 shall return a NULL value.

7.8 DiagnosticInfo

The components of this parameter are defined in Table 123.

Table 123 – DiagnosticInfo

Name	Type	Description
DiagnosticInfo	structure	Vendor-specific diagnostic information.
identifier	structure	The vendor-specific identifier of an error or condition.
namespaceUri	Int32	The symbolic id defined by the *symbolicIdIndex* parameter is defined within the context of a namespace. This namespace is represented as a string and is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *namespaceIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified.
symbolicId	Int32	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 *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *symbolicIdIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified.
locale	Int32	The locale part of the vendor-specific localized text describing the symbolic id. This localized text string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *localizedTextIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified.
localizedText	Int32	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 *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *localizedTextIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified.
additionalInfo	String	Vendor-specific diagnostic information.
innerStatusCode	StatusCode	The *StatusCode* from the inner operation. Many applications will make calls into underlying systems during OPC UA request processing. An OPC 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 *StatusCode*.

7.9 EndpointDescription

The components of this parameter are defined in Table 124.

Table 124 – EndpointDescription

Name	Type	Description
EndpointDescription	structure	Describes an *Endpoint* for a *Server*.
endpointUrl	String	The URL for the *Endpoint* described.
server	ApplicationDescription	The description for the *Server* that the *Endpoint* belongs to. The *ApplicationDescription* type is defined in 7.1.
serverCertificate	ApplicationInstance Certificate	The application instance *Certificate* issued to the *Server*. The *ApplicationInstanceCertificate* type is defined in 7.2.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type *MessageSecurityMode* type is defined in 7.14. A *SecureChannel* may have to be created even if the *securityMode* is NONE. The exact behaviour depends on the mapping used and is described in the Part 6.
securityPolicyUri	String	The URI for *SecurityPolicy* to use when securing messages. The set of known URIs and the *SecurityPolicies* associated with them are defined in Part 7.
userIdentityTokens[]	UserTokenPolicy	The user identity tokens that the *Server* will accept. The *Client* shall pass one of the *UserIdentityTokens* in the *ActivateSession* request. The *UserTokenPolicy* type is described in 7.36.
transportProfileUri	String	The URI of the *Transport Profile* supported by the *Endpoint*. Part 7 defines URIs for the *Transport Profiles*.
securityLevel	Byte	A numeric value that indicates how secure the EndpointDescription is compared to other EndpointDescriptions for the same Server. A value of 0 indicates that the EndpointDescription is not recommended and is only supported for backward compatibility.

7.10 ExpandedNodeId

The components of this parameter are defined in Table 125. *ExpandedNodeIds* allow the namespace to be specified explicitly as a string or with an index in the *Server*'s namespace table.

Table 125 – ExpandedNodeId

Name	Type	Description
ExpandedNodeId	structure	The *NodeId* with the namespace expanded to its string representation.
serverIndex	Index	Index that identifies the *Server* that contains the *TargetNode*. This *Server* may be the local *Server* or a remote *Server*. 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 Part 3 and Part 5). The *Client* may read the *Server* table *Variable* to access the description of the target *Server*
namespaceUri	String	The URI of the namespace. If this parameter is specified then the namespace index is ignored. 5.4 and Part 12 describes discovery mechanism that can be used to resolve URIs into URLs.
namespaceIndex	Index	The index in the *Server*'s namespace table. This parameter shall be 0 and is ignored in the *Server* if the namespace URI is specified.
identifierType	IdType	Type of the identifier element of the *NodeId*.
identifier	*	The identifier for a *Node* in the address space of an OPC UA *Server*. (see *NodeId* definition in Part 3).

7.11 ExtensibleParameter

The extensible parameter types can only be extended by additional parts of this multi-part specification.

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 126.

Concrete extensible parameters that are common to OPC UA are defined in Clause 7. Additional parts of this multi-part specification can define additional extensible parameter types.

Table 126 – 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.

7.12 Index

This primitive data type is an UInt32 that identifies an element of an array.

7.13 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.

7.14 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 127.

Table 127 – MessageSecurityMode Values

Value	Description
INVALID_0	The MessageSecurityMode is invalid. This value is the default value to avoid an accidental choice of no security. This choice will always be rejected.
NONE_1	No security is applied.
SIGN_2	All messages are signed but not encrypted.
SIGNANDENCRYPT_3	All messages are signed and encrypted.

7.15 MonitoringParameters

The components of this parameter are defined in Table 128.

Table 128 – MonitoringParameters

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

7.16 MonitoringFilter parameters

7.16.1 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 129. Other types can be defined by additional parts of this multi-part specification or other specifications based on OPC UA. The *ExtensibleParamter* type is defined in 7.11.

Each *MonitoringFilter* may have an associated *MonitoringFilterResult* structure which returns revised parameters and/or error information to clients in the response. The result structures, when they exist, are described in the section that defines the *MonitoringFilter*.

Table 129 – MonitoringFilter parameterTypeIds

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

7.16.2 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 130.

Table 130 – DataChangeFilter

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

The *DataChangeFilter does* not have an associated result structure.

7.16.3 EventFilter

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* shall include the fields defined by the *selectClauses* parameter of the *EventFilter*. The defined *EventType*s are specified in Part 5.

The *selectClause* and *whereClause* parameters are specified with the *SimpleAttributeOperand* structure (see 7.4.4.5). This structure requires the *NodeId* of an *EventType* supported by the *Server* and a path to an *InstanceDeclaration*. An *InstanceDeclaration* is a *Node* which can be found by following forward hierarchical references from the fully inherited *EventType* where the *Node* is also the source of a *HasModellingRule* reference. *EventTypes*, *InstanceDeclarations* and *Modelling Rules* are described completely in Part 3.

In some cases the same *BrowsePath* will apply to multiple *EventTypes*. If the *Client* specifies the *BaseEventType* in the SimpleAttributeOperand then the *Server* shall evaluate the *BrowsePath* without considering the *Type*.

Each *InstanceDeclaration* in the path shall be *Object* or *Variable Node*. The final *Node* in the path may be an *Object Node*, however, *Object Nodes* are only available for *Events* which are visible in the *Server's Address Space*.

The *SimpleAttributeOperand* structure allows the Client to specify any *Attribute*, however, the *Server* is only required to support the *Value Attribute* for *Variable Nodes* and the *NodeId Attribute* for *Object Nodes*. That said, profiles defined in Part 7 may make support for additional *Attributes* mandatory.

The *SimpleAttributeOperand* structure is used in the *selectClause* to select the value to return if an *Event* meets the criteria specified by the *whereClause*. A null value is returned in the corresponding event field in the Publish response if the selected *field* is not part of the *Event* or an error was returned in the *selectClauseResults* of the *EventFilterResult*. If the selected *field* is available but cannot be returned to *Client* then the *Server* shall return a *StatusCode* that indicates the reason for the error. For example, the *Server* shall return a *Bad_UserAccessDenied* error if the value is not accessible to the user associated with the *Session*. If a *Value Attribute* has an uncertain or bad *StatusCode* associated with it then the *Server* shall return the *StatusCode* instead of the *Value*.

The *Server* shall validate the *selectClauses* when a *Client* creates or updates the *EventFilter*. Any errors which are true for all possible *Events* are returned in the *selectClauseResults* parameter described in Table 132. The *Server* shall not report errors that might occur depending on the state or the *Server* or type of *Event*. For example, a *selectClause* that requests a single element in an array would always produce an error if the *DataType* of the *Attribute* is a scalar. However, even if the *DataType* is an array an error could occur if the requested index does not exist for a particular *Event*, the *Server* would not report an error in the *selectClauseResults* parameter if the latter situation existed.

The *SimpleAttributeOperand* is used in the *whereClause* to select a value which forms part of a logical expression. These logical expressions are then used to determine whether a particular *Event* should be reported to the *Client*. The *Server* shall use a null value for if any error occurs when a *whereClause* is evaluated for a particular *Event*. If a *Value Attribute* has an uncertain or bad *StatusCode* associated with it then the *Server* shall use a null value instead of the *Value*.

Any basic *FilterOperator* in Table 110 may be used in the *whereClause*, however, only the OfType *FilterOperator* from Table 111 is permitted.

The *Server* shall validiate the *whereClauses* when a *Client* creates or updates the *EventFilter*. Any structural errors in the construction of the filter and any errors which are true for all possible *Events* are returned in the *whereClauseResult* parameter described in Table 132. Errors that could occur depending on the state of the *Server* or the *Event* are not reported.

SubscriptionControlEvents are special *Events* which are used to provide control information to the *Client*. These *Events* are only published to the *MonitoredItems* in the *Subscription* that produced the *SubscriptionControlEvent*. These *Events* bypass the *whereClause*.

Table 131 defines the EventFilter structure.

Table 131 – EventFilter structure

Name	Type	Description
EventFilter	structure	
selectClauses []	SimpleAttribute Operand	List of the values to return with each *Event* in a *Notification*. At least one valid clause shall be specified. See 7.4.4.5 for the definition of *SimpleAttributeOperand*.
whereClause	ContentFilter	Limit the *Notifications* to those *Events* that match the criteria defined by this ContentFilter. The ContentFilter structure is described in 7.4. The *AttributeOperand* structure may not be used in *EventFilters*.

Table 132 defines the EventFilterResult structure.

Table 132 – EventFilterResult structure

Name	Type	Description
EventFilterResult	structure	
selectClauseResults[]	StatusCode	List of status codes for the elements in the select clause. The size and order of the list matches the size and order of the elements in the *selectClauses* request parameter. The Server returns null for unavailable or rejected *Event* fields.
selectClauseDiagnosticInfos[]	DiagnosticInfo	A list of diagnostic information for individual elements in the select clause. The size and order of the list matches the size and order of the elements in the *selectClauses* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the select clauses.
whereClauseResult	ContentFilter Result	An results associated with the *whereClause* request parameter. The *ContentFilterResult* type is defined in 7.4.2.

Table 133 defines values for the selectClauseResults parameter. Common *StatusCodes* are defined in Table 166.

Table 133 – EventFilterResult Result Codes

Symbolic Id	Description
Bad_TypeDefinitionInvalid	See Table 166 for the description of this result code. The typeId is not the NodeId for BaseEventType or a subtype of it
Bad_NodeIdUnknown	See Table 166 for the description of this result code. The browsePath is specified but it will never exist in any Event.
Bad_BrowseNameInvalid	See Table 166 for the description of this result code. The browsePath is specified and contains a null element
Bad_AttributeIdInvalid	See Table 166 for the description of this result code. The node specified by the browse path will never allow the given attribute id to be returned.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_TypeMismatch	See Table 166 for the description of this result code. The indexRange is valid but the value of the Attribute is never an array.

7.16.4 AggregateFilter

The *AggregateFilter* defines the aggregate function that should be used to calculate the values to be returned. See Part 13 for details on possible aggregate functions. It specifies a startTime of the first aggregate to be calculated. The samplingInterval of the MonitoringAttributes (see 7.15) defines how the server should internally sample the underlying data source. The processingInterval specifies the size of a period where the aggregate is calculated. The queueSize from the MonitoringAttributes specifies the number of processed values that should be kept.

The intention of the AggregateFilter is not to read historical data, the HistoryRead service should be used for this purpose. However, it is allowed that the startTime is set to a time that is in the past when received from the server. The number of aggregates to be calculated in the past should not exceed the queueSize defined in the MonitoringAttributes since the values exceeding the queueSize would directly be discharged and never returned to the client.

The startTime and the processingInterval can be revised by the server, but the startTime should remain in the same boundary (startTime + revisedProcessingInterval * n = revisedStartTime). That behaviour simplifies accessing historical values of the aggregates using the same boundaries by calling the HistoryRead service. The extensible Parameter AggregateFilterResult is used to return the revised values for the AggregateFilter.

Some underlying systems may poll data and produce multiple samples with the same value. Other systems may only report changes to the values. The definition for each aggregate type explains how to handle the two different scenarios.

The *MonitoredItem* only reports values for intervals that have completed when the publish timer expires. Unused data is carried over and used to calculate a value returned in the next publish.

The *ServerTimestamp* for each interval shall be the time of the end of the processing interval.

The AggregateFilter is defined in Table 134.

Table 134 – AggregateFilter structure

Name	Type	Description
AggregateFilter	structure	
startTime	UtcTime	Beginning of period to calculate the aggregate the first time. The size of each period used to calculate the aggregate is defined by the samplingInterval of the MonitoringAttributes (see 7.15).
aggregateType	NodeId	The NodeId of the *HistoryAggregate* object that indicates the list of *Aggregates* to be used when retrieving processed data. See Part 13 for details.
processingInterval	Duration	The period be used to compute the aggregate.
aggregateConfiguration	Aggregate Configuration	This parameter allows *Clients* to override the *Aggregate* configuration settings supplied by the *AggregateConfiguration* object on a per monitored item basis. See Part 13 for more information on *Aggregate* configurations. If the *Server* does not support the ability to override the *Aggregate* configuration settings it shall return a *StatusCode* of Bad_AggregateListMismatch.
useSeverCapabilities Defaults	Boolean	If value = TRUE use Aggregate configuration settings as outlined by the AggregateConfiguration object. If value=FALSE use configuration settings as outlined in the following aggregateConfiguration parameters. Default is TRUE.
treatUncertainAsBad	Boolean	As described in Part 13.
percentDataBad	Byte	As described in Part 13.
percentDataGood	Byte	As described in Part 13.
steppedSloped Extrapolation	Boolean	As described in Part 13.

