OPC FOUNDATION

# OPC Unified Architecture

Specification

Part 4: Services

Version 1.01.05

February 01, 2007

## CONTENTS

# 1 Scope

This specification specifies the OPC Unified Architecture *Services*.

# 2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later
http://www.opcfoundation.org/UA/Part1/

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part2/

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part3/

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part5/

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

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

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later
http://www.opcfoundation.org/UA/Part8/

[UA Part 11] OPC UA Specification: Part 11 – Historical Access, Version 1.0 or later
http://www.opcfoundation.org/UA/Part11/

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

# 3 Terms, definitions, and conventions

## 3.1 OPC UA Part 1 terms

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

1) AddressSpace
2) Attribute
3) Certificate
4) Client
5) Communication Stack
6) Event
7) EventNotifier
8) Message
9) MonitoredItem
10) Node
11) NodeClass
12) Notification

13) NotificationMessage
14) Object
15) ObjectType
16) Profile
17) Reference
18) ReferenceType
19) Server
20) Service
21) Service Set
22) Session
23) Subscription
24) Variable
25) View

## 3.2 OPC UA Part 2 terms

The following terms defined in [UA Part 2] apply.

1) Authentication
2) Authorization
3) Confidentiality
4) Integrity
5) Nonce
6) OPC UA Application
7) SecureChannel
8) SecurityToken
9) SessionKeySet
10) PrivateKey
11) PublicKey
12) X.509 Certificate

## 3.3 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

1) EventType
2) HierarchicalReference
3) InstanceDeclaration
4) ModellingRule
5) Property
6) SourceNode
7) TargetNode
8) TypeDefinitionNode
9) VariableType

## 3.4 OPC UA Services terms

### 3.4.1 Deadband

A *Deadband* specifies a permitted range for value changes that will not trigger a data change *Notification*. It can be applied as filter when subscribing to *Variables* and is used to keep noisy signals from updating the *Client* unnecessarily.

This specification defines *AbsoluteDeadband* as a common filter. [UA Part 8] defines an additional *Deadband* filter.

### 3.4.2 Endpoint

An *Endpoint* is a physical address available on a network that allows *Clients* to access one or more *Services* provided by a *Server*. Each *Server* may have multiple *Endpoints*.

### 3.4.3 ServerUri

A *ServerUri* is a globally unique identifier for a *Server* application instance. It may be a *GUID* generated automatically during install or it could be a unique *URL* assigned by the administrator.

### 3.4.4 SoftwareCertificate

A digital certificate for a software product, which can be installed on several hosts to describe the capabilities of the software product. Different installations of one software product could have the same software certificate.

## 3.5 Abbreviations and symbols

| | |
|---|---|
| API | Application Programming Interface |
| BNF | Backus-Naur Form |
| CA | Certificate Authority |
| CRL | Certificate Revocation List |
| CTL | Certificate Trust List |
| DA | Data Access |
| UA | Unified Architecture |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |

## 3.6 Conventions for Service definitions

OPC UA *Services* contain parameters that are conveyed between the *Client* and the *Server*. The UA *Service* specifications use tables to describe *Service* parameters, as shown in Table 1. Parameters are organised in this table into request parameters and response parameters.

**Table 1 – Service Definition Table**

| Name | Type | Description |
|---|---|---|
| **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 [UA Part 3]. The base types used in *Services* are listed in Table 2. Data types that are common to multiple *Services* are defined in Clause 7 and Clause 8. Data types that are *Service*-specific are defined in the parameter table of the *Service*.

**Table 2 – Parameter Types defined in [UA Part 3]**

| Parameter Type |
|---|
| BaseDataType |
| NodeId |
| QualifiedName |
| LocaleId |
| Boolean |
| ByteString |
| Double |
| Guid |
| Int32 |
| String |
| UInt32 |
| UtcTime |
| XmlElement |

## 4 Overview

### 4.1 Service Set model

This clause specifies the OPC UA *Services*. The OPC UA *Service* definitions are abstract descriptions and do not represent a specification for implementation. The mapping between the abstract descriptions and the *Communication Stack* derived from these *Services* are defined in [UA Part 6]. In the case of an implementation as web services, the OPC UA *Services* correspond to the web service and an OPC UA *Service* corresponds to an operation of the web service.

These *Services* are organised into *Service Sets*. Each *Service Set* defines a set of related *Services*. The organisation in *Service Sets* is a logical grouping used in the specification and is not used in the implementation.

The *Discovery Service Set*, illustrated in Figure 1, defines *Services* that allow a *Client* to discover the *Endpoints* implemented by a *Server* and to read the security configuration for each of those *Endpoints*

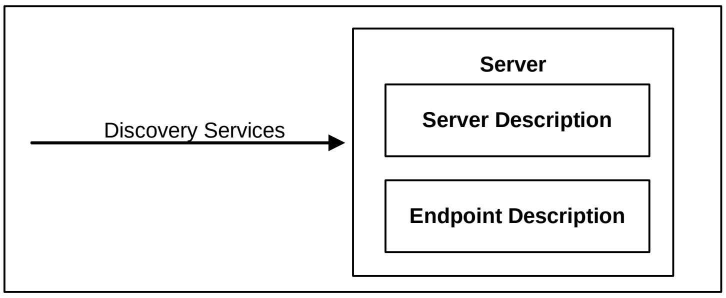

**Figure 1 – Discovery Service Set**

The *SecureChannel Service Set*, illustrated in Figure 2, defines *Services* that allow a *Client* to to establish a communication channel to ensure the *Confidentiality* and *Integrity* of *Messages* exchanged with the *Server*.

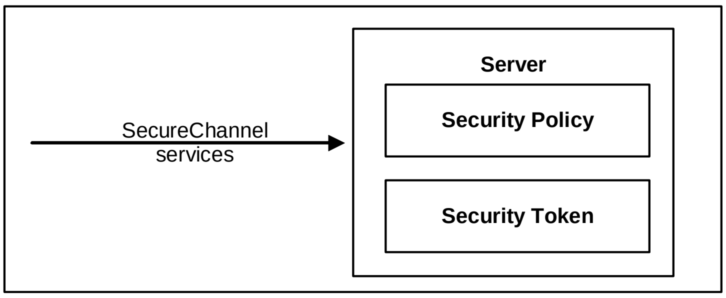

**Figure 2 – SecureChannel Service Set**

The *Session Service Set*, illustrated in Figure 3, defines *Services* that allow the *Client* to authenticate the User it is acting on behalf of and to manage *Sessions*.

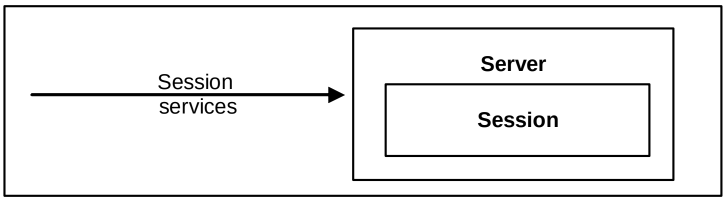

**Figure 3 – Session Service Set**

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

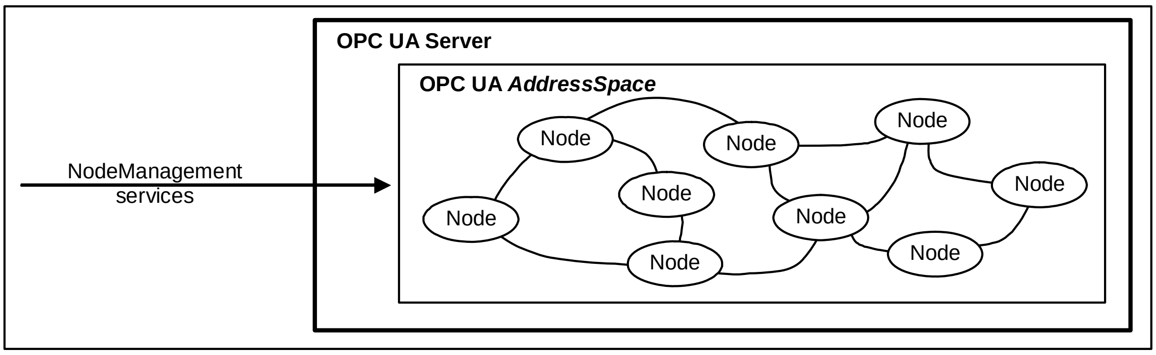

**Figure 4 – NodeManagement Service Set**

The *View Service Set*, illustrated in Figure 5, defines *Services* that allow *Clients* to browse through the *AddressSpace* or subsets of the *AddressSpace* called *Views*. The *Query Service Set* allows *Clients* to return a subset of data from the *View*.

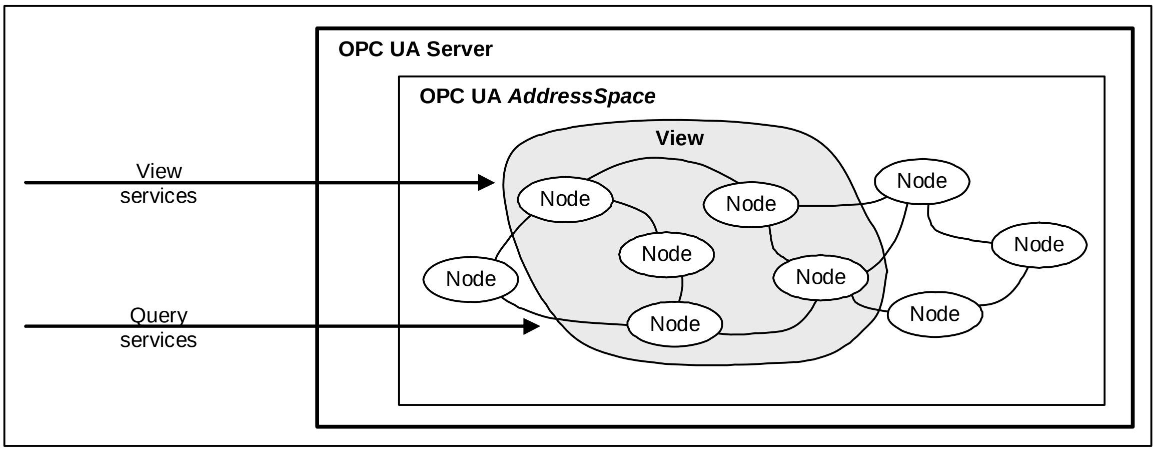

**Figure 5 – View Service Set**

The *Attribute Service Set* is illustrated in Figure 6. It defines *Services* that allow *Clients* to read and write *Attributes* of *Nodes*, including their historical values. Since the value of a *Variable* is modelled as an *Attribute*, these *Services* allow *Clients* to read and write the values of *Variables*.

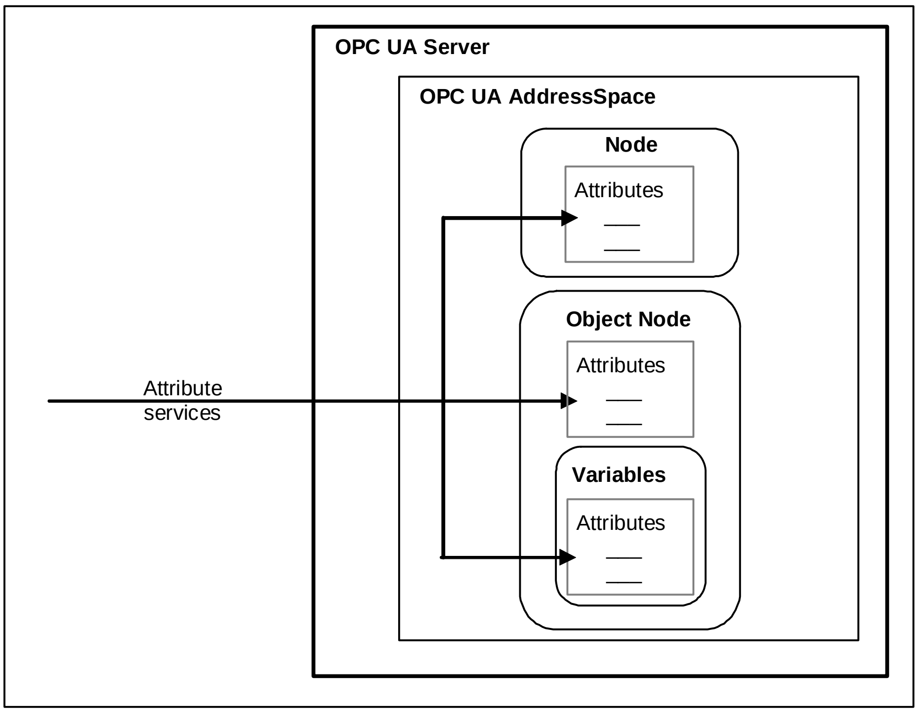

**Figure 6 – Attribute Service Set**

The *Method Service Set* is illustrated in Figure 7. It defines *Services* that allow *Clients* to call methods. Methods run to completion when called. They may be called with method-specific input parameters and may return method-specific output parameters.