The AggregateFilterResult defines the revised AggregateFilter the server can return when an AggregateFilter is defined for a MonitoredItem in the CreateMonitoredItems or ModifyMonitoredItems services. The AggregateFilterResult is defined in Table 135.

Table 135 – AggregateFilterResult structure

Name	Type	Description
AggregateFilterResult	structure	
revisedStartTime	UtcTime	The actual StartTime interval that the *Server* shall use. This value is based on a number of factors, including capabilities of the server to access historical data. The revisedStartTime should remain in the same boundary as the startTime (startTime + samplingInterval * n = revisedStartTime).
revisedProcessingInterval	Duration	The actual processingInterval that the *Server* shall use. The revisedProcessingInterval shall be at least two times the revisedSamplingInterval for the MonitoredItem.

7.17 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 136.

Table 136 – 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.

7.18 NodeAttributes parameters

7.18.1 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 137. The *ExtensibleParamter* type is defined in 7.11.

Table 137 – NodeAttributes parameterTypeIds

Symbolic Id	Description
ObjectAttributes	Defines the *Attributes* for the *Object NodeClass*.
VariableAttributes	Defines the *Attributes* for the *Variable NodeClass*.
MethodAttributes	Defines the *Attributes* for the *Method NodeClass*.
ObjectTypeAttributes	Defines the *Attributes* for the *ObjectType NodeClass*.
VariableTypeAttributes	Defines the *Attributes* for the *VariableType NodeClass*.
ReferenceTypeAttributes	Defines the *Attributes* for the *ReferenceType NodeClass*.
DataTypeAttributes	Defines the *Attributes* for the *DataType NodeClass*.
ViewAttributes	Defines the *Attributes* for the *View NodeClass*.

Table 138 defines the bit mask used in the NodeAttributes parameters to specify which *Attributes* are set by the *Client*.

Table 138 – Bit mask for specified Attributes

Field	Bit	Description
AccessLevel	0	Indicates if the AccessLevel Attribute is set.
ArrayDimensions	1	Indicates if the ArrayDimensions Attribute is set.
Reserved	2	Reserved to be consistent with WriteMask defined in Part 3.
ContainsNoLoops	3	Indicates if the ContainsNoLoops Attribute is set.
DataType	4	Indicates if the DataType Attribute is set.
Description	5	Indicates if the Description Attribute is set.
DisplayName	6	Indicates if the DisplayName Attribute is set.
EventNotifier	7	Indicates if the EventNotifier Attribute is set.
Executable	8	Indicates if the Executable Attribute is set.
Historizing	9	Indicates if the Historizing Attribute is set.
InverseName	10	Indicates if the InverseName Attribute is set.
IsAbstract	11	Indicates if the IsAbstract Attribute is set.
MinimumSamplingInterval	12	Indicates if the MinimumSamplingInterval Attribute is set.
Reserved	13	Reserved to be consistent with WriteMask defined in Part 3.
Reserved	14	Reserved to be consistent with WriteMask defined in Part 3.
Symmetric	15	Indicates if the Symmetric Attribute is set.
UserAccessLevel	16	Indicates if the UserAccessLevel Attribute is set.
UserExecutable	17	Indicates if the UserExecutable Attribute is set.
UserWriteMask	18	Indicates if the UserWriteMask Attribute is set.
ValueRank	19	Indicates if the ValueRank Attribute is set.
WriteMask	20	Indicates if the WriteMask Attribute is set.
Value	21	Indicates if the Value Attribute is set.
Reserved	22:32	Reserved for future use. Shall always be zero.

7.18.2 ObjectAttributes parameter

Table 139 defines the *ObjectAttributes* parameter.

Table 139 – ObjectAttributes

Name	Type	Description
ObjectAttributes	structure	Defines the *Attributes* for the *Object NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
eventNotifier	Byte	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.3 VariableAttributes parameter

Table 140 defines the *VariableAttributes* parameter.

Table 140 – VariableAttributes

Name	Type	Description
VariableAttributes	structure	Defines the *Attributes* for the *Variable NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
value	Defined by the *DataType Attribute*	See Part 3 for the description of this *Attribute*.
dataType	NodeId	See Part 3 for the description of this *Attribute*.
valueRank	Int32	See Part 3 for the description of this *Attribute*.
arrayDimensions	UInt32[]	See Part 3 for the description of this *Attribute*.
accessLevel	Byte	See Part 3 for the description of this *Attribute*.
userAccessLevel	Byte	See Part 3 for the description of this *Attribute*.
minimumSamplingInterval	Duration	See Part 3 for the description of this *Attribute*.

historizing	Boolean	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.4 MethodAttributes parameter

Table 141 defines the *MethodAttributes* parameter.

Table 141 – MethodAttributes

Name	Type	Description
BaseAttributes	structure	Defines the *Attributes* for the *Method NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
executable	Boolean	See Part 3 for the description of this *Attribute*.
userExecutable	Boolean	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.5 ObjectTypeAttributes parameter

Table 142 defines the *ObjectTypeAttributes* parameter.

Table 142 – ObjectTypeAttributes

Name	Type	Description
ObjectTypeAttributes	structure	Defines the *Attributes* for the *ObjectType NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
isAbstract	Boolean	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.6 VariableTypeAttributes parameter

Table 143 defines the *VariableTypeAttributes* parameter.

Table 143 – VariableTypeAttributes

Name	Type	Description
VariableTypeAttributes	structure	Defines the *Attributes* for the *VariableType NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
value	Defined by the *DataType Attribute*	See Part 3 for the description of this *Attribute*.
dataType	NodeId	See Part 3 for the description of this *Attribute*.
valueRank	Int32	See Part 3 for the description of this *Attribute*.
arrayDimensions	UInt32[]	See Part 3 for the description of this *Attribute*.
isAbstract	Boolean	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.7 ReferenceTypeAttributes parameter

Table 144 defines the *ReferenceTypeAttributes* parameter.

Table 144 – ReferenceTypeAttributes

Name	Type	Description
ReferenceTypeAttributes	structure	Defines the *Attributes* for the *ReferenceType NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
isAbstract	Boolean	See Part 3 for the description of this *Attribute*.
symmetric	Boolean	See Part 3 for the description of this *Attribute*.
inverseName	LocalizedText	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.8 DataTypeAttributes parameter

Table 145 defines the *DataTypeAttributes* parameter.

Table 145 – DataTypeAttributes

Name	Type	Description
DataTypeAttributes	structure	Defines the *Attributes* for the *DataType NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
isAbstract	Boolean	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.18.9 ViewAttributes parameter

Table 146 defines the *ViewAttributes* parameter.

Table 146 – ViewAttributes

Name	Type	Description
ViewAttributes	structure	Defines the *Attributes* for the *View NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See Part 3 for the description of this *Attribute*.
description	LocalizedText	See Part 3 for the description of this *Attribute*.
containsNoLoops	Boolean	See Part 3 for the description of this *Attribute*.
eventNotifier	Byte	See Part 3 for the description of this *Attribute*.
writeMask	UInt32	See Part 3 for the description of this *Attribute*.
userWriteMask	UInt32	See Part 3 for the description of this *Attribute*.

7.19 NotificationData parameters

7.19.1 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 147. Other types can be defined by additional parts of this multi-part specification or other specifications based on OPC UA. The *ExtensibleParamter* type is defined in 7.11.

There may be multiple notifications for a single MonitoredItem in a single NotificationData structure. When that happens the Server shall ensure the notifications appear in the same order that they were queued in the MonitoredItem. Theses notifications do not need to appear as a contiguous block.

Table 147 – NotificationData parameterTypeIds

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

7.19.2 DataChangeNotification parameter

Table 148 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 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 shall be queued without applying the filter. If the current value is not available after the first sampling interval the first *Notification* shall be queued after getting the initial value or status from the data source.

Table 148 – DataChangeNotification

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

7.19.3 EventNotificationList parameter

Table 149 defines the *NotificationData* parameter used for *EventNotifications*.

The EventNotificationList 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 149 – EventNotificationList

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

7.19.4 StatusChangeNotification parameter

Table 150 defines the *NotificationData* parameter used for a *StatusChangeNotification*.

The *StatusChangeNotification* informs the client about a change in the status of a *Subscription*.

Table 150 – StatusChangeNotification

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

7.20 NotificationMessage

The components of this parameter are defined in Table 151.

Table 151 – NotificationMessage

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

7.21 NumericRange

This parameter is defined in Table 152. A formal BNF definition of the numeric range can be found in Appendix A3.

The syntax for the string contains one of the following two constructs. 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. The second construct is a range represented by two integers separated by the colon (“:”) character. The first integer shall 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. No other characters, including white-space characters, are permitted.

Multi-dimensional arrays can be indexed by specifying a range for each dimension separated by a ‘,’. For example, a 2x2 block in a 4x4 matrix could be selected with the range “1:2,0:1”. A single element in a multi-dimensional array can be selected by specifying a single number instead of a range. For example, “1,1” specifies selects the [1,1] element in a two dimensional array.

Dimensions are specified in the order that they appear in the *ArrayDimensions Attribute*. All dimensions shall be specified for a *NumericRange* to be valid.

All indexes start with 0. The maximum value for any index is one less than the length of the dimension.

When reading a value the indexes may not specify a range that is within the bounds of the array. The *Server* shall return a partial result if some elements exist within the range. The *Server* shall return a *Bad_OutOfRange* If no elements exist within the range.

When writing a value the size of the array shall match the size specified by the *NumericRange*. The *Server* shall return an error if it cannot write all elements specified by the *Client*.

The *NumericRange* can also be used to specify substrings for *ByteString* and *String* values. Arrays of *ByteString* and *String* values are treated as two dimensional arrays where the final index specifies the substring range within the *ByteString* or *String* value. The entire *ByteString* or *String* value is selected if the final index is omitted.

Table 152 – NumericRange

Name	Type	Description
NumericRange	String	A number or a numeric range. A null string indicates that this parameter is not used.

7.22 QueryDataSet

The components of this parameter are defined in Table 153.

Table 153 – QueryDataSet

Name	Type	Description
QueryDataSet	structure	Data related to a *Node* returned in a Query response.
nodeId	ExpandedNodeId	The *NodeId* for this *Node* description.
typeDefinitionNode	ExpandedNodeId	The *NodeId* for the type definition for this *Node* description.
values[]	BaseDataType	Values for the selected *Attributes.* The order of returned items matches the order of the requested items. There is an entry for each requested item for the given *TypeDefinitionNode* that matches the selected instance, this includes any related nodes that were specified using a relative path from the selected instance’s *TypeDefinitionNode*. If no values where 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 *ReferenceDescription* or array of *ReferenceDescriptions* are returned for that item.

7.23 ReadValueId

The components of this parameter are defined in Table 154.

Table 154 – ReadValueId

Name	Type	Description
ReadValueId	structure	Identifier for an item to read or to monitor.
nodeId	NodeId	*NodeId* of a *Node*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in Part 6.
indexRange	NumericRange	This parameter is used to identify a single element of 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 *NumericRange* type is defined in 7.21. This parameter is null if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array, and this parameter is null, then all elements are to be included in the range.
dataEncoding	QualifiedName	This parameter specifies the *BrowseName* of the *DataTypeEncoding* that the *Server* should use when returning the Value *Attribute* of a *Variable*. It is an error to specify this parameter for other *Attributes*. A *Client* can discover what *DataTypeEncoding*s are available by following the *HasEncoding Reference* from the *DataType Node* for a *Variable*. OPC UA defines *BrowseNames* which *Servers* shall recognize even if the *DataType Nodes* are not visible in the *Server* address space. These *BrowseNames* are: DefaultBinary The default or native binary (or non-XML) encoding. DefaultXML The default XML encoding. Each *DataType* shall support at least one of these encodings. *DataTypes* that do not have a true binary encoding (e.g. they only have a non-XML text encoding) should use the *DefaultBinary* name to identify the encoding that is considered to be the default non-XML encoding. *DataTypes* that support at least one XML-based encoding shall identify one of the encodings as the DefaultXML encoding. Other standards bodies may define other well-known data encodings that could be supported. If this parameter is not specified then the *Server* shall choose either the DefaultBinary or *DefaultXML* encoding according to what *Message* encoding (see Part 6) is used for the *Session*. If the *Server* does not support the encoding that matches the *Message* encoding then the *Server* shall choose the default encoding that it does support. If this parameter is specified for a *MonitoredItem*, the *Server* shall set the *StructureChanged* bit in the *StatusCode* (see 7.33) if the *DataTypeEncoding* changes. The *DataTypeEncoding* changes if the *DataTypeVersion* of the *DataTypeDescription* or the *DataTypeDictionary* associated with the *DataTypeEncoding* changes.

7.24 ReferenceDescription

The components of this parameter are defined in Table 155.

Table 155 – ReferenceDescription

Name	Type	Description
ReferenceDescription	structure	Reference parameters returned for the *Browse Service*.
referenceTypeId	NodeId	*NodeId* of the *ReferenceType* that defines the *Reference*.
isForward	Boolean	If the value is TRUE, the *Server* followed a forward *Reference*. If the value is FALSE, the *Server* followed an inverse *Reference*.
nodeId	Expanded NodeId	*NodeId* of the *TargetNode* as assigned by the *Server* identified by the *Server* index. The *ExpandedNodeId* type is defined in 7.10. If the *server*Index indicates that the *TargetNode* is a remote *Node*, then the *nodeId* shall contain the absolute namespace URI. If the *TargetNode* is a local *Node* the *nodeId* shall contain the namespace index.
browseName[1]	QualifiedName	The *BrowseName* of the *TargetNode*.
displayName	LocalizedText	The *DisplayName* of the *TargetNode*.
nodeClass[1]	NodeClass	*NodeClass* of the *TargetNode*.
typeDefinition[1]	Expanded NodeId	Type definition *NodeId* of the *TargetNode*. Type definitions are only available for the *NodeClasses Object* and *Variable*. For all other *NodeClasses* a null NodeId shall be returned.
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.		

7.25 RelativePath

The components of this parameter are defined in Table 156.

Table 156 – RelativePath

Name	Type	Description
RelativePath	structure	Defines a sequence of *References* and *BrowseNames* to follow.
elements []	RelativePath Element	A sequence of *References* and *BrowseNames* to follow. Each element in the sequence is processed by finding the targets and then using those targets as the starting nodes for the next element. The targets of the final element are the target of the *RelativePath*.
referenceTypeId	NodeId	The type of reference to follow from the current node. The current path can not be followed any further if the referenceTypeId is not available on the Node instance.
isInverse	Boolean	Indicates whether the inverse *Reference* should be followed. The inverse reference is followed if this value is TRUE.
includeSubtypes	Boolean	Indicates whether subtypes of the *ReferenceType* should be followed. Subtypes are included if this value is TRUE.
targetName	QualifiedName	The *BrowseName* of the target node. The final element may have an empty *targetName*. In this situation all targets of the references identified by the referenceTypeId are the targets of the *RelativePath*. The *targetName* shall be specified for all other elements. The current path can not be followed any further if no targets with the specified *BrowseName* exist.

A *RelativePath* can be applied to any starting *Node*. The targets of the *RelativePath* are the set of *Nodes* that are found by sequentially following the elements in *RelativePath*.

A text format for the *RelativePath* can be found in Appendix A2. This format is used in examples that explain the *Services* that make use of the *RelativePath* structure.

7.26 RequestHeader

The components of this parameter are defined in Table 157.

Table 157 – RequestHeader

Name	**Type**	**Description**
RequestHeader	structure	Common parameters for all requests submitted on a *Session*.
authenticationToken	Session AuthenticationToken	The secret *Session* identifier used to verify that the request is associated with the *Session.* The *SessionAuthenticationToken* type is defined in 7.29.
timestamp	UtcTime	The time the *Client* sent the request.
requestHandle	IntegerId	A *requestHandle* associated with the request. This client defined handle can be used to cancel the request. It is also returned in the response.
returnDiagnostics	UInt32	A bit mask that identifies the types of vendor-specific diagnostics to be returned in *diagnosticInfo r*esponse 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. Bit Value — Diagnostics to return 0x0000 0001 — ServiceLevel / SymbolicId 0x0000 0002 — ServiceLevel / LocalizedText 0x0000 0004 — ServiceLevel / AdditionalInfo 0x0000 0008 — ServiceLevel / Inner *StatusCode* 0x0000 0010 — ServiceLevel / Inner Diagnostics 0x0000 0020 — OperationLevel / SymbolicId 0x0000 0040 — OperationLevel / LocalizedText 0x0000 0080 — OperationLevel / AdditionalInfo 0x0000 0100 — OperationLevel / Inner *StatusCode* 0x0000 0200 — OperationLevel / Inner Diagnostics Each of these values is composed of two components, *level* and *type*, as described below. If none are requested, as indicated by a 0 value, or if no diagnostic information was encountered in processing of the request, then diagnostics information is not returned. *Level*: ServiceLevel — return diagnostics in the *diagnosticInfo* of the *Service*. OperationLevel — return diagnostics in the *diagnosticInfo* defined for individual operations requested in the *Service*. *Type:*: 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 *StatusCode* associated with the operation or *Service*. InnerDiagnostics — return the inner diagnostic info associated with the operation or *Service*. 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.
auditEntryId	String	An identifier that identifies the *Client*'s security audit log entry associated with this request. An empty string value means that this parameter is not used. The *AuditEntryId* typically contains who initiated the action and from where it was initiated. The *AuditEventId* is included in the *AuditEvent* to allow human readers to correlate an *Event* with the initiating action. More details of the *Audit* mechanisms are defined in 6.2 and in Part 3.
timeoutHint	UInt32	This timeout in milliseconds is used in the *Client* side *Communication Stack* to set the timeout on a per-call base. For a *Server* this timeout is only a hint and can be used to cancel long running operations to free resources. If the Server detects a timeout, he can cancel the operation by sending the *Service* result *Bad_Timeout*. The *Server* 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.

7.27 ResponseHeader

The components of this parameter are defined in Table 158.

Table 158 – ResponseHeader

Name	Type	Description
ResponseHeader	structure	Common parameters for all responses.
timestamp	UtcTime	The time the *Server* sent the response.
requestHandle	IntegerId	The requestHandle given by the *Client* to the request.
serviceResult	StatusCode	OPC UA-defined result of the *Service* invocation. The *StatusCode* type is defined in 7.33.
serviceDiagnostics	DiagnosticInfo	Diagnostic information for the *Service* invocation. This parameter is empty if diagnostics information was not requested in the request header. The *DiagnosticInfo* type is defined in 7.8.
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 7.8). 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.

7.28 ServiceFault

The components of this parameter are defined in Table 159.

The *ServiceFault* parameter is returned instead of the Service response message when a service level error occurs. The *requestHandle* 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 Part 6.

Table 159 – ServiceFault

Name	Type	Description
ServiceFault	structure	An error response sent when a service level error occurs.
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).

7.29 SessionAuthenticationToken

The *SessionAuthenticationToken* type is an opaque identifier that is used to identify requests associated with a particular *Session*. This identifier is used in conjunction with the *SecureChannelId* or *Client Certificate* to authenticate incoming messages. It is the secret form of the *sessionId* for internal use in the *Client* and *Server Applications*.

A Server returns a *SessionAuthenticationToken* in the *CreateSession* response. The *Client* then sends this value with every request which allows the *Server* to verify that the sender of the request is the same as the sender of the original *CreateSession* request.

For the purposes of this discussion, a *Server* consists of application (code) and a *Communication Stack* as shown in Figure 27. The security provided by the *SessionAuthenticationToken* depends on a trust relationship between the *Server* application and the *Communication Stack*. The *Communication Stack* shall be able to verify the sender of the message and it uses the *SecureChannelId* or the *Client Certificate* to identify the sender to the *Server*. In these cases, the *SessionAuthenticationToken* is a UInt32 identifier that allows the *Server* to distinguish between different *Sessions* created by the same sender.

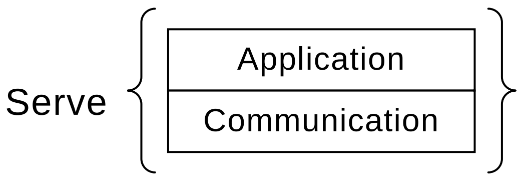

Figure 27 – Logical layers of a *Server*

In some cases, the application and the *Communication Stack* cannot exchange information at runtime which means the application will not have access to the *SecureChannelId* or the *Certificate* used to create the *SecureChannel.* In these cases the application shall create a random *ByteString* value that is at least 32 bytes long. This value shall be kept secret and shall always be exchanged over a *SecureChannel* with encryption enabled. The Administrator is responsible for ensuring that encryption is enabled. The *Profiles* in Part 7 may define additional requirements for a *ByteString SessionAuthenticationToken.*

Client and *Server* applications should be written to be independent of the *SecureChannel* implementation. Therefore, they should always treat the *SessionAuthenticationToken* as secret information even if it is not required when using some *SecureChannel* implementations.

Figure 28 illustrates the information exchanged between the Client, the Server and the Server Stack when the *Client* obtains a *SessionAuthenticationToken.* In this figure the GetSecureChannelInfo step represents an API that depends on the *Communication Stack* implementation.

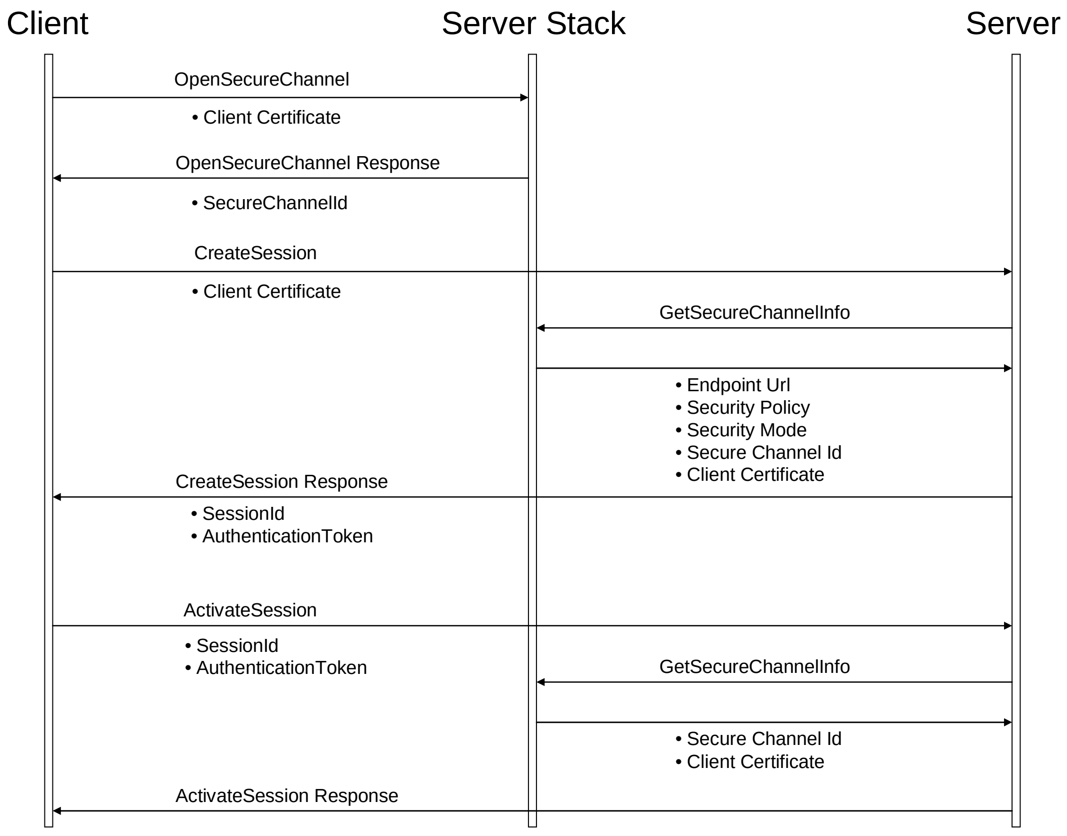

Figure 28 – Obtaining a SessionAuthenticationToken

The *SessionAuthenticationToken* is a subtype of the *NodeId* data type, however, it is never used to identify a *Node* in the address space. *Servers* may assign a value to the *NamespaceIndex*, however, its meaning is *Server* specific.

7.30 SignatureData

The components of this parameter are defined in Table 160.

Table 160 – SignatureData

Name	Type	Description
SignatureData	structure	Contains a digital signature created with a *Certificate*.
signature	ByteString	This is a signature generated with the private key associated with a *Certificate*.
algorithm	String	A string containing the URI of the algorithm. The URI string values are defined as part of the security profiles specified in Part 7.

7.31 SignedSoftwareCertificate

A *SignedSoftwareCertificate* is a *ByteString* containing an encoded *Certificate.* The encoding of a *SignedSoftwareCertificate* depends on the security technology mapping and is defined completely in Part 6. Table 161 specifies the information that shall be contained in a *SignedSoftwareCertificate*.

Table 161 – SignedSoftwareCertificate

Name	Type	Description
SignedSoftwareCertificate	structure	A *SoftwareCertificate* with a signature created by Certifying Authority.
version	String	An identifier for the version of the *Certificate* encoding.
serialNumber	ByteString	A unique identifier for the *Certificate* assigned by the Issuer.
signatureAlgorithm	String	The algorithm used to sign the *Certificate*. The syntax of this field depends on the *Certificate* encoding.
signature	ByteString	The signature created by the Issuer.
issuer	Structure	A name that identifies the Issuer *Certificate* used to create the signature.
validFrom	UtcTime	When the *Certificate* becomes valid.
validTo	UtcTime	When the *Certificate* expires.
subject	Structure	A name that identifies the product which the *Certificate* describes. This field shall contain the *productName* and *vendorName* from the *SoftwareCertificate*.
subjectAltName[]	Structure	A list of alternate names for the product. This list shall include the *productUri* specified in the *SoftwareCertificate*.
publicKey	ByteString	The public key associated with the *Certificate*.
keyUsage[]	String	Specifies how the *Certificate* key may be used. *SignedSoftwareCertificates* may only be used for creating Digital Signatures and Non-Repudiation. The contents of this field depends on the *Certificate* encoding.
softwareCertificate	ByteString	The XML encoded form of the *SoftwareCertificate* stored as UTF8 text. Part 6 describes the XML representation for the *SoftwareCertificate.*

7.32 SoftwareCertificate

The components of this parameter are defined in Table 162.