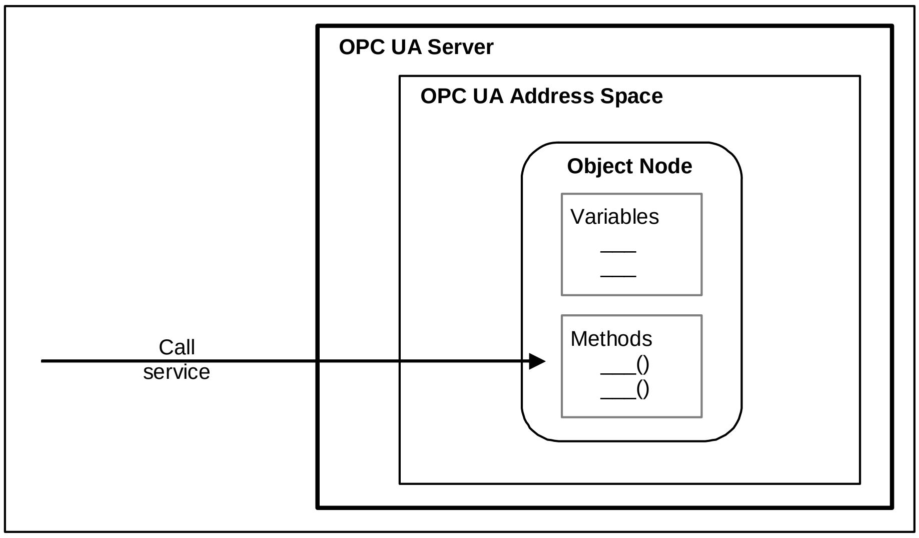

**Figure 7 – Method Service Set**

The *MonitoredItem Service Set* and the *Subscription Service Set*, illustrated in Figure 8, are used together to subscribe to *Nodes* in the OPC UA *AddressSpace*.

The *MonitoredItem Service Set* defines *Services* that allow *Clients* to create, modify, and delete *MonitoredItems* external to the OPC UA *AddressSpace*. *MonitoredItems* monitor *Attribut*es for value changes and *Nodes* for *Events*, and generate *Notifications* for them.

These *Notifications* are queued for transfer to the *Client* by *Subscriptions*.

The *Subscription Service Set* defines *Services* that allow *Clients* to create, modify and delete *Subscriptions*. *Subscriptions* send *Notifications* generated by *MonitoredItems* to the *Client*. *Subscription Services* also provide for *Client* recovery from missed *Messages* and communication failures.

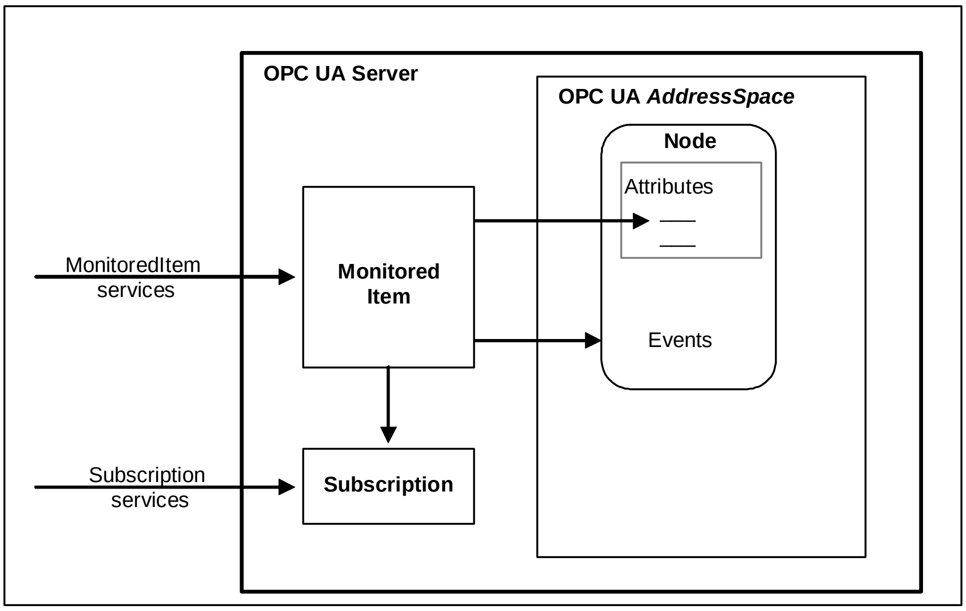

**Figure 8 – MonitoredItem and Subscription Service Sets**

## 4.2 Request/response Service procedures

Request/response *Service* procedures describe the processing of requests received by the *Server*, and the subsequent return of responses. The procedures begin with the requesting *Client* submitting a *Service* request *Message* to the *Server*.

Upon receipt of the request, the *Server* processes the *Message* in two steps. In the first step, it attempts to decode and locate the *Service* to execute. The error handling is specific to the communication technology used and is described in [UA Part 6].

If it succeeds, then it attempts to access each operation identified in the request and perform the requested operation. For each operation in the request, it generates a separate success/failure code that it includes in a positive response *Message* along with any data that is to be returned.

To perform these operations, both the *Client* and the *Server* may make use of the API of a *Communication Stack* to construct and interpret *Messages* and to access the requested operation.

The implementation of each service request or response handling must check that each service parameter lies within the specified range for that parameter.

# 5 Service Sets

## 5.1 General

This clause defines the OPC UA *Service Sets* and their *Services*. Clause 7 and Clause 8 contain the definitions of common parameters used by these *Services*.

Whether or not a *Server* supports a *Service Set*, or a *Service* within a *Service Set*, is defined by its *Profile*. *Profiles* are described in [UA Part 7].

## 5.2 Service request and response header

Each *Service* request has a *RequestHeader* and each *Service* response has a *ResponseHeader*.

The *RequestHeader* structure is defined in Clause 7.19 and contains common request parameters such as *sessionId*, *timestamp* and *sequenceNumber*.

The *ResponseHeader* structure is defined in Clause 7.20 and contains common response parameters such as *serviceResult* and *diagnosticInfo*.

## 5.3 Service results

*Service* results are returned at two levels in OPC UA responses, one that indicates the status of the *Service* call, and the other that indicates the status of each operation requested by the *Service*.

*Service* results are defined via the *StatusCode* (see Clause 7.28).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see Clause 7.20). The mechanism for returning this parameter is specific to the communication technology used to convey the *Service* response and is defined in [UA Part 6].

The status of partial operations in a request is represented by individual *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 Clause 7.23.
b) The good code is returned in *serviceResult* if the *Service* fully or partially succeeded. In this case, other response parameters are returned. The *Client* must always check the response parameters, especially all *StatusCodes* associated with each operation. These *StatusCodes* may indicate bad or uncertain results for one or more operations requested in the *Service* call.

All *Services* with arrays of operations in the request must return a bad code in the *serviceResult* if the array is empty. The bad *StatusCode* indicates which parameter was wrong.

The *Services* define various specific StatusCodes and a *Server* must use these specific *StatusCodes* as described in the *Service*.

If the *Server* discovers, through some out-of-band mechanism, that the application or user credentials used to create a *Session* or *SecureChannel* have been compromised, then the *Server* must immediately terminates all sessions and channels that use those credentials. In this case, the *Service* result code must be either *Bad_UserIdentityNoLongerValid* or *Bad_CertificateNoLongerValid*.

## 5.4 Discovery Service Set

### 5.4.1 Overview

This *Service Set* defines *Services* used to discover the *Endpoints* implemented by a *Server* and to read the security configuration for those *Endpoints*. The *Discovery Services* are implemented by individual *Servers* and by dedicated *Discovery Servers*. [UA Part 12] describes how to use the *Discovery Services* with dedicated *Discovery Servers.*

Every *Server* must have a *Discovery Endpoint* that *Clients* can access without establishing a *Session*. This *Endpoint* may or might not be the same *Session Endpoint* that *Clients* use to establish a *SecureChannel*. Clients read the security information necessary to establish a *SecureChannel* by calling the *GetEndpoints Service* on the *Discovery Endpoint*.

In addition, *Servers* may register themselves with a well known *Discovery Server* using the *RegisterServers* service. Clients can later discover any registered *Servers* by calling the *FindServers Service* on the *Discovery Server*.

The complete discovery process is illustrated in Figure 9.

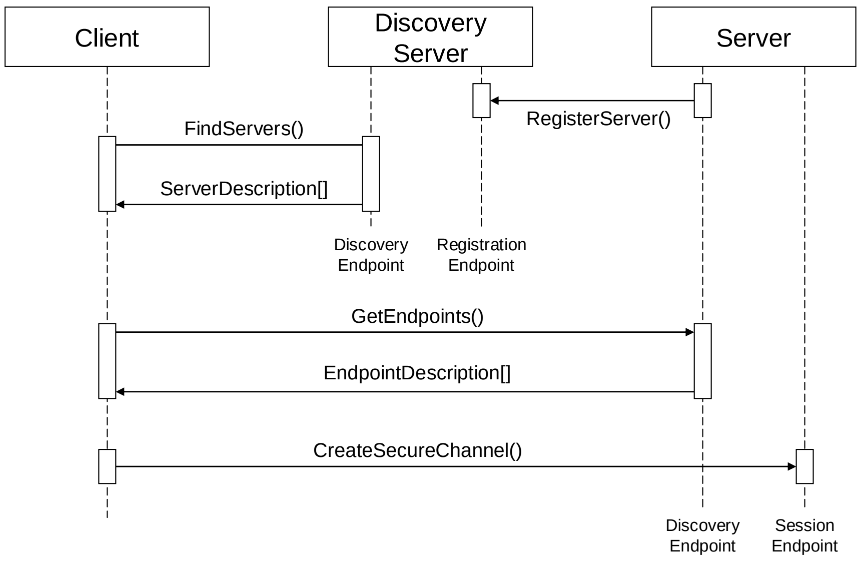

**Figure 9 – The Discovery Process**

The URL for a *Discovery Endpoint* must provide all of the information that the *Client* needs to connect to the *Endpoint*. This implies that no security can be applied to service call itself, however, some implementations may use transport layer security where the security protocol is identified in the URL (e.g. HTTPS).

Once a *Client* retrieves the *Endpoints*, the 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 that could mean the *Server* configuration has changed and the *Client* needs to go through the discovery process again.

### 5.4.2 FindServers

#### 5.4.2.1 Description

This *Service* returns the *Servers* known to a *Discovery Server*. The behavoir of *Discovery Servers* is described in detail in [UA Part 12].

The *Client* may reduce the number of results returned by specifying filter criteria. A *Discovery Server* returns an empty list if no *Servers* match the criteria specified by the client. The filter criteria supported by this *Service* are described in Clause 5.4.2.2.

Every *Server* must provide a *Discovery Endpoint* that supports this *Service*, however, the *Server* will only return a single record that describes itself.

Every *Server* must have a globally unique identifier called the *ServerUri*. This identifier should be a fully qualified domain name, however, it may be a GUID or similar construct that ensures global uniqueness. The *ServerUri* returned by this *Service* must be the same value that appears in index 0 of the *ServerArray* property (see [UA Part 5]).

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

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

#### 5.4.2.2 Parameters

Table 3 defines the parameters for the *Service*.

**Table 3 – FindServers Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters. The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.19. |
| localeIds [] | LocaleId | List of locales to use.<br>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 must use the locale that appears first in this list. If the server does not support any of the requested locales it chooses an appropriate default locale.<br>The server chooses an appropriate default locale if this list is empty. |
| serverUris [] | String | List of servers to return.<br>All known servers are returned if the list is empty. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters. The *sessionId* is always set to 0 in this request. The type *RequestHeader* is defined in Clause 7.20. |
| servers [] | ServerDescription | List of *Servers* that meet criteria specified in the request.<br>This list is empty if no servers meet the criteria.<br>The *ServerDescription* type is defined in Clause 7.23. |

#### 5.4.2.3 Service results

Common *StatusCodes* are defined in Table 156.

### 5.4.3 GetEndpoints

#### 5.4.3.1 Description

This *Service* returns the *Endpoints* supported by a *Server* and all of the configuration information required to establish a *SecureChannel and a Session*.

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

A *Client* may reduce the number of results returned by specifying filter criteria. The *Server* returns an empty list if no *Endpoints* match the criteria specified by the client. The filter criteria supported by this *Service* are described in Clause 5.4.3.2.

A *Server* may support multiple security configurations for the same *Endpoint*. In this situation, the *Server* must return separate *EndpointDescription* records for each available configuration. *Clients* should treat each of these configurations as distinct *Endpoints* even if the physical URL happens to be the same.

The security configuration for an *Endpoint* has four components:

1) Server Application Instance Certificate
2) Message Security Mode
3) Security Policy
4) Supported User Identity Tokens

The *ApplicationInstanceCertificate* is used to secure the *OpenSecureChannel* request (See Clause 5.5.2). The *MessageSecurityMode* and the *SecurityPolicy* tell the *Client* how to secure messages sent via the *SecureChannel*. The *UserIdentityTokens* tell the client what user credentials must be passed to the *Server* in the *CreateSession* request (See Clause 5.6.2).

Each *EndpointDescription* also specifies one or more URIs for supported *Profiles*. These values indicate which *Transport Profiles* defined in [UA Part 7] that the *Endpoint* supports*. Transport Profiles* specify information such as message encoding format and protocol version. The *Server* should only return *Profiles* that a *Client* needs to be aware of before it connects to the *Server*. *Clients* must fetch the *Server's SoftwareCertificates* if they want to discover the complete list of *Profiles* supported by the *Server* (See Clause 7.25).