Table 162 – SoftwareCertificate

Name	Type	Description
SoftwareCertificate	structure	A *Certificate* describing a product.
productName	String	The name of the product that is certified. This field shall be specified.
productUri	String	A globally unique identifier for the product that is certified. This field shall be specified.
vendorName	String	The name of the vendor responsible for the product. This field shall be specified.
vendorProductCertificate	ByteString	The DER encoded form of the X.509 *Certificate* which is assigned to the product by the vendor. This field may be omitted.
softwareVersion	String	Software version. This field shall be specified.
buildNumber	String	Build number. This field shall be specified.
buildDate	UtcTime	Date and time of the build. This field shall be specified.
issuedBy	String	URI of the certifying authority. This field shall be specified.
issueDate	UtcTime	Specifies when the *Certificate* was issued by the certifying authority. This field shall be specified.
supportedProfiles []	structure	List of supported *Profiles*
organizationUri	String	A URI that identifies the organization that defined the profile.
profileId	String	A string that identifies the *Profile*
complianceTool	String	A string that identifies the tool or certification method used for compliance testing.
complianceDate	UtcTime	Date and time of the compliance test.
complianceLevel	enum Compliance Level	An enumeration that specifies the compliance level of the *Profile*. It has the following values : UNTESTED_0 the profiled capability has not been tested successfully PARTIAL_1 the profiled capability has been partially 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 by the certifying authority.
unsupportedUnitIds[]	String	The identifiers for the optional conformance units that were not tested. See Part 7 for a detailed explanation.

7.33 StatusCode

A *StatusCode* in OPC UA is numerical value that is used to report the outcome of an operation performed by an OPC 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 an OPC UA *Service*.

The *StatusCode* is a 32-bit unsigned integer. The top 16 bits represent the numeric value of the code that shall 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 OPC UA *Clients* shall always check the *StatusCode* associated with a result before using it. Results that have an uncertain/warning status associated with them shall be used with care since these results might not be valid in all situations. Results with a bad/failed status shall never be used.

OPC UA *Servers* should return good/success *StatusCodes* if the operation completed normally and the result is always valid. Different *StatusCode* values can provide additional information to the *Client*.

OPC 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 163.

Table 163 – StatusCode Bit Assignments

Field	Bit Range	Description
Severity	30:31	Indicates whether the *StatusCode* represents a good, bad or uncertain condition. These bits have the following meanings: Good Success — 00 — Indicates that the operation was successful and the associated results may be used. Uncertain Warning — 01 — Indicates that the operation was partially successful and that associated results might not be suitable for some purposes. Bad Failure — 10 — Indicates that the operation failed and any associated results cannot be used. Reserved — 11 — Reserved for future use. All *Clients* should treat a *StatusCode* with this severity as “Bad”.
Reserved	29:28	Reserved for future use. Shall 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 OPC UA specification refer to the symbolic name. Part 6 maps the symbolic names onto a numeric value.
StructureChanged	15:15	Indicates that the structure of the associated data value has changed since the last *Notification*. *Clients* should not process the data value unless they re-read the metadata. *Servers* shall set this bit if the *DataTypeEncoding* used for a *Variable* changes. 7.23 describes how the *DataTypeEncoding* is specified for a *Variable*. The bit is also set if the data type *Attribute* of the *Variable* changes. A *Variable* with data type *BaseDataType* does not require the bit to be set when the data type changes. *Servers* shall also set this bit if the *ArrayDimensions* or the *ValueRank Attribute* or the *EnumStrings Property* of the *DataType* of the *Variable* changes. This bit is provided to warn *Clients* that parse complex data values that their parsing routines could fail because the serialized form of the data value has changed. This bit has meaning only for *StatusCodes* returned as part of a data change *Notification* or the *HistoryRead*. *StatusCodes* used in other contexts shall always set this bit to zero.
SemanticsChanged	14:14	Indicates that the semantics of the associated data value have changed. *Clients* should not process the data value until they re-read the metadata associated with the *Variable*. *Servers* should set this bit if the metadata has changed in way that could cause application errors if the *Client* does not re-read the metadata. For example, a change to the engineering units could create problems if the *Client* uses the value to perform calculations. Part 8 defines the conditions where a *Server* shall set this bit for a DA *Variable*. Other specifications may define additional conditions. A *Server* may define other conditions that cause this bit to be set. This bit has meaning only for *StatusCodes* returned as part of a data change *Notification* or the *HistoryRead*. *StatusCodes* used in other contexts shall always set this bit to zero.
Reserved	12:13	Reserved for future use. Shall always be zero.
InfoType	10:11	The type of information contained in the info bits. These bits have the following meanings: NotUsed — 00 — The info bits are not used and shall be set to zero. DataValue — 01 — The *StatusCode* and its info bits are associated with a data value returned from the *Server*. Reserved — 1X — Reserved for future use. The info bits shall be ignored.
InfoBits	0:9	Additional information bits that qualify the *StatusCode*. The structure of these bits depends on the Info Type field.

Table 164 describes the structure of the *InfoBits* when the Info Type is set to *DataValue* (01).

Table 164 – 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:: **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 *MonitoredItem* reached its limit and had to purge out data.
Reserved	5:6	Reserved for future use. Shall 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 *Client* uses the data value. The historian bits have the following meaning: 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) Part 11 describes how these bits are used in more detail.

Common *StatusCodes*

Table 165 defines the common *StatusCodes* for all *Service* results. Part 6 maps the symbolic names to a numeric value.

Table 165 – Common Service Result Codes

Symbolic Id	**Description**
Good	The operation was successful.
Good_CompletesAsynchronously	The processing will complete asynchronously.
Good_SubscriptionTransferred	The subscription was transferred to another session.
Bad_CertificateHostNameInvalid	The *HostName* used to connect to a *Server* does not match a *HostName* in the *Certificate*.
Bad_CertificateIssuerRevocationUnknown	It was not possible to determine if the Issuer *Certificate* has been revoked.
Bad_CertificateIssuerUseNotAllowed	The Issuer *Certificate* may not be used for the requested operation.
Bad_CertificateIssuerTimeInvalid	An Issuer *Certificate* has expired or is not yet valid.
Bad_CertificateIssuerRevoked	The Issuer *Certificate* has been revoked.
Bad_CertificateInvalid	The certificate provided as a parameter is not valid.
Bad_CertificateRevocationUnknown	It was not possible to determine if the *Certificate* has been revoked.
Bad_CertificateRevoked	The *Certificate* has been revoked.
Bad_CertificateTimeInvalid	The *Certificate* has expired or is not yet valid.
Bad_CertificateUriInvalid	The URI specified in the *ApplicationDescription* does not match the URI in the *Certificate*.
Bad_CertificateUntrusted	The *Certificate* is not trusted.
Bad_CertificateUseNotAllowed	The *Certificate* may not be used for the requested operation.
Bad_CommunicationError	A low level communication error occurred.
Bad_DataTypeIdUnknown	The extension object cannot be (de)serialized because the data type id is not recognized.
Bad_DecodingError	Decoding halted because of invalid data in the stream.
Bad_EncodingError	Encoding halted because of invalid data in the objects being serialized.
Bad_EncodingLimitsExceeded	The message encoding/decoding limits imposed by the stack have been exceeded.
Bad_IdentityTokenInvalid	The user identity token is not valid.
Bad_IdentityTokenRejected	The user identity token is valid but the server has rejected it.
Bad_InternalError	An internal error occurred as a result of a programming or configuration error.
Bad_InvalidArgument	One or more arguments are invalid. Each service defines parameter-specific *StatusCodes* and these *StatusCodes* shall be used instead of this general error code. This error code shall be used only by the communication stack and in services where it is defined in the list of valid *StatusCodes* for the service.
Bad_InvalidState	The operation cannot be completed because the object is closed, uninitialized or in some other invalid state.
Bad_InvalidTimestamp	The timestamp is outside the range allowed by the server.
Bad_NonceInvalid	The nonce does appear to be not a random value or it is not the correct length.
Bad_NothingToDo	There was nothing to do because the client passed a list of operations with no elements.
Bad_OutOfMemory	Not enough memory to complete the operation.
Bad_RequestCancelledByClient	The request was cancelled by the client.
Bad_RequestTooLarge	The request message size exceeds limits set by the server.
Bad_ResponseTooLarge	The response message size exceeds limits set by the client.
Bad_RequestHeaderInvalid	The header for the request is missing or invalid.
Bad_ResourceUnavailable	An operating system resource is not available.
Bad_SecureChannelIdInvalid	The specified secure channel is not longer valid.
Bad_SecurityChecksFailed	An error occurred verifying security.
Bad_ServerHalted	The server has stopped and cannot process any requests.
Bad_ServerNotConnected	The operation could not complete because the client is not connected to the server.
Bad_ServerUriInvalid	The *Server* URI is not valid.
Bad_ServiceUnsupported	The server does not support the requested service.
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_Shutdown	The operation was cancelled because the application is shutting down
Bad_SubscriptionIdInvalid	The subscription id is not valid.
Bad_Timeout	The operation timed out.
Bad_TimestampsToReturnInvalid	The timestamps to return parameter is invalid.
Bad_TooManyOperations	The request could not be processed because it specified too many operations.

Bad_UnexpectedError	An unexpected error occurred.
Bad_UnknownResponse	An unrecognized response was received from the server.
Bad_UserAccessDenied	User does not have permission to perform the requested operation.
Bad_ViewIdUnknown	The view id does not refer to a valid view Node.
Bad_ViewTimestampInvalid	The view timestamp is not available or not supported.
Bad_ViewParameterMismatchInvalid	The view parameters are not consistent with each other.
Bad_ViewVersionInvalid	The view version is not available or not supported.

Table 166 defines the common *StatusCodes* for all operation level results. Part 6 maps the symbolic names to a numeric value. The common *Service* result codes can be also contained in the operation level.

Table 166 – Common Operation Level Result Codes

Symbolic Id	**Description**
Good_Clamped	The value written was accepted but was clamped.
Good_Overload	Sampling has slowed down due to resource limitations.
Uncertain	The value is uncertain but no specific reason is known
Bad	The value is bad but no specific reason is known.
Bad_AttributeIdInvalid	The attribute is not supported for the specified node.
Bad_BrowseDirectionInvalid	The browse direction is not valid.
Bad_BrowseNameInvalid	The browse name is invalid.
Bad_ContentFilterInvalid	The content filter is not valid.
Bad_ContinuationPointInvalid	The continuation point provide is longer valid.
Bad_DataEncodingInvalid	The data encoding is invalid.
Bad_DataEncodingUnsupported	The server does not support the requested data encoding for the node.
Bad_EventFilterInvalid	The event filter is not valid.
Bad_FilterNotAllowed	A monitoring filter cannot be used in combination with the attribute specified.
Bad_FilterOperandInvalid	The operand used in a content filter is not valid.
Bad_HistoryOperationInvalid	The history details parameter is not valid.
Bad_HistoryOperationUnsupported	The server does not support the requested operation.
Bad_IndexRangeInvalid	The syntax of the index range parameter is invalid.
Bad_IndexRangeNoData	No data exists within the range of indexes specified.
Bad_MonitoredItemFilterInvalid	The monitored item filter parameter is not valid.
Bad_MonitoredItemFilterUnsupported	The server does not support the requested monitored item filter.
Bad_MonitoredItemIdInvalid	The monitoring item id does not refer to a valid monitored item.
Bad_MonitoringModeInvalid	The monitoring mode is invalid.
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_NoContinuationPoints	The operation could not be processed because all continuation points have been allocated.
Bad_NodeClassInvalid	The node class is not valid.
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_NoDeleteRights	The *Server* will not allow the node to be deleted.
Bad_NodeNotInView	The nodeToBrowse is not part of the view.
Bad_NotFound	A requested item was not found or a search operation ended without success.
Bad_NotImplemented	Requested operation is not implemented.
Bad_NotReadable	The access level does not allow reading or subscribing to the *Node*.
Bad_NotSupported	The requested operation is not supported.
Bad_NotWritable	The access level does not allow writing to the *Node*.
Bad_ObjectDeleted	The object cannot be used because it has been deleted.
Bad_OutOfRange	The value was out of range.
Bad_ReferenceTypeIdInvalid	The reference type id does not refer to a valid reference type node.
Bad_SourceNodeIdInvalid	The source node id does not refer to a valid node.
Bad_StructureMissing	A mandadatory structured parameter was missing or null.