Messages are secured by applying standard cryptography algorithms to the messages before they are sent over the network. The exact set of algorithms used depends on the *SecurityPolicy* for the Endpoint. [UA Part 7] defines *Profiles* for common *SecurityPolicies* and assigns a unique URI to them. It is expected that applications have built in knowledge of the *SecurityPolicies* that they support, as a result, only the Profile URI for the *SecurityPolicy* is specified in the *EndpointDescription*. A *Client* cannot connect to an *Endpoint* that does not support a *SecurityPolicy* that it recognizes.

An *EndpointDescription* may specify that the message security mode is NONE. This configuration is not recommended unless the applications are communicating on a physically isolated network where the risk of intrusion is extremely small. If the message security is NONE then it is possible for *Clients* to deliberately or accidentally hijack *Sessions* created by other *Clients*.

#### 5.4.3.2 Parameters

Table 4 defines the parameters for the *Service*.

**Table 4 – GetEndpoints Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters.<br>The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.19. |
| localeIds [] | LocaleId | List of locales to use.<br>Specifies the locale to use when returning human readable strings.<br>This parameter is described in Clause 5.4.2.2. |
| profileUris [] | String | List of profiles that the returned *Endpoints* must support.<br>All *Endpoints* are returned if the list is empty. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters.<br>The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.20. |
| Endpoints [] | EndpointDescription | List of *Endpoints* that meet criteria specified in the request.<br>This list is empty if no *Endpoints* meet the criteria. |
| endpointUrl | String | The URL for the *Endpoint* described. |
| server | ServerDescription | The description for the *Server* that the *Endpoint* belongs to.<br>The *ServerDescription* type is defined in Clause 7.23. |
| serverCertificate | ByteString | The application instance *Certificate* issued to the *Server*.<br>This is a DER encoded X509v3 certificate. |
| securityMode | Enum<br>MessageSecurityMode | The type of security to apply to the messages.<br>The type *MessageSecurityMode* type is defined in Clause 7.10.<br>A *SecureChannel* may have to be created even if the *securityMode* is NONE. The exact behavoir depends on the mapping used and is described in the [UA Part 6]. |
| securityPolicy | String | The URI for *SecurityPolicy* to use when securing messages.<br>This value identifies an instance of the S*ecurityPolicy* type described in Clause 7.22.<br>The set of known URIs and the S*ecurityPolicies* associated with them are defined in [UA Part 7]. |
| userIdentityTokens[] | UserTokenPolicy | The user identity tokens that the *Server* will accept.<br>The *Client* must pass one of the *UserIdentityTokens* in the *ActivateSession* request. The *UserTokenPolicy* type is described in Clause 7.30. |
| supportedProfiles | String | The URI of the *Transport Profile* supported by the *Endpoint*.<br>[UA Part 7] defines URIs for the standard *Transport Profiles*. |

#### 5.4.3.3 Service Results

Common *StatusCodes* are defined in Table 156.

### 5.4.4 RegisterServer

#### 5.4.4.1 Description

This *Service* registers a *Server* with a *Discovery Server*. This *Service* will be called by a *Server* or a separate configuration utility. *Clients* will not use this *Service*.

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

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

*Servers* must be able to register themselves with a *Discovery Server* running on the same machine. The exact mechanisms depend on the *Discovery Server* implementation and are described in [UA Part 6].

There are two types of *Server* applications: those which are manually launched and those that are automatically launched when a *Client* attempts to connect. The registration process that a Server must use depends on which category it falls into.

The registration process for manually launched Servers is illustrated in Figure 10.

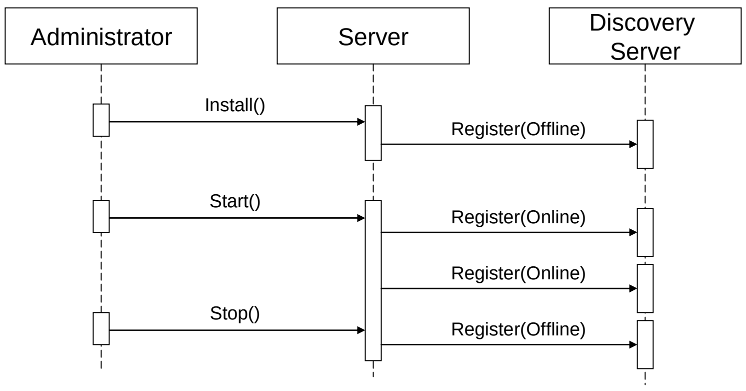

**Figure 10 – The Registration Process – Manually Launched Servers**

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

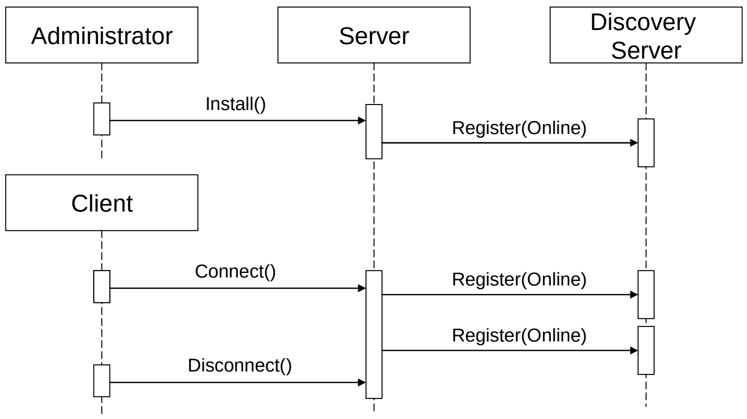

**Figure 11 – The Registration Process – Automatically Launched Servers**

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

Under normal conditions, *Servers* must periodically register with the *Discovery Server* as long as they are able to receive connections from *Clients*. If a *Server* goes offline then it must register itself once more and indicate that it is going offline. The registration frequency should be configurable, however, the default is 10 minutes.

If an error occurs during registration (e.g. the Discovery Server is not running) then the *Server* must periodically re-attempt registration. The frequency of these attempts should start at 1 second but gradually increase until the registration frequency is the same as what it would be if not errors occurred. The recommended approach would double the period each attempt until reaching the maximum.

When a *Server* registers with the a *Discovery Server* it may choose to provide a semaphore file which the *Discovery Server* can use to determine if the *Server* has been uninstalled from the machine. The *Discovery Server* must have read access to the file system that contains the file.

#### 5.4.4.2 Parameters

Table 5 defines the parameters for the *Service*.

**Table 5 – RegisterServers Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters.<br>The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.19. |
| server | RegisteredServer | The server to register. |
| serverUri | String | The globally unique identifier for the *Server*. |
| serverNames [] | LocalizedText | A list of localized descriptive names for the *Server*.<br>The list must have at least one valid entry. |
| discoveryUrls [] | String | A list of *Discovery Endpoints* for the *Server*.<br>The list must have at least one valid entry. |
| semaphoreFilePath | String | The path to the semaphore file used to identify the server instance.<br>The *Discovery Server* will always check that this file exists before returning the ServerDescription to the client.<br>If the same semaphore file is used by another *Server* then that registration is deleted and replaced by the one being passed into this method.<br>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.<br>The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.20. |

#### 5.4.4.3 Service Results

Table 6 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 6 – RegisterServers Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_ServerUriInvalid | The *ServerUri* is not a valid URI. |
| Bad_ServerNameMissing | No *ServerName* was specified. |
| Bad_DiscoveryUrlMissing | No *DiscoveryUrl* was specified. |
| Bad_SempahoreFileMissing | The semaphore file specified by the client is not valid. |

## 5.5 SecureChannel Service Set

### 5.5.1 Overview

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

The *SecureChannel Services* are unlike other *Services* because they are not implemented directly by the *UA Application*. Instead, they are provided by the *Communication Stack* on which the UA *Application* is built. For example, a UA *Server* may be built on a SOAP stack that allows applications to establish a *SecureChannel* using the WS Secure Conversation specification. In these cases, the *UA Application* must verify that the *Message* it received was in the context of a WS Secure Conversation. [UA Part 6] describes how the *SecureChannel Services* are implemented.

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

An *EndpointDescription* tells a Client how to establish a *SecureChannel* with a given *Endpoint*. A *Client* may obtain the *EndpointDescription* from a *Discovery Server*, via some non-UA defined directory server or from its own configuration.

The exact algorithms used to authenticate and encrypt *Messages* are described in the *SecurityPolicy* field of the *EndpointDescription*. A *Client* must use these algorithms when it creates a *SecureChannel*.

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

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

The correlation between the *UA Application Session* and the *SecureChannel* is illustrated in Figure 12. The *Communication Stack* is used by the *UA Applications* to exchange *Messages*. In a first step, the *SecureChannel Services* are used to establish a *SecureChannel* between the two *Communication Stacks* to exchange *Messages* in a secure way. In a second step, the *UA Applications* use the *Session Service Set* to establish a *UA Application Session*.

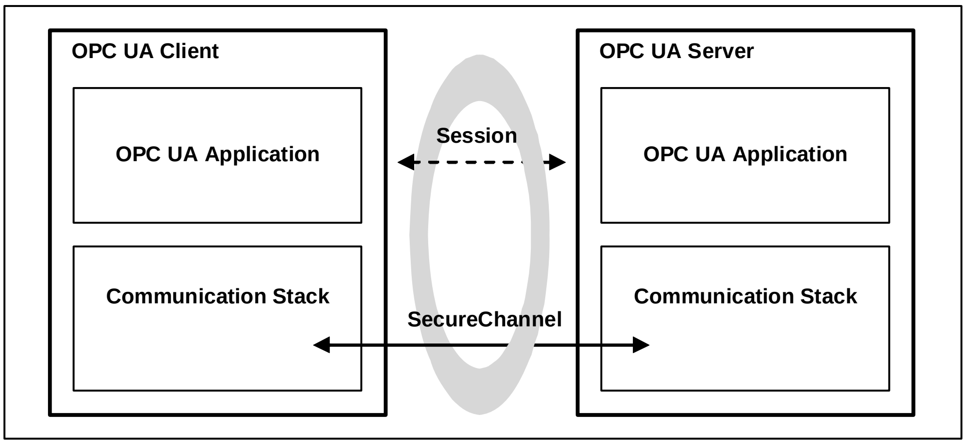

**Figure 12 – SecureChannel and Session Services**

Once a *Client* has established a *Session* it may wish to access the *Session* from a different *SecureChannel*. The Client can do this by validating the new *SecureChannel* with the *ActivateSession Service* described in Clause 5.6.3.

If a *Server* acts as a *Client* to other *Servers*, which is commonly refered to as *Server* chaining, then the Server must be able to maintain user level security. By this we mean that the user identity should be passed to the underlying server or it should be mapped to an appropriate user identity in the underlying server. It is unacceptable to ignore user level security. This is required to ensure that security is maintained and that a user does not obtain information that they should not have access to. For example the server may establish a *SecureChannel* to each *Server* for which it is a *Client*. Further more for each chained *Server* it may establish a *Session*. The *Session* that is established to a chained *Server* may be established using the same user identity, that was used to connect to the orginal *Server*. Another example would be to establish the *Session* using a different user identity, but for each command passed to the underlying server the user Identity must be switched via the *ActivateSession Service* to the user identity of the originating *Servers Client*. Another possibility would be for the highlevel server to map it's *Clients* to distinct underlying *Server* accounts or account groups.

### 5.5.2 OpenSecureChannel

#### 5.5.2.1 Description

This *Service* is used to open or renew a *SecureChannel* that can be used to ensure *Confidentiality* and *Integrity* for *Message* exchange during a *Session*. This *Service* requires the *Communication Stack* to apply the various security algorithms to the *Messages* as they are sent and received. Specific implementations of this *Service* for different *Communication Stacks* are described in [UA Part 6].

Each *SecureChannel* has a globally-unique identifier and is valid for a specific combination of *Client* and *Server* application instances. Each channel contains one or more *SecurityTokens* that identify a set of cryptography keys that are used to encrypt and authenticate *Messages*. *SecurityTokens* also have globally-unique identifiers which are attached to each *Message* secured with the token. This allows an authorized receiver to know how to decrypt and verify the *Message*.

*SecurityTokens* have a finite lifetime negotiated with this *Service*. However, differences between the system clocks on different machines and network latencies mean that valid *Messages* could arrive after the token has expired. To prevent valid *Messages* from being discarded, the applications should do the following:

1. *Clients* should request a new *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 it expires, even if it has been renewed early. This should ensure that *Clients* do not reject *Messages* secured with the new *SecurityToken* that arrive before the *Client* receives the new *SecurityToken*.
3. *Clients* should accept *Messages* secured by an expired *SecurityToken* for up to 25% of the token lifetime. This should ensure that *Messages* sent by the *Server* before the token expired are not rejected because of network delays.

Each *SecureChannel* exists until it is explicitly closed or until the last token has expired and the overlap period has elapsed.

The *OpenSecureChannel* request and response *Messages* must be signed with the sender's *Certificate*. These *Messages* must always be encrypted. If the transport layer does not provide encryption, then these *Messages* must be encrypted with the receiver's *Certificate*.

The *Certificates* used in the *OpenSecureChannel* service should be the application instance Certificates, however, some communication stacks will not allow *Certificates* that are specific to single application. If this is the case, then *Certificates* which are specific to a machine or an individual user may be used instead.