Bad_TargetNodeIdInvalid	The target node id does not refer to a valid node.
Bad_TypeDefinitionInvalid	The type definition node id does not reference an appropriate type node.
Bad_TypeMismatch	The value supplied for the attribute is not of the same type as the attribute's value.
Bad_WaitingForInitialData	Waiting for the server to obtain values from the underlying data source. After creating a *MonitoredItem*, it may take some time for the server to actually obtain values for these items. In such cases the server can optionally send a *Notification* with this status prior to the *Notification* with the first valid value.

7.34 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 167.

Table 167 – TimestampsToReturn Values

Value	Description
SOURCE_0	Return the source timestamp. If used in *HistoryRead* the source timestamp is used to determine which historical data values are returned.
SERVER_1	Return the *Server* timestamp. If used in *HistoryRead* the *Server* timestamp is used to determine which historical data values are returned.
BOTH_2	Return both the source and *Server* timestamps. If used in *HistoryRead* the source timestamp is used to determine which historical data values are returned.
NEITHER_3	Return neither timestamp. This is the default value for *MonitoredItems* if a *Variable* value is not being accessed. For *HistoryRead* this is not a valid setting.

7.35 UserIdentityToken parameters

7.35.1 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 168 defines the current set of user identity tokens. The *ExtensibleParamter* type is defined in 7.11.

Table 168 – UserIdentityToken parameterTypeIds

Symbolic Id	Description
AnonymousIdentityToken	No user information is available.
UserNameIdentityToken	A user identified by user name and password.
X509IdentityToken	A user identified by an X509v3 *Certificate*.
IssuedIdentityToken	A user identified by a WS-*SecurityToken*.

The *Client* shall always prove possession of a *UserIdentityToken* when it passes it to the *Server*. Some tokens include a secret such as a password which the *Server* will accept as proof. In order to protect these secrets the *Token* shall be encrypted before it is passed to the *Server*. Other types of tokens allow the *Client* to create a signature with the secret associated with the *Token*. In these cases, the *Client* proves possession of a *UserIdentityToken* by appending the last *ServerNonce* to the *ServerCertificate* and uses the secret to produce a *Signature* which is passed to the *Server*.

Each *UserIdentityToken* allowed by an *Endpoint* shall have a *UserTokenPolicy* specified in the *EndpointDescription*. The *UserTokenPolicy* specifies what *SecurityPolicy* to use when encrypting or signing. If this *SecurityPolicy* is omitted then the *Client* uses the *SecurityPolicy* in the *EndpointDescription*. If the matching *SecurityPolicy* is set to None then no encryption or signature is required. It is recommended that Applications never set the SecurityPolicy to None for UserTokens that include a secret because these secrets could be used by an attacker to gain access to the system.

Table 169 describes how to serialize *UserIdentityTokens* before applying encryption.

Table 169 – UserIdentityToken Encrypted Token Format

Name	Type	Description
length	Byte[4]	The length of the encrypted data including the *ServerNonce* but excluding the *length* field. This field is a 4 byte unsigned integer encoded with the least significant bytes appearing first.
tokenData	Byte[*]	The token data.
serverNonce	Byte[*]	The last *ServerNonce* returned by the server in the *CreateSession* or *ActivateSession* response.

7.35.2 AnonymousIdentityToken

The AnonymousIdentityToken is used to indicate that the Client has no user credentials.

Table 170 defines the AnonymousIdentityToken parameter.

Table 170 – AnonymousIdentityToken

Name	Type	Description
AnonymousIdentityToken	structure	An anonymous user identity.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.

7.35.3 UserNameIdentityToken

The *UserNameIdentityToken* is used to pass simple username/password credentials to the *Server*.

This token shall be encrypted if required by the *SecurityPolicy*. The *Server* should specify a *SecurityPolicy* for the *UserTokenPolicy* if the *SecureChannel* has a *SecurityPolicy* of None

If the token is encrypted password shall be converted to a UTF8 *ByteString* and then serialized as shown in Table 169.

The *Server* shall decrypt the password and verify the *ServerNonce*.

If the SecurityPolicy is None then the password only contains the UTF-8 encoded password.

Table 171 defines the UserNameIdentityToken parameter.

Table 171 – UserNameIdentityToken

Name	Type	Description
UserNameIdentityToken	structure	UserName value.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.
userName	String	A string that identifies the user.
password	ByteString	The password for the user. This parameter shall be encrypted with the Server's Certificate using the algorithm specified by the *SecurityPolicy.*
encryptionAlgorithm	String	A string containing the URI of the *encryptionAlgorithm*. The URI string values are defined as part of the security profiles specified in Part 7. This parameter is null if the password is not encrypted.

7.35.4 X509IdentityToken

The X509IdentiyToken is used to pass an X509v3 *Certificate* which is issued by the user.

This token shall always be accompanied by a signature if required by the *SecurityPolicy*. The *Server* should specify a *SecurityPolicy* for the *UserTokenPolicy* if the *SecureChannel* has a *SecurityPolicy* of None.

Table 172 defines the X509IdentityToken parameter.

Table 172 – X509IdentityToken

Name	Type	Description
X509IdentityToken	structure	X509v3 value.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.
certificateData	ByteString	The X509 *Certificate* in DER format.

7.35.5 IssuedIdentityToken

The *IssuedIdentityToken* is used to pass WS-Security compliant *SecurityToken*s 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 shall be exchanged in OPC UA as XML Security Tokens.

The WSS X509 and UserName tokens should not be exchanged as XML security tokens. OPC UA applications should use the appropriate OPC UA identity tokens to pass the information contained in these types of WSS *SecurityTokens.*

These tokens may be encrypted or require a signature. Part 7 defines profiles that include user related secuity, they also include any requirements for encryption and signatures. Additional security profiles specify encryption and signature algorithms.

If the token is encrypted then the XML shall be converted to an UTF8 *ByteString* and then serialized as shown in Table 169.

The *Server* shall decrypt the tokenData and verify the *ServerNonce*.

If the SecurityPolicy is None or if the token only requires signing then the tokenData contains the UTF-8 encoded XML representation of the token.

Table 173 defines the IssuedIdentityToken parameter.

Table 173 – IssuedIdentityToken

Name	Type	Description
IssuedIdentityToken	structure	WSS value.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.
tokenData	ByteString	The XML representation of the token. This parameter may be encrypted with the Server’s Certificate.
encryptionAlgorithm	String	A string containing the URI of the *encryptionAlgorithm*. The URI string values are defined as part of the security profiles specified in Part 7. This parameter is null if the token is not encrypted.

7.36 UserTokenPolicy

The components of this parameter are defined in Table 174.

Table 174 – UserTokenPolicy

Name	**Type**	**Description**
UserTokenPolicy	structure	Specifies a *UserIdentityToken* that a *Server* will accept.
policyId	String	An identifier for the UserTokenPolicy assigned by the Server. The Client specifies this value when it constructs a UserIdentityToken that conforms to the policy. This value is only unique within the context of a single Server.
tokenType	Enum UserIdentity TokenType	The type of user identity token required. This value is an enumeration with one of the following values: ANONYMOUS_0 No token is required. USERNAME_1 A username/password token. CERTIFICATE_2 An X509v3 certificate token. ISSUEDTOKEN_3 Any WS-Security defined token. A tokenType of ANONYMOUS indicates that the *Server* does not require any user identification. In this case the *Client* application instance *Certificate* is used as the user identification.
issuedTokenType	String	This field may only be specified if *TokenType* is ISSUEDTOKEN. A URI for the type of token. Part 7 defines URIs for supported token types. Vendors may specify their own token. WS-Security tokens are sometimes identified by XML QualifiedNames. A URI for the token can be constructed by appending the name to namespace with a ‘:’ separator. The XML QualifiedName can be reconstructed by searching for the last ‘:’ delimiter.
issuerEndpointUrl	String	A optional URL for the token issuing service. The meaning of this value depends on the *issuedTokenType*
securityPolicyUri	String	The security policy to use when encrypting or signing the *UserToken* when it is passed to the *Server* in the *ActivateSession* request. Section 7.35 describes how this parameter is used. The security policy for the SecureChannel is used if this value is omitted.

7.37 ViewDescription

The components of this parameter are defined in Table 175.

Table 175 – ViewDescription

Name	**Type**	**Description**
ViewDescription	structure	Specifies a *View.*
viewId	NodeId	*NodeId* of the *View* to *Query*. A null value indicates the entire *AddressSpace.*
timestamp	UtcTime	The time date desired. The corresponding version is the one with the closest previous creation timestamp. Either the *Timestamp* or the *viewVersion* parameter may be set by a *Client*, but not both. If *ViewVersion* is set this parameter shall be null.
viewVersion	UInt32	The version number for the *View* desired. When *Nodes* are added to or removed from a *View*, the value of a View’s *ViewVersion Property* is updated. Either the *Timestamp* or the *viewVersion* parameter may be set by a *Client*, but not both. The ViewVersion *Property* is defined in Part 3. If *timestamp* is set this parameter shall be 0. The current view is used if timestamp is null and viewVersion is 0.

Annex A (informative): 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 pathes and `|´ to identify alternatives. The ‘(‘ and ‘)’ symbols are used it indicate sets.

A.2 BNF of RelativePath

A *RelativePath* is a structure that describes a sequence of *References* and *Nodes* to follow. This Appendix describes a text format for a RelativePath that can be used in documentation or in files used to store configuration information.

The components of a *RelativePath* text format are specified in Table 176.

Table 176 – RelativePath

Symbol	Meaning
/	The forward slash character indicates that the *Server* is to follow any subtype of *HierarchicalReferences*.
.	The period (dot) character indicates that the *Server* is to follow any subtype of a *Aggregates ReferenceType*.
<[#!ns:]ReferenceType>	A string delimited by the ‘<’ and ‘>’ symbols specifies the *BrowseName* of a *ReferenceType* to follow. By default, any *References* of the subtypes the *ReferenceType* are followed as well. A ‘#’ placed in front of the BrowseName indicates that subtypes should not be followed. A ‘!’ in front of the BrowseName is used to indicate that the inverse *Reference* should be followed. The *BrowseName* may be qualified with a namespace index (indicated by a numeric prefix followed by a colon). This namespace index is used specify the namespace component of the *BrowseName* for the *ReferenceType.* If the namespace prefix is omitted then namespace index 0 is used.
[ns:]BrowseName	A string that follows a ‘/’, ‘.’ or ‘>’ symbol specifies the *BrowseName* of a target *Node* to return or follow. This BrowseName may be prefixed by its namespace index. If the namespace prefix is omitted then namespace index 0 is used. Omitting the final *BrowseName* from a path is equivalent to a wildcard operation that matches all *Nodes* which are the target of the *Reference* specified by the path.
&	The & sign character is the escape character. It is used to specify reserved characters that appear within a *BrowseName*. A reserved character is escaped by inserting the ‘&’ in front of it. Examples of *BrowseNames* with escaped characters are: Received browse path name — Resolves to “&/Name_1” — “/Name_1” “&.Name_2” — “.Name_2” “&:Name_3” — “:Name_3” “&&Name_4” — “&Name_4”

Table 177 provides examples of *RelativePaths* specified with the text format.

Table 177 – RelativePath Examples

Browse Path	Description
"/2:Block&.Output"	Follows any forward hierarchical *Reference* with target *BrowseName* = "2:Block.Output".
"/3:Truck.0:NodeVersion"	Follows any forward hierarchical *Reference* with target *BrowseName* = "3:Truck" and from there a forward *Aggregates Reference* to a target with *BrowseName* "0:NodeVersion".
"<1:ConnectedTo>1:Boiler/1:HeatSensor"	Follows any forward Reference with a *BrowseName* = '1:ConnectedTo' and finds targets with *BrowseName* = '1:Boiler'. From there follows any hierarchical *Reference* and find targets with *BrowseName* = '1:HeatSensor'.
"<1:ConnectedTo>1:Boiler/"	Follows any forward Reference with a *BrowseName* = '1:ConnectedTo' and finds targets with *BrowseName* = '1:Boiler'. From there it finds all targets of hierarchical *References* .
"<0:HasChild>2:Wheel"	Follows any forward Reference with a *BrowseName* = 'HasChild' and qualified with the default OPC UA namespace. Then find targets with *BrowseName* = 'Wheel' qualified with namespace index '2'.
"<!HasChild>Truck"	Follows any inverse Reference with a *BrowseName* = 'HasChild' (i.e. follows the *HasParent Reference*). Then find targets with *BrowseName* = 'Truck'. In both cases, the namespace component of the *BrowseName* is assumed to be 0.
"<0:HasChild>"	Finds all targets of forward *References* with a *BrowseName* = 'HasChild' and qualified with the default OPC UA namespace.

The following BNF describes the syntax of the *RelativePath* text format.

```
<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 Part 3) excluding reserved-chars.
```

A.3 BNF of NumericRange

The following BNF describes the syntax of the NumericRange parameter type.

```
<numeric-range>  ::= <dimension> [',' <dimension>]

<dimension>      ::= <index> [':' <index>]

<index>          ::= <digit> [<digit>]

<digit>          ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9'
```

Annex B (informative): Content Filter and Query Examples

B.1 Simple ContentFilter examples

B.1.1 Overview

These examples provide fairly simple content filters. Filter similar to these examples may be used in processing events.

The following conventions apply to these examples with regard to how Attribute operands are used (for a definition of this operand see 7.4.4):

- AttributeOperand: Refers to a *Node,* an *Attribute* of a *Node* or the *Value Attribute* of a *Property* associated with a *Node*. In the examples character names of NodeIds are used instead of an actual nodeId, this also applies to Attribute Ids.
- The string representation of relative paths is used instead of the actual structure.
- The NamespaceIndex used in all examples is 12 (it could just as easily have been 4 or 23 or any value). For more information about NamespaceIndex see Part 3. The use of the NamespaceIndex illustrates that the information model being used in the examples is not a model defined by this Specification, but one created for the examples.

B.1.2 Example 1

For example the logic describe by '(((AType.A = 5) or InList(BType.B, 3,5,7)) and BaseObjectType.displayName LIKE "Main%")' would result in a logic tree as shown in Figure 29 and a ContentFilter as shown in Table 178. For this example to return anything AType and BType both must be sub types of BaseObjectType, or the resulting "And" operation would always be false.

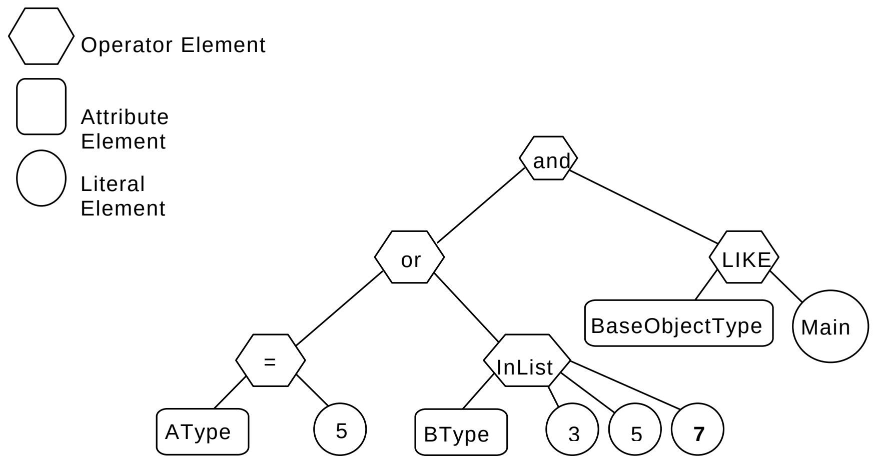

Figure 29 – Filter Logic Tree Example

Table 178 describes the elements, operators and operands used in the example.

Table 178 – 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	AttributeOperand = NodeId: AType, BrowsePath: ".12:A", AttributeId:value	LiteralOperand = '5'		
3	InList	AttributeOperand = NodeId: BType, BrowsePath: ".12:B", AttributeId:value	LiteralOperand = '3'	LiteralOperand = '5'	LiteralOperand = '7'
4	Like	AttributeOperand = NodeId: BaseObjectType, BrowsePath: ".", AttributeId: displayName	LiteralOperand = "Main%"		

B.1.3 Example 2

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 30 and a ContentFilter as shown in Table 179.

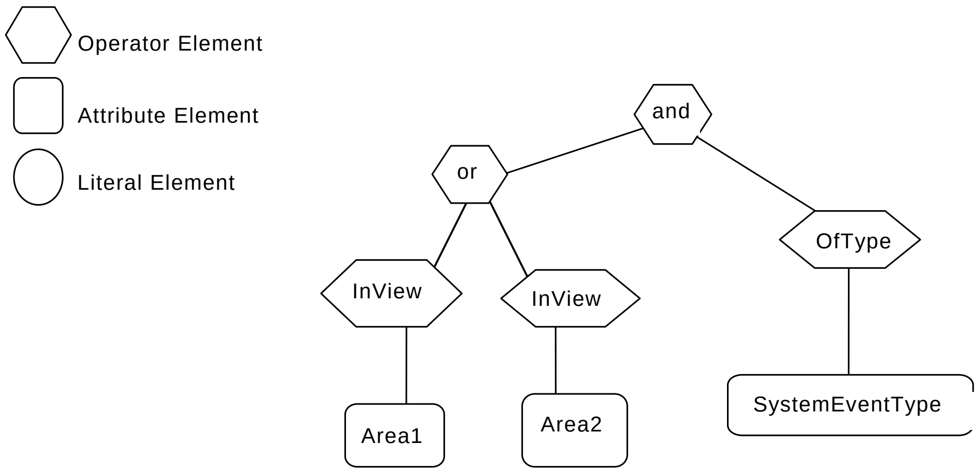

Figure 30 – Filter Logic Tree Example

Table 179 describes the elements, operators and operands used in the example.

Table 179 – ContentFilter Example

Element[]	Operator	Operand[0]	Operand[1]
0	And	ElementOperand = 1	ElementOperand = 4
1	Or	ElementOperand = 2	ElementOperand = 3
2	InView	AttributeOperand = NodeId: Area1, BrowsePath: ".", AttributeId: NodeId	
3	InView	AttributeOperand = NodeId: Area2, BrowsePath: ".", AttributeId: NodeId	
4	OfType	AttributeOperand = NodeId: SystemEventType, BrowsePath: ".", AttributeId: NodeId"	

B.2 Complex Examples of ContentFilters (Queries)

B.2.1 Overview

These query examples illustrate complex *ContentFilters.* The following conventions apply to these examples with regard to Attribute operands (for a definition of these operands see 7.4.4):

- AttributeOperand: Refers to a *Node*, an *Attribute* of a *Node* or the *Value Attribute* of a *Property* associated with a *Node*. In the examples character names of NodeIds are used instead of an actual nodeId, this also applies to Attribute Ids.
- The string representation of relative paths is used instead of the actual structure.
- The NamespaceIndex used in all examples is 12 (it could just as easily have been 4 or 23 or any value). For more information about NameSpacesIndex see Part 3. The use of the NamespaceIndex illustrates that the information model being used in the examples is not a model defined by this Specification, but one created for the examples.

B.2.2 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 31. In this Figure, the OPC UA 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*.

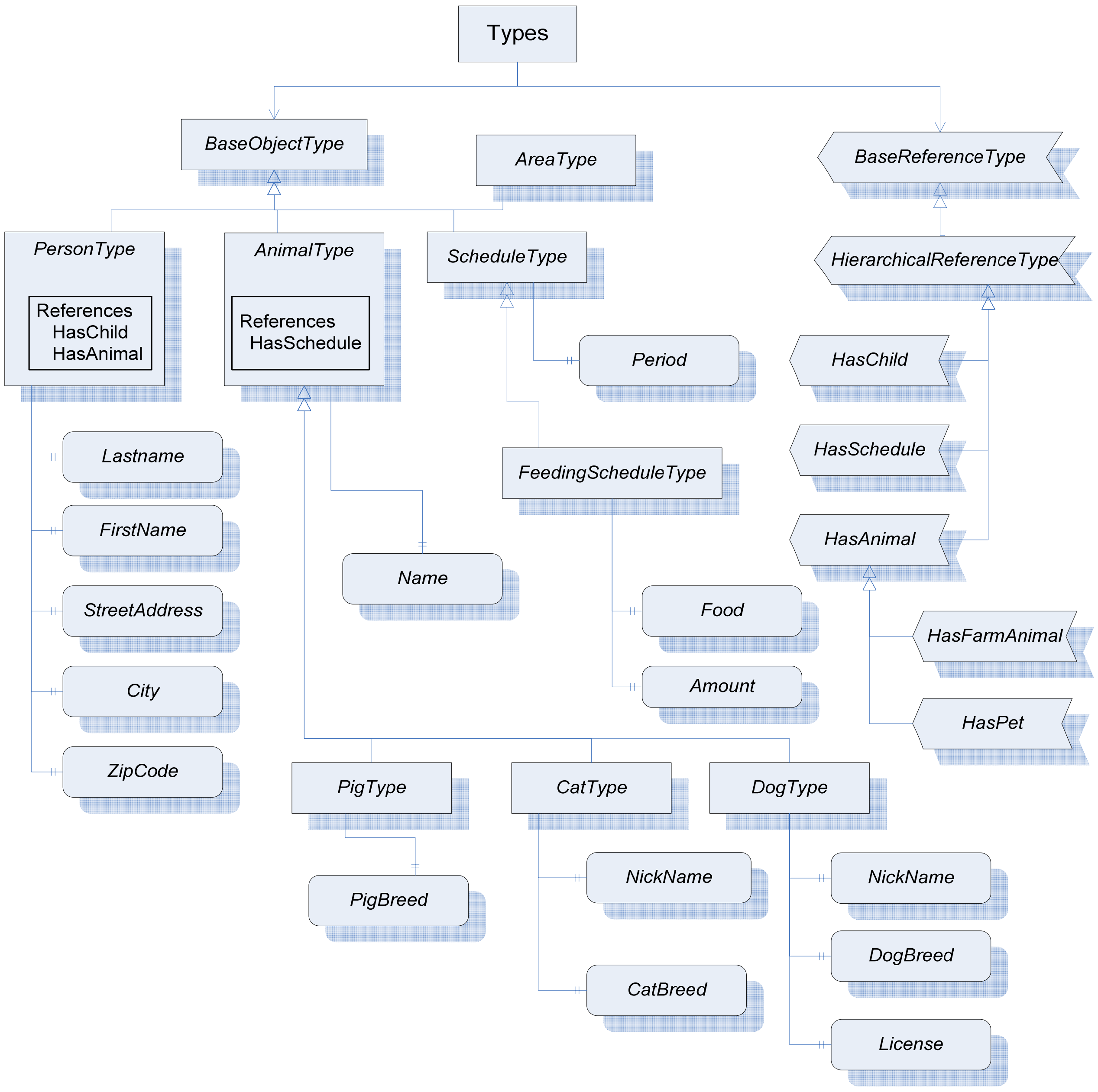

Figure 31 – Example Type Nodes

A corresponding example set of instances is shown in Figure 32. 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. Defined *Reference* types use the OPC UA 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. All custom nodes are defined in namespace 12 which is not included in the Figure.

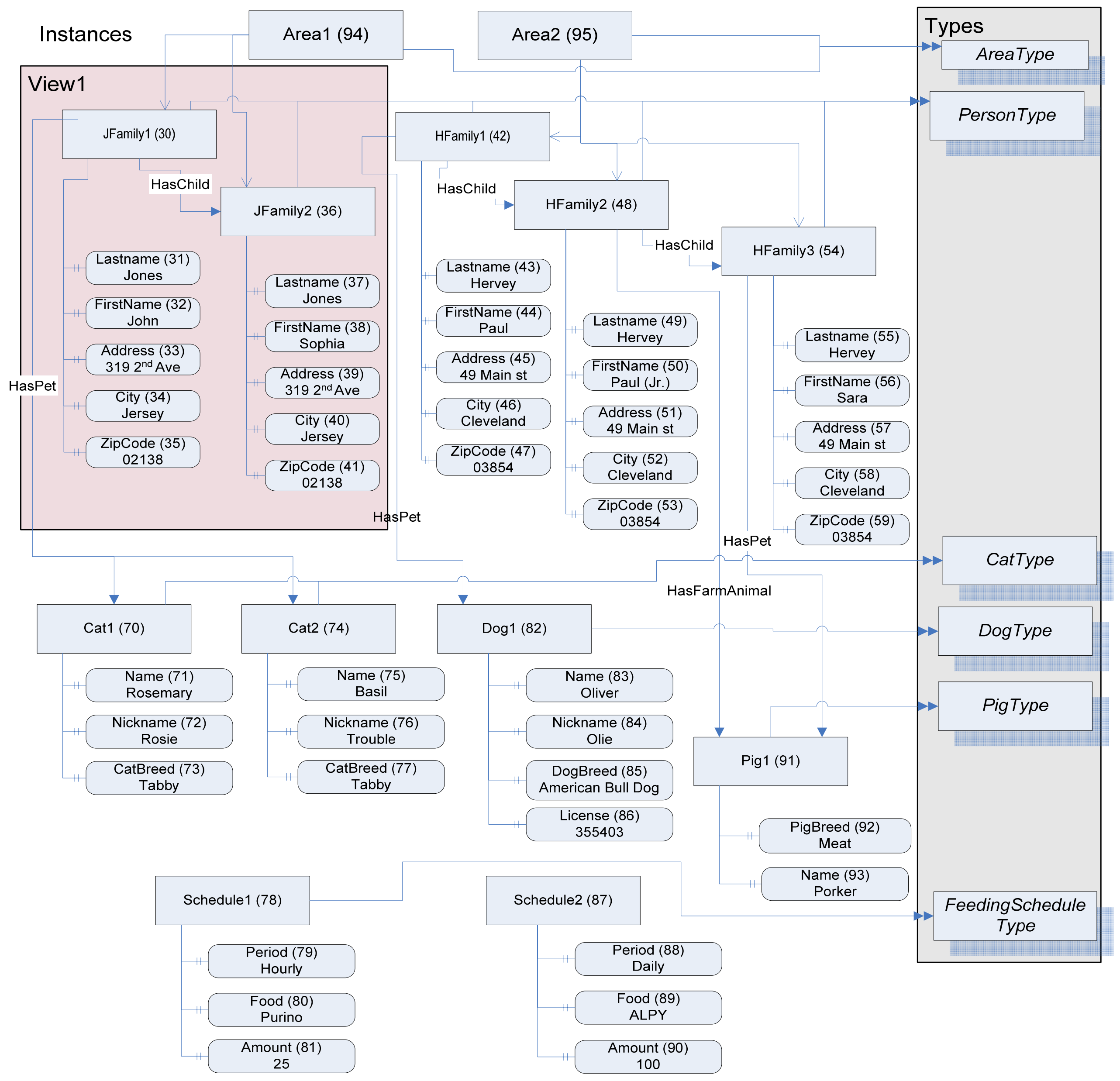

Figure 32 - Example Instance Nodes

B.2.3 Example Notes

For all of the examples in 7.4.4, the type definition *Node* is listed in its symbolic form, in the actual call it would be the *NodeId* assigned to the *Node*. The *AttributeId* 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.