### 5.5.2.2 Parameters

Table 7 defines the parameters for the *Service*.

**Table 7 – OpenSecureChannel Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters. The *sessionId* is always set to 0 in this request.<br>The type *RequestHeader* is defined in Clause 7.19. |
| clientCertificate | ByteString | A *Certificate* that identifies the *Client*.<br>The *OpenSecureChannel* request must be signed with this *Certificate*. |
| requestType | enum<br>SecurityToken RequestType | The type of *SecurityToken* request:<br>An enumeration that must be one of the following:<br>ISSUE_0 creates a new *SecurityToken* for a new *SecureChannel*.<br>RENEW_1 creates a new *SecurityToken* for an existing *SecureChannel*. |
| secureChannelId | Guid | The identifier for the *SecureChannel* that the new token should belong to. This parameter must be null when creating a new *SecureChannel*. |
| securityPolicy | String | The URI for *SecurityPolicy* to use when securing messages sent over the *SecureChannel*.<br>This value identifies an instance of the *SecurityPolicy* described in Clause 7.22.<br>The set of known URIs and the *SecurityPolicies* associated with them are defined in [UA Part 7]. |
| clientNonce | ByteString | A random number that must not be used in any other request. A new *clientNonce* must be generated for each time a *SecureChannel* is renewed.<br>This parameter must have a length equal to the *symmetricKeyLength* specified in the requested *SecurityPolicy*. See Clause 7.22 for *SecurityPolicy* definition. |
| requestedLifetime | Duration | The requested lifetime, in milliseconds, for the new *SecurityToken* (see Clause 7.6 for *Duration* definition). 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* will be rejected by the receiver. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* type definition). The *sessionId* is always set to 0 and the securityHeader is always set to null in this response. |
| serverCertificate | ByteString | A *Certificate* that identifies the *Server*.<br>The *OpenSecureChannel* response must be signed with this *Certificate*. |
| securityToken | ChannelSecurity Token | Describes the new *SecurityToken* issued by the *Server*. |
| channelId | Guid | A globally-unique identifier for the *SecureChannel*. This is the identifier that must be supplied whenever the *SecureChannel* is renewed. |
| tokenId | String | A globally-unique identifier for a single *SecurityToken* within the channel. This is the identifier that must 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 (see Clause 7.6 for *Duration* definition). The UTC expiration time for the token may be calculated by adding the lifetime to the *createdAt* time. |
| serverNonce | ByteString | A random number that must not be used in any other request.<br>This parameter must have a length equal to the *symmetricKeyLength* specified in the requested *SecurityPolicy*. See Clause 7.22 for *SecurityPolicy* definition. A new *serverNonce* must be generated for each time a *SecureChannel* is renewed. |

#### 5.5.2.3 Service results

Table 8 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 8 – OpenSecureChannel Service Result Codes**

| **Symbolic Id** | **Description** |
|---|---|
| Bad_CertificateInvalid | The *Client* certificate is not valid. |
| Bad_CertificateExpired | The *Client* certificate is expired or not yet valid. |
| Bad_CertificateRevoked | The *Client* certificate has been revoked by the certification authority. |
| Bad_CertificateUntrusted | The *Client* certificate is valid; however, the server does not recognize it as a trusted certificate. |
| Bad_RequestTypeInvalid | The security token request type is not valid. |
| Bad_SecurityPolicyRejected | The security policy does not meet the requirements set by the *Server*. |
| Bad_SecureChannelIdInvalid | See Table 156 for the description of this result code. |
| Bad_NonceInvalid | See Table 156 for the description of this result code. |

### 5.5.3 CloseSecureChannel

#### 5.5.3.1 Description

This *Service* is used to terminate a *SecureChannel*.

The request *Messages* must be signed with the same *Certificate* that was used to sign the *OpenSecureChannel* request.

#### 5.5.3.2 Parameters

Table 9 defines the parameters for the *Service*.

**Table 9 – CloseSecureChannel Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters. The *sessionId* is always set to 0 in this request. The type *RequestHeader* is defined in Clause 7.19. |
| secureChannelId | Guid | The identifier for the *SecureChannel* to close. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 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 156.

**Table 10 – CloseSecureChannel Service Result Codes**

| **Symbolic Id** | **Description** |
|---|---|
| Bad_SecureChannelIdInvalid | See Table 156 for the description of this result code. |

## 5.6 Session Service Set

### 5.6.1 Overview

This *Service Set* defines *Services* for an application layer connection establishment in the context of a *Session*.

### 5.6.2 CreateSession

#### 5.6.2.1 Description

This *Service* is used by an OPC UA *Client* to create a *Session* and the *Server* returns a *sessionId* that uniquely identifies the *Session*. The *Client* submits this *sessionId* on all subsequent *Service* requests that it submits on the *Session*.

Before calling this *Service*, the *Client* must create a *SecureChannel* with the *OpenSecureChannel Service* to ensure the *Integrity* of all *Messages* exchanged during a *Session*. This *SecureChannel* has a unique identifier, which the *Server* must associate with the *sessionId*. The *Server* may accept requests with the *sessionId* only if they are associated with the same *SecureChannel* that was used to create the *Session*. The *Client* may associate a new *SecureChannels* with the *Session* by calling the *ActivateSession* method. The *Server* must verify that the *Certificate* the *Client* used to create the new *SecureChannel* is the same as the *Certificate* used to create the original *SecureChannel*.

In most cases, the *SecureChannel* will be managed by the *Communication Stack*, so the *Server* must be able to *Query* the *Communication Stack* for the *secureChannelId* associated with an incoming request. The exact mechanism depends on the implementation. [UA Part 6] describes how this is done with common *Communication Stacks*.

The *Session* created with this *Service* must not be used until the *Client* calls the *ActivateSession Service* and provides its *SoftwareCertificates* and proves possession of it’s application instance *Certificate* and any user identity token that it provided.

The response also contains a list of *SoftwareCertificates* that identify the capabilities of the *Server*. It contains the list of OPC UA *Profiles* supported by the *Server*. OPC UA *Profiles* are defined in [UA Part 7].

Additional *Certificates* issued by other organisations may be included to identify additional *Server* capabilities. Examples of these *Profiles* include support for specific information models and support for access to specific types of devices.

When a *Session* is created, the *Server* adds an entry for the *Client* in its *SessionDiagnosticArray Variable*. See [UA Part 5] for a description of this *Variable*.

*Sessions* are created to be independent of the underlying communications connection. Therefore, if a communications connection fails, the *Session* is not immediately affected. The exact mechanism to recover from an underlying communication connection error depends on the SecureChannel mapping described in [UA Part 6].

*Sessions* are terminated by the *Server* automatically if the *Client* fails to issue a *Service* request on the *Session* within the timeout period negotiated by the *Server* in the *CreateSession Service* response. This protects the *Server* against *Client* failures and against situations where a failed underlying connection cannot be re-established. *Clients* must be prepared to submit requests in a timely manner to prevent the *Session* from closing automatically. *Clients* may explicitly terminate *Sessions* using the *CloseSession Service*.

When a *Session* is terminated, all outstanding requests on the *Session* are aborted and *Bad_SessionClosed StatusCodes* are returned to the *Client*. In addition, the *Server* deletes the entry for the *Client* from its *SessionDiagnosticArray Variable* and notifies any other *Clients* who were subscribed to this entry.

*Subscriptions* assigned to the *Client*, however, are not necessarily terminated when the *Session* is terminated. Each has its own lifetime to protect against data loss if a *Session* terminates abruptly. In these cases, the *Subscription* can be reassigned to another *Client* before its lifetime expires.

Some *Servers*, such as aggregating *Servers*, also act as *Clients* to other *Servers*. These *Servers* typically support more than one system user, acting as their agent to the *Servers* that they represent. Security for these *Servers* is supported at two levels.

First, each UA *Service* request contains a string parameter that is used to carry an audit record id. A *Client*, or any *Server* operating as a *Client*, such as an aggregating *Server*, can create a local audit log entry for a request that it submits. This parameter allows the *Client* to pass the identifier for this entry with the request. If the *Server* also maintains an audit log, it can include this id in the audit log entry that it writes. When the log is examined and the entry is found, the examiner will be able to relate it directly to the audit log entry created by the *Client*. This capability allows for traceability across audit logs within a system. See [UA Part 2] for additional information on auditing. A *Server* that maintains an audit log must provide the audit log entries also via standard event *Messages*. The *Audit EventType* is defined in [UA Part 3].

Second, these aggregating *Servers* may open independent *Sessions* to the underlying *Servers* for each *Client* that accesses data from them. Figure 13 illustrates this concept.

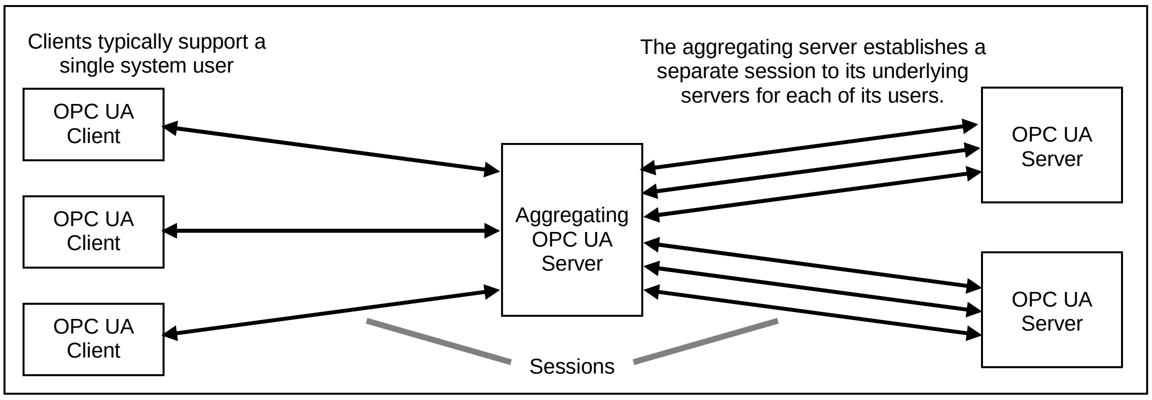
Clients typically support a single system user
The aggregating server establishes a separate session to its underlying servers for each of its users.
OPC UA Client
OPC UA Client
OPC UA Client
Aggregating OPC UA Server
OPC UA Server
OPC UA Server
Sessions

**Figure 13 – Multiplexing Users on a Session**

#### 5.6.2.2 Parameters

Table 11 defines the parameters for the *Service*.

**Table 11 – CreateSession Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters. The *sessionId* is set to 0 in this request. The type *RequestHeader* is defined in Clause 7.19. |
| clientName | String | Displayable string that identifies the OPC *Client* application. 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*, to make diagnostic analysis possible in multi-*Client* environments. A unique name for an instance can be built, for example, by combining the application name, the node name of the system the *Client* is running on and the process Id. |
| clientNonce | ByteString | A random number that should never be used in any other request. This number must have a minimum length of 32 bytes. The *Server* must use this value to prove possession of its application instance *Certificate* in the response. |
| clientCertificate | ByteString | The application instance *Certificate* issued to the *Client*. |
| requestedSession Timeout | 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* will automatically terminate the *Client Session*. |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* type). |
| sessionId | IntegerId | *Server*-unique number that identifies the *Session* (see Clause 7.9 for *IntegerId* definition). This identifier must be assigned so that the *Server* can unambiguously identify the *Session*. The values used must not be reused such that the *Client* or the *Server* has a chance of confusing them with a previous or existing *Session*. 0 is an invalid *sessionId*. |
| revisedSession Timeout | Duration | Actual maximum number of milliseconds that a *Session* will remain open without activity (see Clause 7.6 for *Duration* definition). 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.<br>This number must have a minimum length of 32 bytes.<br>The *Client* must use this value to prove possession of its application instance *Certificate* in the *ActivateSession* request.<br>This value may also be used to prove possession of the *userIdentityToken* it specified in the *ActivateSession* request. |
| serverCertificate | ByteString | The application instance *Certificate* issued to the *Server*.<br>A *Server* must prove possession by using the private key to sign the *Nonce* provided by the *Client* in the request. |
| serverSoftware Certificates [] | SignedSoftware Certificate | These are the *SoftwareCertificates* which have been issued to the *Server* application. Each *SoftwareCertificate* has a signature created by the certification authority that issued it. The *Client* should check this signature to determine if the *SoftwareCertificates* are valid. The *SignedSoftwareCertificate* type is defined in Clause 7.25. This parameter is not specified if the *Server* does not have any *SoftwareCertificates*.<br>*Clients* should call *CloseSession* if they are not satisfied with the *SoftwareCertificates* provided by the *Server*. |
| 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 *serverCertificate* and signing the resulting sequence of bytes.<br>The *SignatureAlgorithm* must be the *asymmetricSignature* algorithm specified in the *SecurityPolicy* for the *Endpoint*.<br>The *SignatureData* type is defined in Clause 7.25. |

#### 5.6.2.3 Service results

Table 12 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 12 – CreateSession Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SecureChannelIdInvalid | See Table 156 for the description of this result code. |
| Bad_NonceInvalid | See Table 156 for the description of this result code. |
| Bad_CertificateInvalid | See Table 156 for the description of this result code. |
| Bad_CertificateExpired | See Table 156 for the description of this result code. |
| Bad_CertificateRevoked | See Table 156 for the description of this result code. |
| Bad_CertificateUntrusted | See Table 156 for the description of this result code. |
| Bad_TooManySessions | The server has reached its maximum number of sessions. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |

### 5.6.3 ActivateSession

#### 5.6.3.1 Description

This *Service* is used by the *Client* to submit its *SoftwareCertificates* to the *Server* for validation and to specify the identity of the user associated with the *Session*. This *Service* request must be issued by the *Client* before it issues any other *Service* request after *CreateSession*. Failure to do so will cause the *Server* to close the *Session*.