The examples assume namespace 12 is the namespace for all of the custom definitions described for the examples.

B.2.4 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 180.

Table 180 – Example 1 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".12:LastName"	value	N/A
		"<12:HasPet>12:AnimalType. 12:name"	value	N/A
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule. 12:period"	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 33.

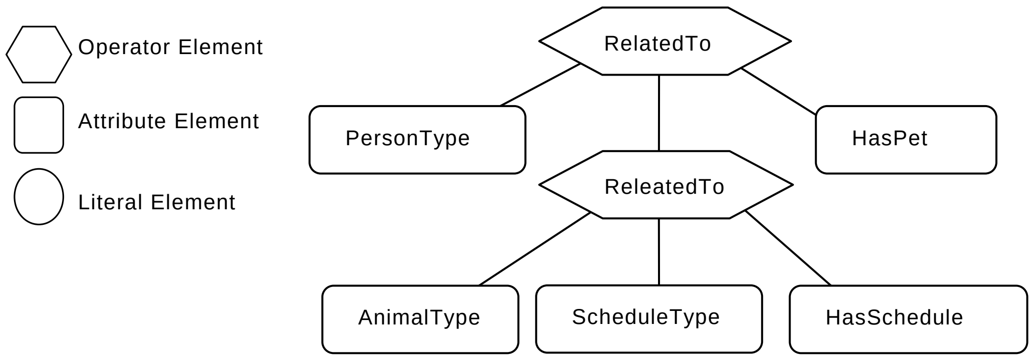

Figure 33 - Example 1 Filter

Table 181 describes the *ContentFilter* elements, operators and operands used in the example.

Table 181 – Example 1 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
1	RelatedTo	AttributeOperand = Nodeid: PersonType, BrowsePath ".", AttributeId: NodeId	ElementOperand = 2	AttributeOperand = NodeId: HasPet, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = NodeId: AnimalType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: ScheduleType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: HasSchedule, BrowsePath ".", AttributeId: NodeId	LiteralOperand= '1'

Table 182 describes the *QueryDataSet* that results from this query if it were executed against the instances described in Figure 32.

Table 182 – Example 1 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (JFamily1)	PersonType	".12:lastName"	Jones
		"<12:HasPet>12:AnimalType. 12:name"	Rosemary
			Basil
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule.12:period"	Hourly
			Hourly
12:42(HFamily1)	PersonType	".12:lastName"	Hervey
		"<12:HasPet>12:AnimalType. 12:name"	Olive
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule.12: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].

B.2.5 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 183.

Table 183 – Example 2 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".12:LastName"	value	N/A
AnimalType	TRUE	".12:name"	value	N/A

The corresponding ContentFilter is illustrated in Figure 34.

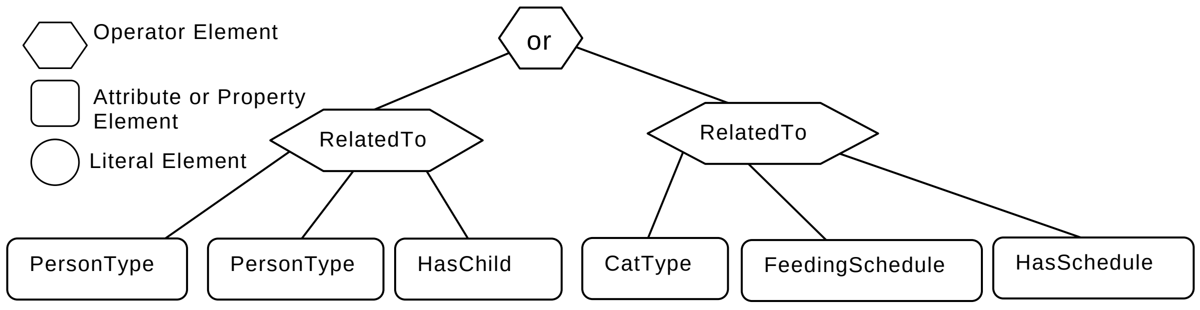

Figure 34 – Example 2 Filter Logic Tree

Table 184 describes the elements, operators and operands used in the example. It is worth noting that a Cattype is a subtype of Animaltype.

Table 184 – Example 2 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: PersonType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: PersonType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: HasChild, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = NodeId: CatType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: FeedingScheduleType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: HasSchedule, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 185.

Table 185 – Example 2 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (Jfamily1)	Persontype	. 12:lastName	Jones
12:42 (HFamily1)	PersonType	. 12:lastName	Hervey
12:48 (HFamily2)	PersonType	. 12:lastName	Hervey
12:70 (Cat1)	CatType	. 12:name	Rosemary
12:74 (Cat2)	CatType	. 12: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 a *NodeId* not the symbolic name that is included in the table].

B.2.6 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 and Amount to feed is > 10.

Table 186 describes the NodeTypeDescription parameters used in the example.

Table 186 – Example 3 - NodeTypeDescriptions

Type Definition Node	Include Subtypes	RelativePath	Attribute Id	Index Range
PersonType	FALSE	"12:PersonType.12:lastName"	Value	N/A
		"12:PersonType<12:HasPet>12:AnimalType. 12:name"	Value	N/A
		"12:PersonType<12:HasPet>12:AnimalType<12:HasSchedule> 12:FeedingSchedule.period"	Value	N/A

The corresponding *ContentFilter* is illustrated in Figure 35.

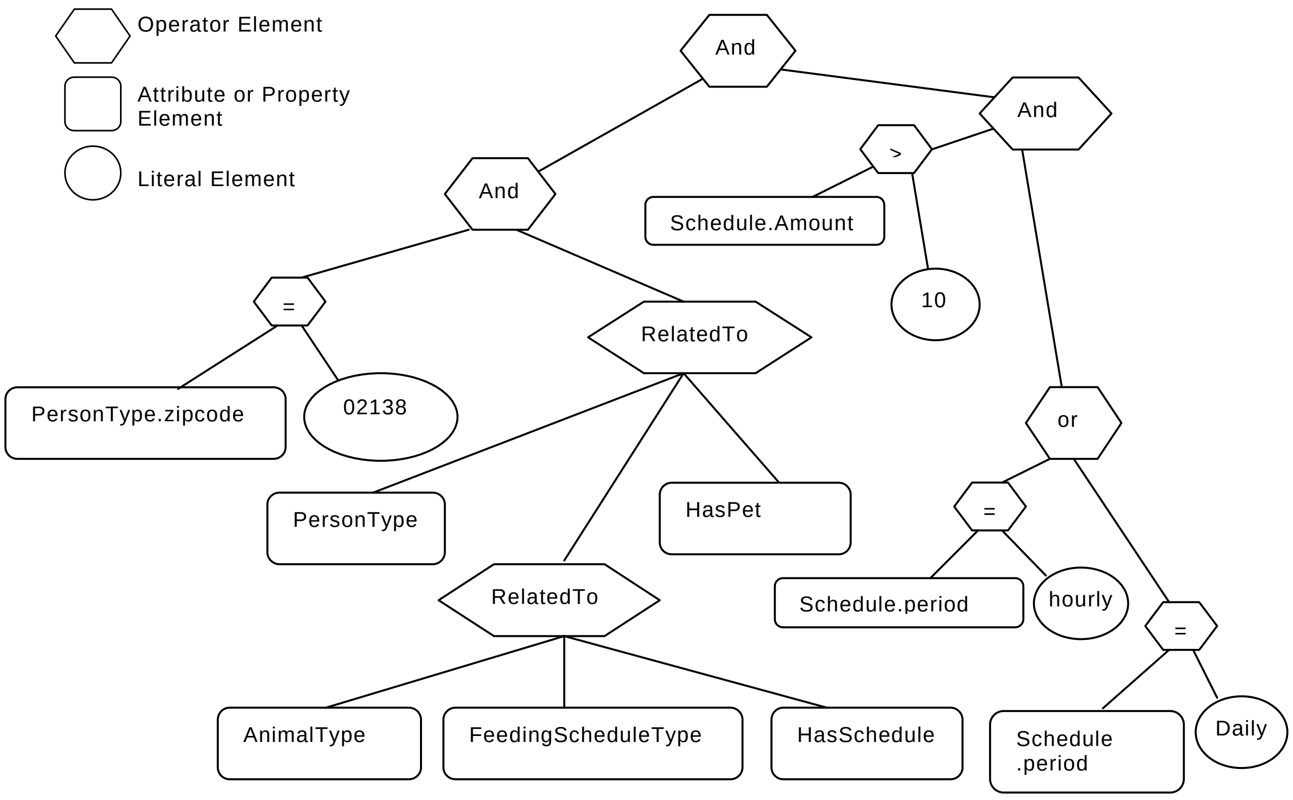

Figure 35 – Example 3 Filter Logic Tree

Table 187 describes the elements, operators and operands used in the example.

Table 187 – 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	And	ElementOperand = 3	ElementOperand = 9		
3	Or	ElementOperand = 7	ElementOperand = 8		
4	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath “.”, AttributeId: NodeId	ElementOperand = 5	AttributeOperand = NodeId: 12:HasPet, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’
5	RelatedTo	AttributeOperand = Node: 12:AnilmalType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’
6	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath “.”, AttributeId: zipcode	LiteralOperand = ‘02138’		
7	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath “12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule”, AttributeId: Period	LiteralOperand = ‘Daily’		
8	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath “12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule”, AttributeId: Period	LiteralOperand = ‘Hourly’		
9	Greater Than	AttributeOperand = NodeId: 12:PersonType BrowsePath “12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule”, AttributeId: Amount	ElementOperand = 10		
10	Cast	LiteralOperand = 10	AttributeOperand = NodeId: String, BrowsePath “.”, AttributeId: NodeIdt		

The results from this query would contain the *QueryDataSets* shown in Table 188.

Table 188 – Example 3 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (JFamily1)	PersonType	“.12:lastName”	Jones
		“<12:hasPet>12:PersonType. 12:name”	Rosemary
			Basil
		“<12:hasPet>12:AnimalType<12:hasSchedule>12:FeedingSchedule. 12:period”	Hourly
			Hourly

[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].

B.2.7 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 189 describes the NodeTypeDescription parameters used in the example.

Table 189 – Example 4 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".12:lastName"	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 36.

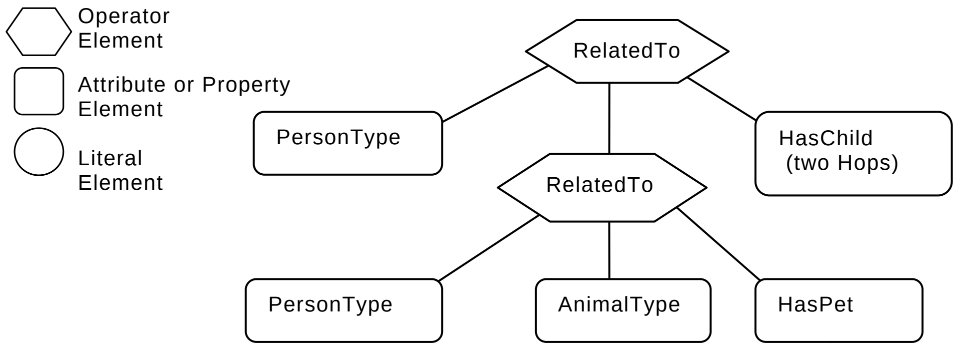

Figure 36 – Example 4 Filter Logic Tree

Table 190 describes the elements, operators and operands used in the example.

Table 190 – Example 4 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId	Element Operand = 1	AttributeOperand = NodeId: 12:HasChild, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '2'
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:AnimalType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:HasPet, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table 191. It is worth noting that the pig "Pig1" is referenced as a pet by Sara, but is referenced as a farm animal by Sara's parent Paul.

Table 191 – Example 4 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	".12:lastName"	Hervey

[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].

B.2.8 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 192 describes the NodeTypeDescription parameters used in the example.

Table 192 – Example 5 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	"<12:HasChild>12:PersonType. 12:lastName"	Value	N/A

The corresponding *ContentFilter* is illustrated in Figure 37.