Whenever the *Client* calls this *Service* the *Client* must prove that it is the same application that called the *CreateSession Service*. The *Client* does this by creating a signature with the private key associated with the *clientCertificate* specified in the *CreateSession* request. This signature is created by appending the last *serverNonce* provided by the *Server* to the *clientCertificate* and calculating the signature of the resulting sequence of bytes.

Once used, a *serverNonce* cannot be used again. For that reason, the *Server* returns a new *serverNonce* each time the *ActivateSession Service* is called.

When the *ActivateSession Service* is called for the first time then the Server must reject the request if the *SecureChannel* is not same as the one associated with the *CreateSession* request. Subsequent calls to *ActivateSession* may be associated with different *SecureChannels.* If this is the case then the *Server* must verify that the *Certificate* the *Client* used to create the new *SecureChannel* is the same as the *Certificate* used to create the original *SecureChannel*. In addition, the Server must verify that the *Client* supplied a *UserIdentityToken* that is identical to the token currently associated with the *Session*. Once the Server accepts the new *SecureChannel* it must reject requests sent via the old *SecureChannel*.

The *ActivateSession Service* is used to associate a user identity with a *Session*. When a *Client* provides a user identity then it must provide proof that is authorized to use that user identity. The exact mechanism used to provide this proof depends on the type of the *UserIdentityToken*. If the token is a *UserNameIdentityToken* then the proof is the *password* that included in the token. If the token is a X509IdentityToken then the proof is a signature generated with private key associated with the *Certificate*. The data to sign is created by appending the last *serverNonce* to the *clientCertificate* specified in the *CreateSession* request. Other types of tokens use a mechanism that depends on the token. If the token does not include a secret which proves possession of the token then the *Client* must create a signature with secret associated with the token using the same data to sign, that is used to prove possession of the *clientCertificate*.

*Clients* can change the identity of a user associated with a *Session* by calling the *ActivateSession Service*. The *Server* validates the signatures provided with the request and then validates the new user identity. If no errors occur the *Server* replaces the user identity for the *Session*. Changing the user identity for a *Session* may cause discontinuities in active *Subscriptions* because the *Server* may have to tear down connections to underlying system and restablish them using the new credentials.

When a *Client* supplies a list of locale ids in the request, each locale id is required to contain the language component. It may optionally contain the <country/region> component. When the *Server* returns the response, it also may return both the language and the country/region or just the language as its default locale id.

When a *Server* returns a string to the *Client*, it first determines if there are available translations for it. If there are, the *Server* returns the string whose locale id exactly matches the locale id with the highest priority in the *Client*-supplied list.

If there are no exact matches, the *Server* ignores the <country/region> component of the locale id, and returns the string whose <language> component matches the <language> component of the locale id with the highest priority in the *Client* supplied list.

If there still are no matches, the *Server* returns the string that it has along with the locale id.

#### 5.6.3.2 Parameters

Table 13 defines the parameters for the *Service*.

**Table 13 – ActivateSession Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters. The type *RequestHeader* is defined in Clause 7.19. |
| clientSignature | SignatureData | This is a signature generated with the private key associated with the *clientCertificate*.<br>The *SignatureAlgorithm* must be the *asymmetricSignature* algorithm specified in the *SecurityPolicy* for the *Endpoint*.<br>The *SignatureData* type is defined in Clause 7.25. |
| clientSoftwareCertificates [] | SignedSoftware Certificate | These are the *SoftwareCertificates* which have been issued to the *Client* application. Each *SoftwareCertificate* has a signature provided by certification authority that issued it. The *Server* should check this signature to determine if the *SoftwareCertificates* are valid. The *SignedSoftwareCertificate* type is defined in Clause 7.25.<br>This parameter is not specified if the *Client* does not have any *SoftwareCertificates*. *Servers* may reject connections from *Clients* if they are not satisfied with the *SoftwareCertificates* provided by the *Client*.<br>This parameter only needs to be specified during the first call to *ActivateSession* during a single application *Session*. |
| 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* will return the translation with the highest priority that it can. If it does not have a translation for any of the locales identified in this list, then it will return the string value that it has and include the locale id with the string. See [UA Part 3] for more detail on locale ids. If the *Client* fails to specify at least one locale id, the *Server* will use any that it has.<br>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* will 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*.<br>The *UserIdentityToken* is an extensible parameter type defined in Clause 8.7.<br>The EndpointDescription specifies what UserIdentityTokens the Server will accept. |
| userTokenSignature | SignatureData | If the *Client* specified a user identity token that supports digital signatures, then it must create a signature and pass it as this parameter.<br>The *SignatureAlgorithm* depends on the identity token type.<br>The *SignatureData* type is defined in Clause 7.25. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| serverNonce | ByteString | A random number that should never be used in any other request.<br>This number must have a minimum length of 32 bytes.<br>The *Client* must 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 Clause 7.28 for *StatusCode* definition). |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information associated with *SoftwareCertificate* validation errors (see Clause 7.5 for *DiagnosticInfo* definition). |

#### 5.6.3.3 Service results

Table 14 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 14 – ActivateSession Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_IdentityTokenInvalid | See Table 156 for the description of this result code. |
| Bad_IdentityTokenRejected | See Table 156 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_ApplicationSignatureInvalid | The signature provided by the client application is missing or invalid. |
| Bad_UserSignatureInvalid | The user token signature is missing or invalid. |
| Bad_NoValidCertificates | The *Client* did not provide at least one software certificate that is valid and meets the profile requirements for the *Server*. |

### 5.6.4 CloseSession

#### 5.6.4.1 Description

This *Service* is used to terminate a *Session*. The *Server* takes the following actions when it receives a *CloseSession* request:

a) It stops accepting requests for the *Session*. All subsequent requests received for the *Session* are discarded.

b) It returns negative responses with the *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 Clause 7.19 for *RequestHeader* definition). |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 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 156.

**Table 16 – CloseSession Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SessionIdInvalid | See Table 156 for the description of this result code. |

### 5.6.5 Cancel

#### 5.6.5.1 Description

This *Service* is used to cancel an outstanding Service request.

#### 5.6.5.2 Parameters

Table 17 defines the parameters for the *Service*.

**Table 17 – Cancel Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| sequenceNumber | Counter | The sequence number given by the *Client* to the request that should be canceled. The *Client* must assign unique sequence numbers to each service request if he wants to cancel service requests. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |

#### 5.6.5.3 Service results

Table 18 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 18 – Cancel Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SequenceNumberInvalid | The sequence number was invalid or the service response was already sent. |

## 5.7 NodeManagement Service Set

### 5.7.1 Overview

This *Service Set* defines *Services* to add and delete *AddressSpace Nodes* and *References* between them. All added *Nodes* continue to exist in the *AddressSpace* even if the *Client* that created them disconnects from the *Server*.

In the *Services* that follow, many of the *NodeIds* are represented by *ExpandedNodeIds*. *ExpandedNodeIds* identify the namespace by their string name rather than by their *NamespaceTable* index. This allows the *Server* to add the namespace to its *NamespaceTable* if necessary.

### 5.7.2 AddNodes

#### 5.7.2.1 Description

This *Service* is used to add one or more *Nodes* into the *AddressSpace* hierarchy. Using this *Service*, each *Node* is added as the *TargetNode* of a *HierarchicalReference* to ensure that the *AddressSpace* is fully connected and that the *Node* is added as a child within the *AddressSpace* hierarchy (see [UA Part 3]).

### 5.7.2.2 Parameters

Table 19 defines the parameters for the *Service*.

**Table 19 – AddNodes Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 | Expanded *NodeId* of the parent *Node* for the *Reference*. The *ExpandedNodeId* type is defined in Clause 7.7. |
| 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 must be 0.<br>If the *Server* cannot use this *NodeId*, it rejects this *Node* and returns the appropriate error code.<br>If the *Client* does not want to request a *NodeId*, then it sets the value of this parameter to the null expanded *NodeId*.<br>If the *Node* to add is a *ReferenceType Node*, its *NodeId* must be a numeric id. See [UA 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<br>NodeAttributes | The *Attributes* that are specific to the *NodeClass*. The *NodeAttributes* parameter type is an extensible parameter type specified in Clause 8.6.<br>The *Client* must provide a value for each mandatory attribute. If an optional attribute is not set by the *Client*, the *Server* behaviour is vendor specific. |
| typeDefinition | Expanded NodeId | *NodeId* of the *TypeDefinitionNode* for the *Node* to add. This parameter must be null for all *NodeClasses* other than *Object* and *Variable* in which case it must be provided. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 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 Clause 7.28 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 Clause 7.5 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. |

### 5.7.2.3 Service results

Table 20 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 20 – AddNodes Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.7.2.4 StatusCodes

Table 21 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 21 – AddNodes Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_ParentNodeIdInvalid | The parent node id does not to refer to a valid node. |
| Bad_ReferenceTypeIdInvalid | The reference type id does not refer to a valid reference type node. |
| Bad_ReferenceNotAllowed | The reference could not be created because it violates constraints imposed by the data model. |
| Bad_NodeIdRejected | The requested node id was rejected either because it was invalid or because the server does not allow node ids to be specified by the client. |
| Bad_NodeIdExists | The requested node id is already used by another node. |
| Bad_NodeClassInvalid | The node class is not valid. |
| Bad_BrowseNameInvalid | The browse name is invalid. |
| Bad_BrowseNameDuplicated | The browse name is not unique among nodes that share the same relationship with the parent. |
| Bad_NodeAttributesInvalid | The node *Attributes* are not valid for the node class. |
| Bad_TypeDefinitionInvalid | The type definition node id does not reference an appropriate type node. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |

### 5.7.3 AddReferences

#### 5.7.3.1 Description

This *Service* is used to add one or more *References* to one or more *Nodes*. The *NodeClass* is an input parameter that is used to validate that the *Reference* to be added matches the *NodeClass* of the *TargetNode*. This parameter is not validated if the *Reference* refers to a *TargetNode* in a remote *Server*.

In certain cases, adding new *References* to the *AddressSpace* will require that the *Server* add new *Server* ids to the *Server*'s *ServerTable Variable*. For this reason, remote *Servers* are identified by their URI and not by their *ServerTable* index. This allows the *Server* to add the remote *Server* URIs to its *ServerTable*.

### 5.7.3.2 Parameters

Table 22 defines the parameters for the *Service*.

**Table 22 – AddReferences Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| referencesToAdd [] | AddReferences Item | List of *Reference* instances to add to the *SourceNode*. The *targetNodeClass* of each *Reference* in the list must match the *NodeClass* of the *TargetNode*. |
| sourceNodeId | NodeId | *NodeId* of the *Node* to which the *Reference* is to be added. The source *Node* must always exist in the *Server* to add the *Reference*. |
| 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*. |
| targetNodeId | Expanded NodeId | Expanded *NodeId* of the *TargetNode*. The *ExpandedNodeId* type is defined in Clause 7.7. |
| targetNodeClass | NodeClass | *NodeClass* of the *TargetNode*. The *Client* must specify this since the *TargetNode* might not be accessible directly by the *Server*. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *References* to add (see Clause 7.28 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 Clause 7.5 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. |

### 5.7.3.3 Service results

Table 23 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 23 – AddReferences Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

### 5.7.3.4 StatusCodes

Table 24 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 24 – AddReferences Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SourceNodeIdInvalid | The source node id does not refer to a valid node. |
| Bad_ReferenceTypeIdInvalid | The reference type id does not refer to a valid reference type node. |
| Bad_ServerUriInvalid | The *Server* URI is not valid. |
| Bad_TargetNodeIdInvalid | The target node id does not refer to a valid node. |
| Bad_NodeClassInvalid | The node class is not valid. |
| Bad_ReferenceNotAllowed | The reference could not be created because it violates constraints imposed by the data model on this server. |
| Bad_ReferenceLocalOnly | The reference type is not valid for a reference to a remote *Server*. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_DuplicateReferenceNotAllowed | The reference type between the nodes is already defined. |
| Bad_InvalidSelfReference | The server does not allow this type of self reference on this node. |

### 5.7.4 DeleteNodes

#### 5.7.4.1 Description

This *Service* is used to delete one or more *Nodes* from the *AddressSpace*. If the *Node* to be deleted is the owner of another *Node* then this *Node* is deleted too. The ownership is defined by the *ModellingRules* specified in [UA Part 3].

When any of the *Nodes* deleted by an invocation of this *Service* is the *TargetNode* of a *Reference*, then those *References* are left unresolved based on the *deleteTargetReference* parameter.

When any of the *Nodes* deleted by an invocation of this *Service* is contained in a *View*, then the *ViewVersion Property* is updated if this *Property* is supported.

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

#### 5.7.4.2 Parameters

Table 25 defines the parameters for the *Service*.

**Table 25 – DeleteNodes Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| nodesToDelete [] | DeleteNodes Item | List of *Nodes* to delete |
| nodeId | NodeId | *NodeId* of the *Node* to delete. |
| deleteTargetReference | Boolean | A *Boolean* parameter with the following values :<br>TRUE delete *References* in *TargetNodes* that *Reference* the *Node* to delete.<br>FALSE delete only the *References* for which the *Node* to delete is the source. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *Nodes* to delete (see Clause 7.28 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 Clause 7.5 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. |

#### 5.7.4.3 Service results

Table 26 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 26 – DeleteNodes Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.7.4.4 StatusCodes

Table 27 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 27 – DeleteNodes Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_NoDeleteRights | The *Server* will not allow the node to be deleted. |

### 5.7.5 DeleteReferences

#### 5.7.5.1 Description

This *Service* is used to delete one or more *References* of a *Node*.

When any of the *References* deleted by an invocation of this *Service* is contained in a *View*, then the *ViewVersion Property* is updated if this *Property* is supported.

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

#### 5.7.5.2 Parameters

Table 25 defines the parameters for the *Service*.

**Table 28 – DeleteReferences Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 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. |
| serverIndex | Index | Index that identifies the *Server* that contains the *TargetNode*. This *Server* may be the local *Server* or a remote *Server*.<br>This index is the index of that *Server* in the local *Server*'s *Server* table. The index of the local *Server* in the *Server* table is always 0. All remote *Servers* have indexes greater than 0. The *Server* table is contained in the *Server Object* in the *AddressSpace* (see [UA Part 5]).<br>The *Client* may read the *Server* table *Variable* to access the description of the target *Server* |
| targetNodeId | ExpandedNodeId | *NodeId* of the *TargetNode* of the *Reference*.<br>If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *nodeId* must contain the absolute namespace URI. If the *TargetNode* is a local *Node* the *nodeId* must contain the namespace index. |
| deleteBidirectional | Boolean | A *Boolean* parameter with the following values :<br>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.<br>FALSE delete only the specified *Reference*. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *References* to delete (see Clause 7.28 for *StatusCode* definition). The size and order of the list matches the size and order of the *referencesToDelete* request parameter. |
| diagnosticInfos [] | Diagnostic Info | List of diagnostic information for the *References* to delete (see Clause 7.5 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. |

#### 5.7.5.3 Service results

Table 29 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 29 – DeleteReferences Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.7.5.4 StatusCodes

Table 30 defines values for the r*esults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 30 – DeleteReferences Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SourceNodeIdInvalid | The source node id does not refer to a valid node. |
| Bad_ReferenceTypeIdInvalid | The reference type id does not refer to a valid reference type node. |
| Bad_ServerIndexInvalid | The server index is not valid. |
| Bad_TargetNodeIdInvalid | The target node id does not refer to a valid node. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_NoDeleteRights | The server will not allow the reference to be deleted. |

## 5.8 View Service Set

### 5.8.1 Overview

A *View* is a subset of the *AddressSpace* created by the *Server*. Future versions of this specification may also define services to create *Client*-defined *Views*. *Views* are represented in the *AddressSpace* by *View Nodes* that are referenced by the standard *Views Folder* using *Organizes References*. *Clients* can use the browse *Service* to identify the structure of the *View* and the *Nodes* contained in it. See [UA Part 5] for a description of the organisation of views in the *AddressSpace*.

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

### 5.8.2 BrowseProperties

#### 5.8.2.1 Description

This *Service* is used to discover the *Properties* supported by one or more specified *Nodes*. The read *Service* can be used to access the *Property* values.

### 5.8.2.2 Parameters

Table 31 defines the parameters for the *Service*.

**Table 31 – BrowseProperties Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| nodesToAccess [] | NodeId | List of *NodeIds* of the *Nodes* whose *Properties* are to be accessed. |
| properties[] | QualifiedName | List of *Node Properties* to return. If set to null, then all *Properties* are returned. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *the type* definition). |
| results [] | BrowseProperties Result | List of results for the *Nodes* to access. The size and order of the list matches the size and order of the *nodesToAccess* request parameter. |
| statusCode | StatusCode | *StatusCode* for the *Node* to access (see Clause 7.28 for *StatusCode* definition). |
| propertyResults [] | BrowseProperties PropertyResult | List of results for the *Node* to access. The size and order of the list matches the size and order of the *properties* parameter if the requested list was not empty. Otherwise the list contains all *Properties* of the browsed *Node*. |
| propertyName | QualifiedName | The name of the returned *Property*. |
| propertyDisplayName | LocalizedText | The *DisplayName* of the *Property*. |
| propertyNodeId | NodeId | The *NodeId* of the returned *Property*. |
| propertyStatusCode | StatusCode | *StatusCode* for the returned *Property*. |
| diagnosticInfos [] | DiagnosticInfo | A list of diagnostic information for a *Property* being read. This list is omitted if diagnostics information was not requested. If present, this list must have the same length as the list of *Property* results. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the *Nodes* to access. The size and order of the list matches the size and order of the *nodesToAccess* request parameter. This list is empty if diagnostics information was not requested in the request header (see Clause 7.5 for *DiagnosticInfo* definition). |

### 5.8.2.3 Service results

Table 32 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 32 – BrowseProperties Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

### 5.8.2.4 StatusCodes

Table 33 defines values for the operation level *statusCode* and *propertyStatusCode* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 33 – BrowseProperties Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_PropertyNameUnknown | The property name is not known by the server. |

### 5.8.3 Browse

### 5.8.3.1 Description

This *Service* is used to discover the *References* of a specified *Node.* The browse can be further limited by the use of a *View*. This Browse *Service* also supports a primitive filtering capability.

### 5.8.3.2 Parameters

Table 34 defines the parameters for the *Service*.

**Table 34 – Browse Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| view | ViewDescription | Description of the *View* to browse (see Clause 7.31 for *ViewDescription* definition). A null *ViewDescription* value indicates the entire *AddressSpace*. Use of the null *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. |
| nodeToBrowse | NodeId | *NodeId* of the *Node* to be browsed. The passed *nodeToBrowse* must be part of the passed *view*. |
| maxReferences ToReturn | Counter | Indicates the maximum number of references to return. The value 0 indicates that the *Client* is imposing no limitation (see Clause 7.3 for *Counter* definition). |
| browseDirection | enum BrowseDirection | An enumeration that specifies the direction of *References* to follow. It has the following values :<br>FORWARD_0 select only forward *References*.<br>INVERSE_1 select only inverse *References*.<br>BOTH_2 select forward and inverse *References*.<br>The returned *References* do indicate the direction the *Server* followed in the *isForward* parameter of the *ReferenceDescription*. |
| referenceTypeId | NodeId | Specifies the *NodeId* of the *ReferenceType* to follow. Only instances of this *ReferenceType* or its subtypes are returned.<br>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:<br>**Bit** / **NodeClass**:<br>0 Object<br>1 Variable<br>2 Method<br>3 ObjectType<br>4 VariableType<br>5 ReferenceType<br>6 DataType<br>7 View<br>If set to zero, then all *NodeClasses* are returned. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| continuationPoint | ByteString | *Server* defined opaque value that identifies the continuation point.<br>The continuation point is used only when the browse results are too large to be contained in a single response. “Too large” in this context means that the *Server* is not able to return a larger response, or the number of *Refereces* to return exceeds the maximum number of *References* to return that was specified by the *Client* in the request.<br>The continuation point is used in the *BrowseNext Service*. When not used, the value of this parameter is null. If a continuation point is returned the *Client* must call *BrowseNext* to get the next set of browse information or to free the resources for the continuation point in the *Server*.<br>*Servers* must support at least one continuation point per *Session*. *Servers* specify a max continuation points per *Session* in the *ServerCapabilities Object* defined in [UA Part 5].<br>A continuation point will remain active until the *Client* passes the continuation point to *BrowseNext* or the *Session* is closed. If the max continuation points have been reached the least recently used continuation point will be reset. |
| references [] | Reference Description | List of *References* selected for the browsed *Node*. Empty, if no *References* met the browse direction or *Reference* filter criteria. The Reference Description type is defined in Clause 7.17. |

#### 5.8.3.3 Service results

Table 35 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 35 – Browse Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_ViewIdUnknown | The view id does not refer to a valid view *Node*. |
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_ReferenceTypeIdInvalid | The reference type id does not refer to a valid reference type node. |
| Bad_BrowseDirectionInvalid | The browse direction is not valid. |
| Bad_NodeNotInView | The nodeToBrowse is not part of the view. |

### 5.8.4 BrowseNext

#### 5.8.4.1 Description

This *Service* is used to request the next set of *Browse* or *BrowseNext* response information that is too large to be sent in a single response. “Too large” in this context means that the *Server* is not able to return a larger response or that the number of results to return exceeds the maximum number of results to return that was specified by the *Client* in the original Browse request. The *BrowseNext* must be submitted on the same *Session* that was used to submit the Browse or *BrowseNext* that is being continued.

#### 5.8.4.2 Parameters

Table 36 defines the parameters for the *Service*.

**Table 36 – BrowseNext Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| releaseContinuationPoint | Boolean | A *Boolean* parameter with the following values :<br>TRUE passed *continuationPoint* will be reset to free resources for the continuation point in the *Server*.<br>FALSE passed *continuationPoint* will be used to get the next set of browse information.<br>A *Client* must 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* must be called with this parameter set to TRUE. |
| continuationPoint | ByteString | *Server*-defined opaque value that represents the continuation point. The value of the continuation point was returned to the *Client* in a previous *Browse* or *BrowseNext* response. This value is 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. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| revisedContinuationPoint | ByteString | *Server*-defined opaque value that represents the continuation point. It is used only if the information to be returned is too large to be contained in a single response. When not used or when *releaseContinuationPoint* is set, the value of this parameter is null. |
| references [] | Reference Description | List of *References* selected for the browsed *Node*. Empty, if no *References* met the browse direction or *Reference* filter criteria. The Reference Description type is defined in Clause 7.17. When *releaseContinuationPoint* is set this list is empty. |

#### 5.8.4.3 Service results

Table 37 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 37 – BrowseNext Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_ContinuationPointInvalid | See Table 156 for the description of this result code. |

### 5.8.5 TranslateBrowsePathsToNodeIds

#### 5.8.5.1 Description

This *Service* is used to request the *Server* to translate one or more browse paths to *NodeIds*. Each browse path is constructed of a starting *Node* and a *RelativePath*. The specified starting *Node* identifies the *Node* from which the *RelativePath* is based. The *RelativePath* contains a sequence of *BrowseNames*. The final *BrowseName* in the path is either the *BrowseName* for a *ReferenceType* or a *BrowseName* for a *Node.* If the final *BrowseName* refers to a *ReferenceType* then the path matches all targets of that *Reference*. If the final *BrowseName* refers to a *Node* then the path matches all targets with the specified *BrowseName*.

The syntax of the *RelativePath* is described in Clause 7.18.

One purpose of this *Service* is to allow programming against type definitions. Since *BrowseNames* must be unique in the context of type definitions, a *Client* may create a browse path that is valid for a type definition and use this path on instances of the type. For example, an *ObjectType* “Boiler” may have a “HeatSensor” *Variable* as *InstanceDeclaration*. A graphical element programmed against the “Boiler” may need to display the *Value* of the “HeatSensor”. If the graphical element would be called on “Boiler1”, an instance of “Boiler”, it would need to call this *Service* specifying the *NodeId* of “Boiler1” as starting *Node* and the *BrowseName* of the “HeatSensor” as browse path. The *Service* would return the *NodeId* of the “HeatSensor” of “Boiler1” and the graphical element could subscribe to its *Value Attribute*.

If a *Node* has multiple targets with the same *BrowseName*, the *Server* will return a list of *NodeIds*. However, since one purpose of this *Service* is to support programming against type definitions, the *NodeId* of the *Node* based on the type definition of the starting *Node* is returned first in the list. If no *NamespaceId* is provided for a *BrowseName* in the browse path, the *NodeId* might not be unique in the context of the type definition. In that case, all applicable *NodeIds* based on the type definition are returned first in the list. In that case, the *NamespaceId* describes the order of those *Nodes*, the *NodeId* with the smallest *NamespaceId* in its *BrowseName* occurs first in the list.

### 5.8.5.2 Parameters

Table 38 defines the parameters for the *Service*.

**Table 38 – TranslateBrowsePathsToNodeIds Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 | Browse path relative to the starting *Node* identifies the *Node* for which the *NodeId* is being requested. See Clause 7.18 for the definition of *RelativePath*. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | TranslateBrowse PathResult | 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 Clause 7.28 for *StatusCode* definition). |
| matchingNodeIds [] | NodeId | List of *NodeIds* of the *Nodes* which can be accessed via the browse path specified in the request. *Servers* that have hierarchies without duplicate *BrowseNames* will always return an array with one *NodeId* in this parameter. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the list of browse paths (see Clause 7.5 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. |

### 5.8.5.3 Service results

Table 39 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Clause 7.28.