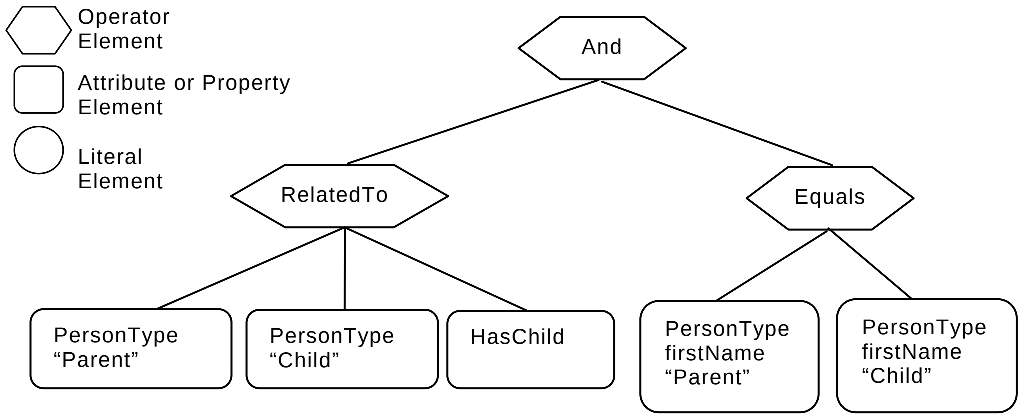

Figure 37 – Example 5 Filter Logic Tree

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 193 describes the elements, operators and operands used in the example.

Table 193 – Example 5 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3}
0	And	ElementOperand = 1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId, Alias: "Parent"	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId, Alias: "Child"	AttributeOperand = NodeId: 12:HasChild, AttributeId: NodeId	LiteralOperand = "1"
2	Equals	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: FirstName, Alias: "Parent"	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: firstName, Alias: "Child"		

The results from this query would contain the *QueryDataSets* shown in Table 194.

Table 194 – Example 5 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	"<12:HasChild>12:PersonType.12:lastName"	Hervey

B.2.9 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.NodeId, AnimalType.NodeId, PersonType.HasChild Reference, PersonType.HasAnimal Reference where a person has a child who has a Animal.

Table 195 describes the NodeTypeDescription parameters used in the example.

Table 195 – Example 6 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".12:NodeId"	value	N/A
		<12:HasChild>12:PersonType <12:HasAnimal>12:AnimalTyp e.NodeId	value	N/A
		<12:HasChild>	value	N/A
		<12:HasChild>12:PersonType <12:HasAnimal>	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 38.

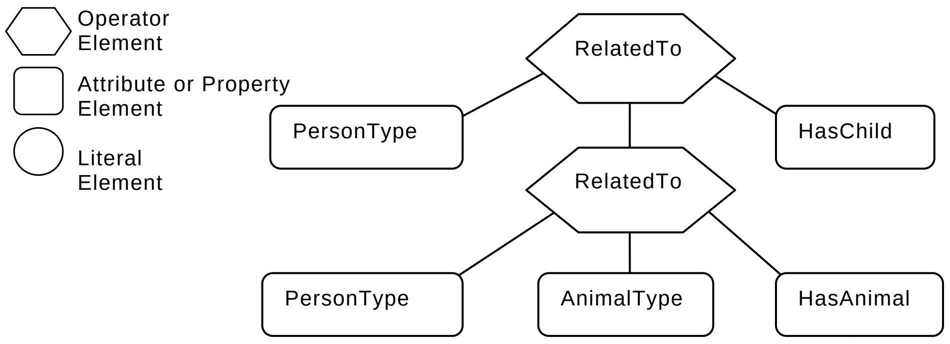

Figure 38 – Example 6 Filter Logic Tree

Table 196 describes the elements, operators and operands used in the example.

Table 196 – Example 6 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId	ElementOperand = 1	AttributeOperand = Node: 12:HasChild, BrowsePath ".",AttributeId:NodeId	LiteralOpera nd = '1'
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:AnimalType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:HasAnimal, BrowsePath ".", AttributeId: NodeId	LiteralOpera nd = '1'

The results from this query would contain the *QueryDataSets* shown in Table 197.

Table 197 – Example 6 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	".NodeId"	12:42 (HFamily1)
		<12:HasChild>12:PersonType<12:HasAnimal> 12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasChild>12:PersonType<12:HasAnimal>	HasFarmAnimal *ReferenceDescription*
12:48 (HFamily2)	PersonType	".NodeId"	12:48 (HFamily2)
		<12:HasChild>12:PersonType<12:HasAnimal> 12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasChild>12:PersonType<12: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 TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table. The value field would in this case be the *NodeId* 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 197, the value would be a *ReferenceDescription* which are described in 7.24].

Table 198 provides an example of the same QueryDataSet as shown in Table 197 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 comas. 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 32.

Table 198 – Example 6 QueryDataSets without Additional Information

NodeId	TypeDefinition NodeId	Value
12:42	PersonType	12:42
		12:91
		[HasChild,TRUE,[48,HFamily2,HFamily2,PersonType]],
		[HasFarmAnimal,TRUE[91,Pig1,Pig1,PigType]
12:48	PersonType	12:54
		12:91
		[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 *ExpandedNodeIds*.

B.2.10 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 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.NodeId, AnimalType.NodeId, PersonType.HasChild Reference, PersonType.HasAnimal Reference where the person is in Area2 (Cleveland nodes) and the person has a child.

Table 199 describes the NodeTypeDescription parameters used in the example.

Table 199 – Example 7 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".NodeId"	value	N/A
		<12:HasChild>	value	N/A
		<12:HasAnimal>NodeId	value	N/A
		<12:HasAnimal>	value	N/A

The corresponding *ContentFilter* is illustrated in Figure 39. Note the *Browse* call would typically return a *NodeId*, thus the first filter is for the *BaseObjectType* with a *NodeId* of 95 where 95 is the *NodeId* associated with the Area2 node, all *Nodes* descend from *BaseObjectType*, and *NodeId* is a base *Property* so this filter will work for all *Queries* of this nature.

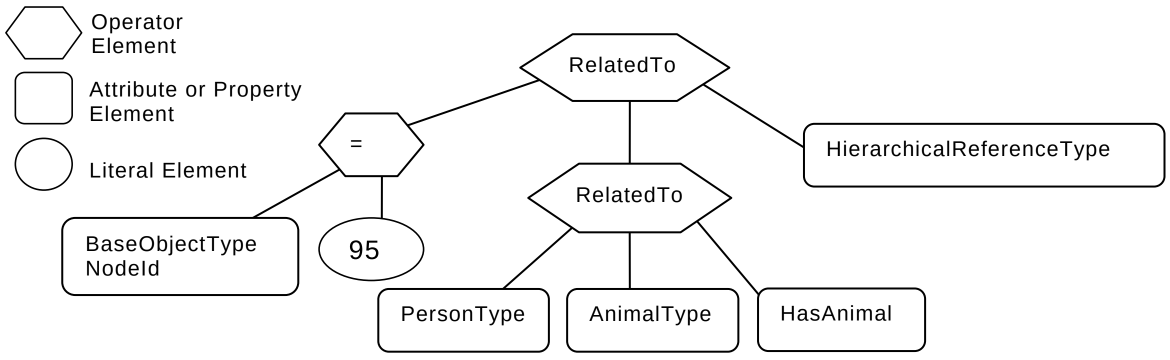

Figure 39 – Example 7 Filter Logic Tree

Table 200 describes the elements, operators and operands used in the example.

Table 200 – Example 7 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	RelatedTo	AttributeOperand = NodeId: BaseObjectType, BrowsePath ".", AttributeId: NodeId	ElementOperand = 1	AttributeOperand = Node:HierachicalReference, BrowsePath ".", AttributeId:NodeId	LiteralOperand = '1'
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:PersonTyp, BrowsePath ".", AttributeId: NodeId	AttributeOperand = NodeId: 12:HasChild, BrowsePath ".", AttributeId: NodeId	LiteralOperand = '1'
2	Equals	AttributeOperand = NodeId: BaseObjectType, BrowsePath ".", AttributeId: NodeId,	LiteralOperand = '95		

The results from this *Query* would contain the *QueryDataSets* shown in Table 201.

Table 201 – Example 7 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	".NodeId"	12:42 (HFamily1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasAnimal>12:AnimalType.NodeId	NULL
		<12:HasAnimal>	HasFarmAnimal *ReferenceDescription*
12:48 (HFamily2)	PersonType	".NodeId"	12:48 (HFamily2)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasAnimal>12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasAnimal>	HasFarmAnimal *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 TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table. The value field would in this case be the *NodeId* 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 201, the value would be a *ReferenceDescription* which are described in 7.24].

B.2.11 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 202

Table 202 – Example 8 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	".12:LastName"	value	N/A
AnimalType	TRUE	".name"	value	N/A

The corresponding ContentFilter is illustrated in Figure 40.

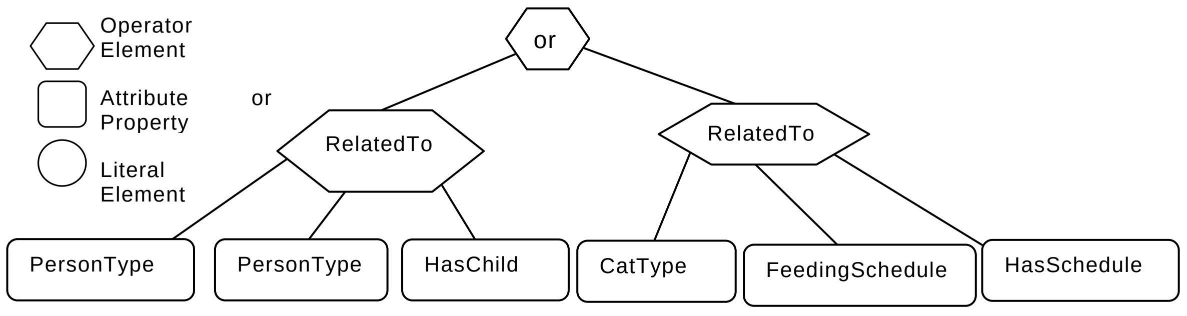

Figure 40 – Example 8 Filter Logic Tree

Table 203 describes the elements, operators and operands used in the example. It is worth noting that a CatType is a subtype of AnimalType.

Table 203 – Example 8 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:PersonType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:HasChild, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’
2	RelatedTo	AttributeOperand = NodeId: 12:CatType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’

The results from this query would contain the *QueryDataSets* shown in Table 204. 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 include in the result set

Table 204 – Example 8 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (Jfamily1)	Persontype	.12:LastName	Jones

[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].

B.2.12 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 205 describes the NodeTypeDescription parameters used in the example.

Table 205 – Example 9 NodeTypeDescription

Type Definition Node	Include Subtypes	Relative Path	Attribute Id	Index Range
PersonType	FALSE	“.NodeId”	value	N/A
		<12:HasChild>12:PersonType<12:HasAnimal>12:AnimalType.NodeId	value	N/A
		<12:HasChild>	value	N/A
		<12:HasChild>12:PersonType <12:HasAnimal>	value	N/A
PersonType	FALSE	“.12:LastName”	value	N/A
		<12:HasAnimal>12:AnimalType. 12:Name	value	N/A
AnimalType	TRUE	“.12:name”	value	N/A

The corresponding ContentFilter is illustrated in Figure 41.

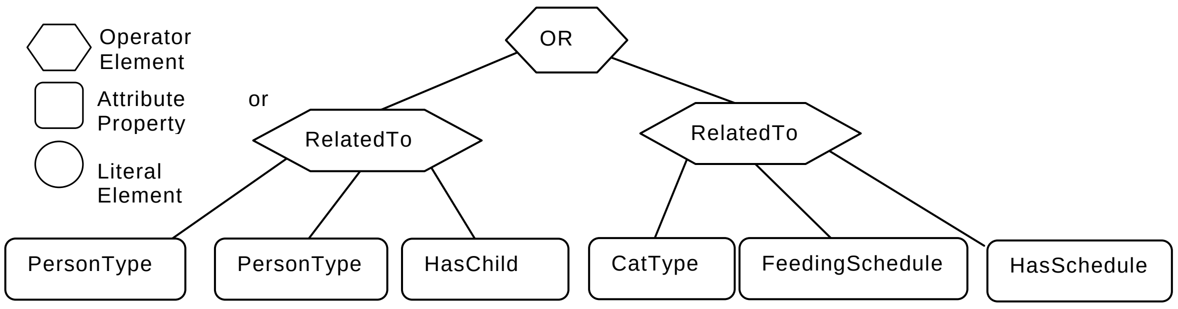

Figure 41 – Example 9 Filter Logic Tree

Table 206 describes the elements, operators and operands used in the example.

Table 206 – Example 9 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:PersonType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:HasChild, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’
2	RelatedTo	AttributeOperand = NodeId: 12:CatType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath “.”, AttributeId: NodeId	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath “.”, AttributeId: NodeId	LiteralOperand = ‘1’

The results from this *Query* would contain the *QueryDataSets* shown in Table 207. 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 207 – Example 9 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (Jfamily1)	Persontype	. 12:LastName	Jones
		<12:HasAnimal>12:AnimalType. 12:Name	Rosemary
		<12:HasAnimal>12:AnimalType. 12: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].