**Table 39 – TranslateBrowsePathsToNodeIds Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

### 5.8.5.4 StatusCodes

Table 40 defines values for the operation level *statusCode* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 40 – TranslateBrowsePathsToNodeIds Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_ReferenceOutOfServer | One of the references to follow in the relative path refers to a node in the address space in another server. |
| Bad_BrowsePathInvalid | The specified browse path is invalid, e.g. no browse name is specified after specifying which reference to follow. |
| Bad_TooManyMatches | The requested operation has too many matches to return.<br>Users should use queries for large result sets. *Servers* should allow at least 10 matches before returning this error code. |
| Bad_QueryTooComplex | The requested operation requires too many resources in the server. |
| Bad_NoMatch | The requested operation has no match to return. |

## 5.9 Query Service Set

### 5.9.1 Overview

This *Service Set* is used to issue a *Query* to a *Server*. UA *Query* is generic in that it provides an underlying storage mechanism independent *Query* capability that can be used to access a wide variety of UA data stores and information management systems. UA *Query* permits a *Client* to access data maintained by a *Server* without any knowledge of the logical schema used for internal storage of the data. Knowledge of the *AddressSpace* is sufficient.

An *OPC UA Application* is expected to use the UA *Query Services* as part of an initialization process or an occasional information synchronization step. For example, UA *Query* would be used for bulk data access of a persistent store to initialise an analysis application with the current state of a system configuration. A *Query* may also be used to initialise or populate data for a report.

A *Query* defines what instances of one or more *TypeDefinitionNodes* in the *AddressSpace* should supply a set of *Attributes*. Results returned by a *Server* are in the form of an array of *QueryDataSets*. The selected *Attribute* values in each *QueryDataSet* come from the definition of the selected *TypeDefinitionNodes or related TypeDefinitionNodes* and appear in results in the same order as the *Attributes* that were passed into the *Query*. *Query* also supports *Node* filtering on the basis of *Attribute* values, as well as relationships between *TypeDefinitionNodes*.

### 5.9.2 Querying Views

A *View* is a subset of the *AddressSpace* available in the *Server*. See [UA Part 5] for a description of the organisation of *Views* in the *AddressSpace*.

For any existing *View*, a *Query* may be used to return a subset of data from the *View*. When an application issues a *Query* against a *View*, only data defined by the *View* is returned. Data not included in the *View* but included in the original *AddressSpace* is not returned.

The *Query Services* supports access to current and historical data. The *Service* supports a *Client* querying a past version of the *AddressSpace.* Clients may specify a *ViewVersion* or a *Timestamp* in a *Query* to access past versions of the *AddressSpace*. UA Query is complementary to Historical Access in that the former is used to *Query* an *AddressSpace* that existed at a time and the latter is used to *Query* for the value of *Attributes* over time. In this way, a *Query* can be used to retrieve a portion of a past *AddressSpace* so that *Attribute* value history may be accessed using Historical Access even if the *Node* is no longer in the current *AddressSpace*.

*Servers* that support *Query* are expect to be able to access the address space that is associated with the local *Server* and any *Views* that are available on the local *Server*. If a *View* or the address space also references a remote *Server*, query may be able to access the address space of remote *Server*, but it is not required. If a *Server* does access a remote *Server* the access must be accomplished using the user identity of the *Client* as described in Clause 5.5.1.

### 5.9.3 QueryFirst

#### 5.9.3.1 Description

This *Service* is used to issue a *Query* request to the *Server*. The complexity of the *Query* can range from very simple to highly sophisticated. The *Query* can simply request data from instances of a *TypeDefinitionNode* or *TypeDefinitionNode* subject to restrictions specified by the filter. On the other hand, the *Query* can request data from instances of related *Node* types by specifying *a RelativePath* from an originating *TypeDefinitionNode.* In the filter, a separate set of paths can be constructed for limiting the instances that supply data. A filtering path can include multiple *RelatedTo* operators to define a multi-hop path between source instances and target instances. For example, one could filter on students that attend a particular school, but return information about students and their families. In this case, the student school relationship is traversed for filtering, but the student family relationship is traversed to select data. For a complete description of *ContentFilter* see Clause 7.2, also see Clause 5.9.5 for additional examples of content filter and queries.

The *Client* provides an array of *NodeTypeDescription* which specify the *NodeId* of a *TypeDefinitionNode* and selects what *Attributes* are to be returned in the response. A client can also provide a set of *RelativePaths* through the type system starting from an originating *TypeDefinitionNode*. Using these paths, the client selects a set of *Attributes* from *Nodes* that are related to instances of the originating *TypeDefinitionNode*. Additionally, the *Client* can request the

*Server* return instances of subtypes of *TypeDefinitionNodes*. If a selected *Attribute* does not exist in a *TypeDefinitionNode* but does exist in a subtype, it is assumed to have a null value in the *TypeDefinitionNode* in question. Therefore, this does not constitute an error condition and a null value is returned for the *Attribute*.

The *Client* can use the filter parameter to limit the result set by restricting *Attributes* and *Properties* to certain values. Another way the *Client* can use a filter to limit the result set is by specifying how instances should be related, using *RelatedTo* operators. In this case, if an instance at the top of the *RelatedTo* path cannot be followed to the bottom of the path via specified hops, no *QueryDataSets* are returned for the starting instance or any of the intermediate instances.

When querying for related instances in the *RelativePath*, the *Client* can optionally ask for *References*. A *Reference* is requested via a RelativePath that only includes a *ReferenceType*. If all *References* are desired then the root *ReferenceType* would be listed. These *Reference* are returned as part of the *QueryDataSets*.

### 5.9.3.2 Parameters

Table 41 defines the request parameters and Table 42 the response parameters for the *QueryFirst Service*.

**Table 41 – QueryFirst Request Parameters**

| Name | Type | Description |
|---|---|---|
| Request | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| view | ViewDescription | Specifies a *View* and temporal context to a *Server* (see Clause 7.31 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[] | DataDescription | 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 Clause 7.18.<br>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 must be a valid *Attribute* Id. The *IntegerId* is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA 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 Clause 7.14.<br>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* will be limited to the *Nodes* matching the criteria defined by the filter. ContentFilter is discussed in Clause 7.2 and Clause 8.4. If an empty filter is provided then the entire address space will be examined and all *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 must not exceed this limit.<br>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 must not exceed this limit.<br>A value of 0 indicates that the *Client* is imposing no limitation.<br>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 42 – QueryFirst Response Parameters**

| Name | Type | Description |
|---|---|---|
| Response | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| queryDataSets [] | QueryDataSet | The array of *QueryDataSet*. This array is empty if no *Nodes* met the *nodeTypes* criteria. In this case the continuationPoint parameter must be empty.<br>The *QueryDataSet* type is defined in Clause 7.15. |
| continuationPoint | ByteString | Server-defined opaque value that identifies the continuation point.<br>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.<br>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* must call *QueryNext* to get the next set of *QueryDataSets* or to free the resources for the continuation point in the *Server*.<br>Servers must support at least one *Query* continuation point per session. Servers specify a max continuation points per session in *Server* capabilities *Object* defined in [UA Part 5].<br>A continuation point will 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 will be reset. |
| results[] | QueryResult | List of results for *QueryFirst*. The size and order of the list matches the size and order of the *NodeTypes* request parameter. |
| statusCode | StatusCode | 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. |
| dataDiagnosticInfos [] | DiagnosticInfo | List of diagnostic information *dataToReturn* (see Clause 7.5 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. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the requested *NodeTypeDescription*. |

### 5.9.3.3 Service results

If the *Query* is invalid or cannot be processed, then *QueryDataSets* are not returned and only a *Service* result and optional *DiagnosticInfo* is returned. Table 43 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 43 – QueryFirst Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_InvalidPath | If the relativePath string is invalid, this result code is returned. |
| Bad_InvalidFilter | If the filter specification is invalid, this result code is returned. |
| Bad_QueryError | If the query cannot be processed for any reason, this result code is returned. |
| Bad_ClientTimeOut | If the query cannot be processed in the time specified by the client or imposed by the server |
| Good_ResultsMayBeIncomplete | The server should have followed a reference to a node in a remote server but did not. The result set may be incomplete. |

### 5.9.4 QueryNext

#### 5.9.4.1 Descriptions

This *Service* is used to request the next set of *QueryFirst* or *QueryNext* response information that is too large to be sent in a single response. “Too large” in this context means that the *Server* is not able to return a larger response or that the number of *QueryDataSets* to return exceeds the maximum number of *QueryDataSets* to return that was specified by the *Client* in the original request. The *QueryNext* must be submitted on the same session that was used to submit the *QueryFirst* or *QueryNext* that is being continued.

#### 5.9.4.2 Parameters

Table 44 defines the parameters for the *Service*.

**Table 44 – QueryNext Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| releaseContinuationPoint | Boolean | A *Boolean* parameter with the following values :<br>TRUE passed *continuationPoint* will be reset to free resources for the continuation point in the *Server*.<br>FALSE passed *continuationPoint* will be used to get the next set of browse information.<br>A *Client* must 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* must be called with this parameter set to TRUE.<br>If the parameter is set to TRUE all array parameters in the response must contain empty arrays. |
| continuationPoint | ByteString | Server defined opaque value that represents the continuation point. The value of the continuation point was returned to the *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. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| queryDataSets [] | QueryDataSet | The array of *QueryDataSets*.<br>The *QueryDataSet* type is defined in Clause 7.15. |
| revisedContinuationPoint | ByteString | Server-defined opaque value that represents the continuation point. It is used only if the information to be returned is too large to be contained in a single response. When not used or when *releaseContinuationPoint* is set, the value of this parameter is null. |

#### 5.9.4.3 Service results

Table 45 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 45 – QueryNext Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_ContinuationPointInvalid | See Table 156 for the description of this result code. |

### 5.9.5 Example for Queries and ContentFilters

#### 5.9.5.1 Overview

As discussed in Clause 7.2 and Clause 8.4, a *ContentFilter* structure defines a collection of elements that make up a filtering criteria. Each element in the collection consists of an operator and an array of operands to be used by the operator.

These query examples illustrate *ContentFilters.* The following of operands are used (for a definition of these operand see Clause 8.4):

- Attribute: Used to refer to a *Node* or an *Attribute* of a *Node*.
- Property: Used to refer to the *Value Attribute* of a *Property* associated with a *Node*.
- Literal: Used to hold a constant.

These examples illustrate the use of a subset of allowed operators (for definition of thee Operators see Clause 7.2):

- Equals (=): Evaluates to TRUE if two operands are equal.
- And: Evaluates to TRUE if both of two content filter sub-trees evaluate to TRUE.
- Or: Evaluates to TRUE if either of two content filter sub-trees evaluate to TRUE.
- RelatedTo: Used to specify the *References* to follow (hop), between two *Nodes*, evaluates to TRUE if the hop exist as specified. The first two operands specify the source and destination node types respectively, the third operand the type of relationship, the last operand specifes the number of hops the relationship should be followed before the destination node type is encountered.

#### 5.9.5.2 Used type model

The following examples use the type model described below:

New Reference types:
“HasChild” derived from HierarchicalReference.
“HasAnimal” derived from HierarchicalReference.
“HasPet” derived from HasAnimal.
“HasFarmAnimal” derived from HasAnimal.
“HasSchedule” derived from HierarchicalReference.

PersonType derived from BaseObjectType adds
HasProperty "LastName"
HasProperty "FirstName"
HasProperty "StreetAddress""
HasProperty "City"
HasProperty "ZipCode"
May have HasChild reference to a node of type PersonType
May have HasAnimal reference to a node of type AnimalType (or a sub type of this *Reference* type)

AnimalType derived from BaseObjectType adds
May have HasSchedule reference to a node of type FeedingScheduleType
HasProperty "Name"

DogType derived from AnimalType adds
HasProperty "NickName"
HasProperty "DogBreed"
HasProperty "License"

CatType derived from AnimalType adds
HasProperty "NickName"
HasProperty "CatBreed"

PigType derived from AnimalType adds
    HasProperty "PigBreed"

ScheduleType derived from BaseObjectType adds
    HasProperty "Period"

FeedingScheduleType derived from ScheduleType adds
    HasProperty "Food"
    HasProperty "Amount"

AreaType derived from *BaseObjectType* is just a simple *Folder* and contains no *Properties*.

This example type system is shown in Figure 14. In this Figure, the standard notation is used for all References to *Object* types, *Properties* and sub-types. Additionally supported *References* are contained in an inner box. The actual references only exist in the instances thus no connections to other *Objects* are shown in the Figure and they may be sub-types of the listed *Reference*.

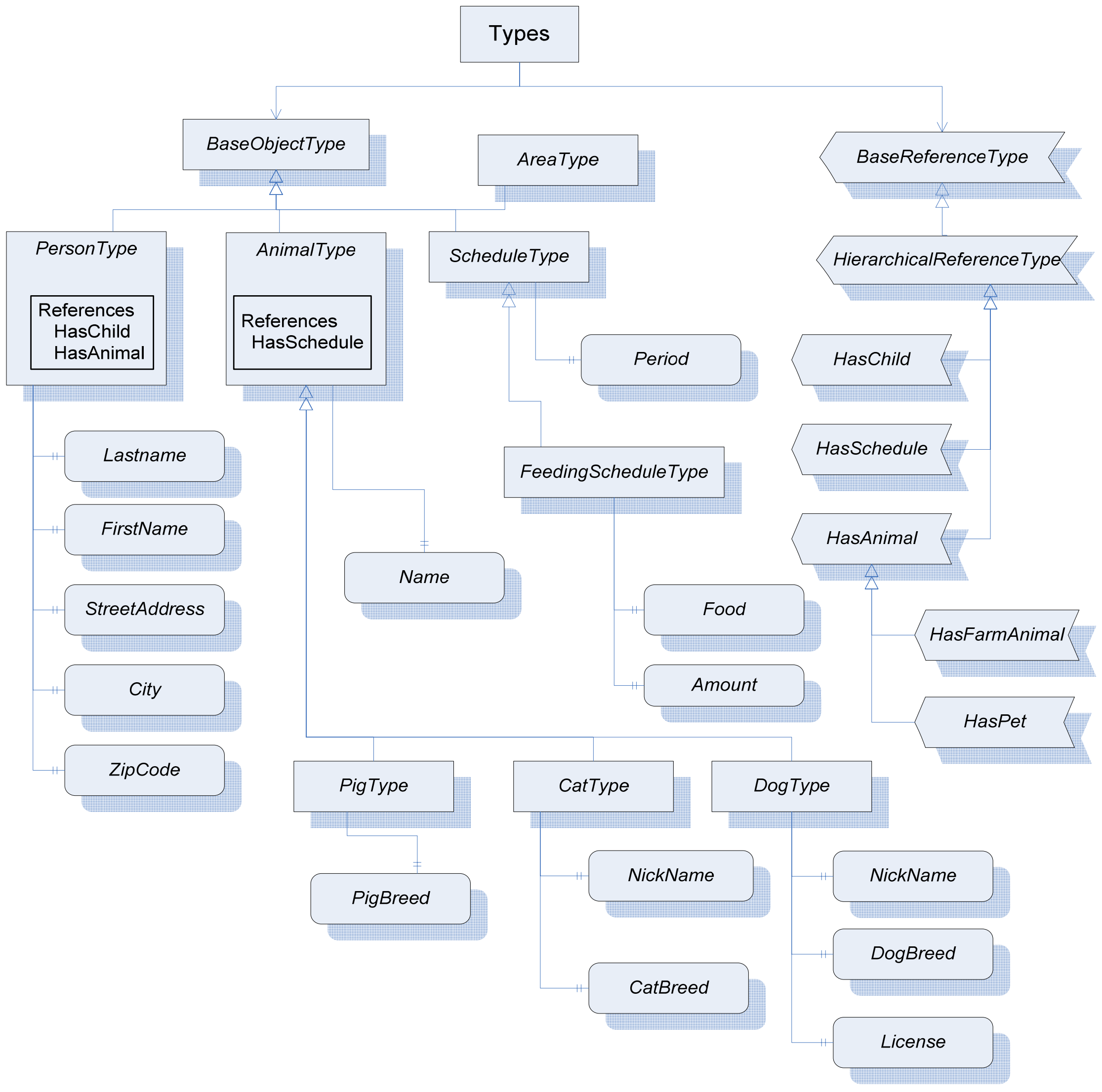

**Figure 14 – Example Type Nodes**

A corresponding example set of instances is shown in Figure 15. These instances include a type *Reference* for *Objects*. Properties also have type *References*, but the *References* are omitted for simplicity. The name of the *Object* is provided in the box and a numeric instance *NodeId* in brackets. Standard *Reference* types use the standard notation, custom *Reference* types are listed with a named *Reference*. For *Properties*, the *BrowseName*, *NodeId*, and *Value* are shown. The *Nodes* that are included in a *View* (View1) are enclosed in the colored box. Two Area nodes are included for grouping of the existing person nodes.

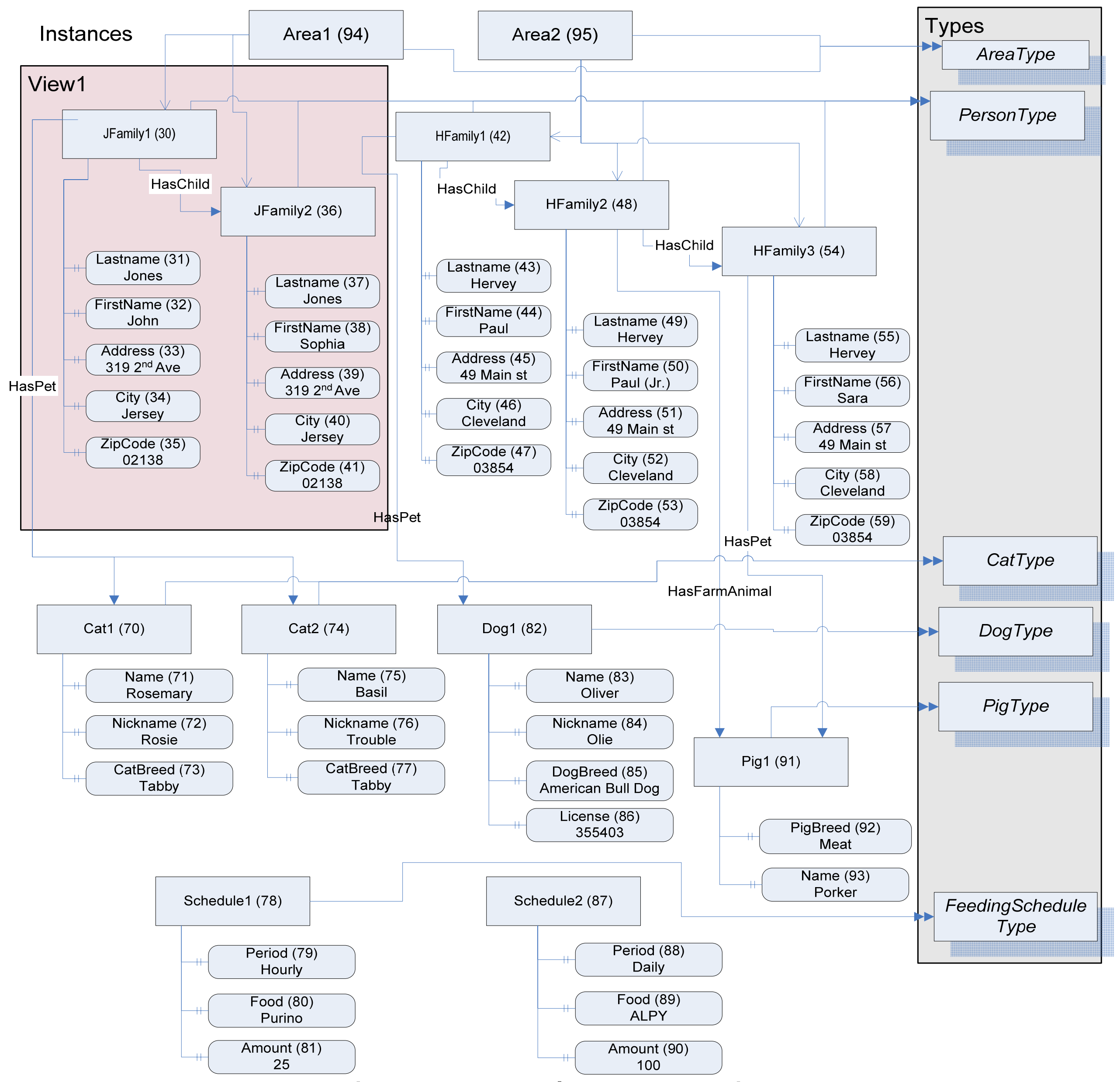

**Figure 15 - Example Instance Nodes**

### 5.9.5.3 Example Notes

For all of the examples in Clause 5.9.5, the type definition *Node* is listed in it's 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, normaly this would not be returned.

### 5.9.5.4 Example 1

This example requests a simple layered filter, a person has a pet and the pet has a schedule.

**Example 1: Get PersonType.lastName, AnimalType.name, ScheduleType.period where the Person Has a Pet and that Pet Has a Schedule.**

The *NodeTypeDescription* parameters used in the example are described in Table 46.

**Table 46 – Example 1 NodeTypeDescription**

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

The corresponding *ContentFilter* is illustrated in Figure 16.

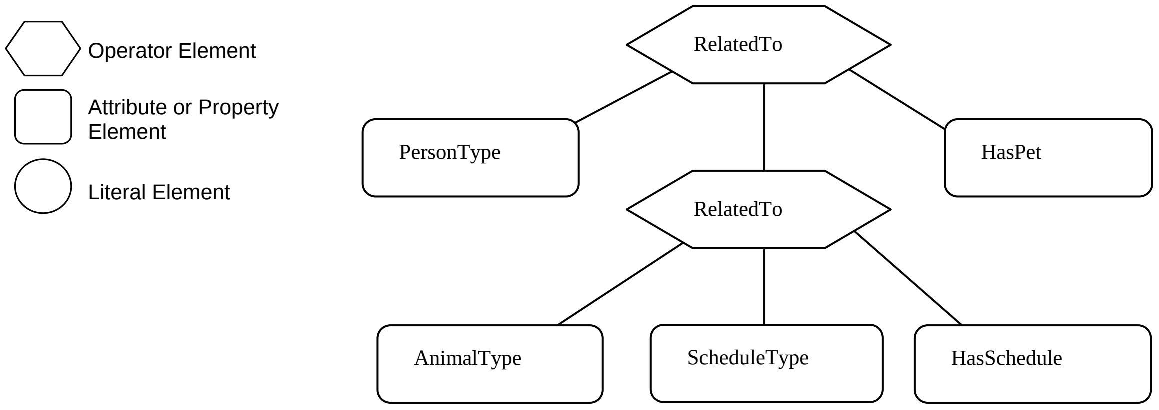

**Figure 16 - Example 1 Filter**

Table 47 describes the *ContentFilter* elements, operators and operands used in the example.

**Table 47 – Example 1 ContentFilter**

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

Table 48 describes the *QueryDataSet* that results from this query if it were executed against the instances described in Figure 15.

**Table 48 – Example 1 QueryDataSets**

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

The Value column is returned as an array for each *Node* description, where the order of the items in the array would correspond to the order of the items that were requested for the given Node Type. In Addition if a single *Attribute* has multiple values then it would be returned as an array within the larger array, for example in this table Rosemary and Basil would be returned in a array the .<hasPet>.AnimalType.name item. They are show as separate rows for ease of viewing.

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The *TypeDefinition NodeId* would be an integer not the symbolic name that is included in the table].

### 5.9.5.5 Example 2

The second example illustrates receiving a list of disjoint *Nodes* and also illustrates that an array of results can be recieved.

**Example 2: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule).**

The NodeTypeDescription parameters used in the example are described in Table 49.

**Table 49 – Example 2 NodeTypeDescription**

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

The corresponding ContentFilter is illustrated in Figure 17.

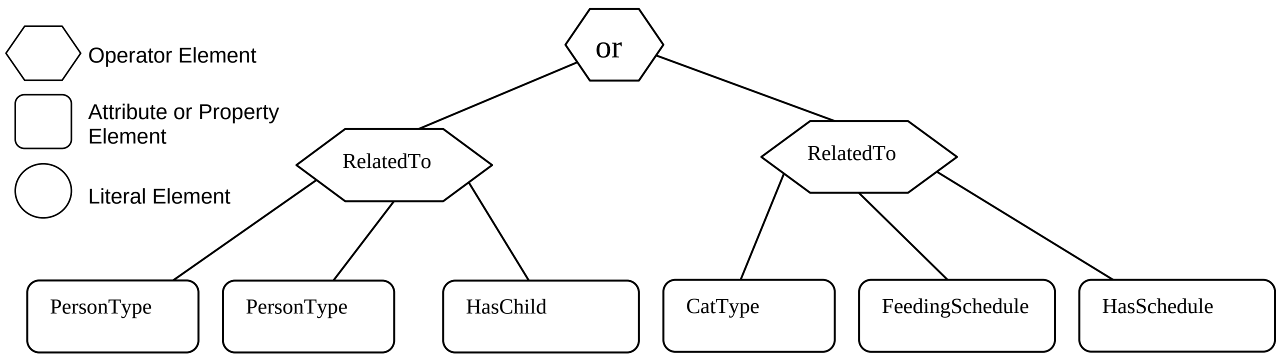

**Figure 17 – Example 2 Filter Logic Tree**

Table 50 describes the elements, operators and operands used in the example. It is worth noting that a Cattype is a subtype of Animaltype.

**Table 50 – Example 2 ContentFilter**

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

The results from this query would contain the *QueryDataSets* shown in Table 51.

**Table 51 – Example 2 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 30 (Jfamily1) | Persontype | .LastName | Jones |
| 42 (HFamily1) | PersonType | .LastName | Hervey |
| 48 (HFamily2) | PersonType | .LastName | Hervey |
| 70 (Cat1) | CatType | .name | Rosemary |
| 74 (Cat2) | CatType | .name | Basil |

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table].

### 5.9.5.6 Example 3

The third example provides a more complex *Query* in which the results are filtered on multiple criteria.

**Example 3: Get PersonType.lastName, AnimalType.name, ScheduleType.period where a person has a pet and the animal has a feeding schedule and the person has a zipcode = '02138' and the schedule.period is daily or hourly.**

Table 52 describes the NodeTypeDescription parameters used in the example.

**Table 52 – Example 3 - NodeTypeDescriptions**

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

The corresponding *ContentFilter* is illustrated in Figure 18.

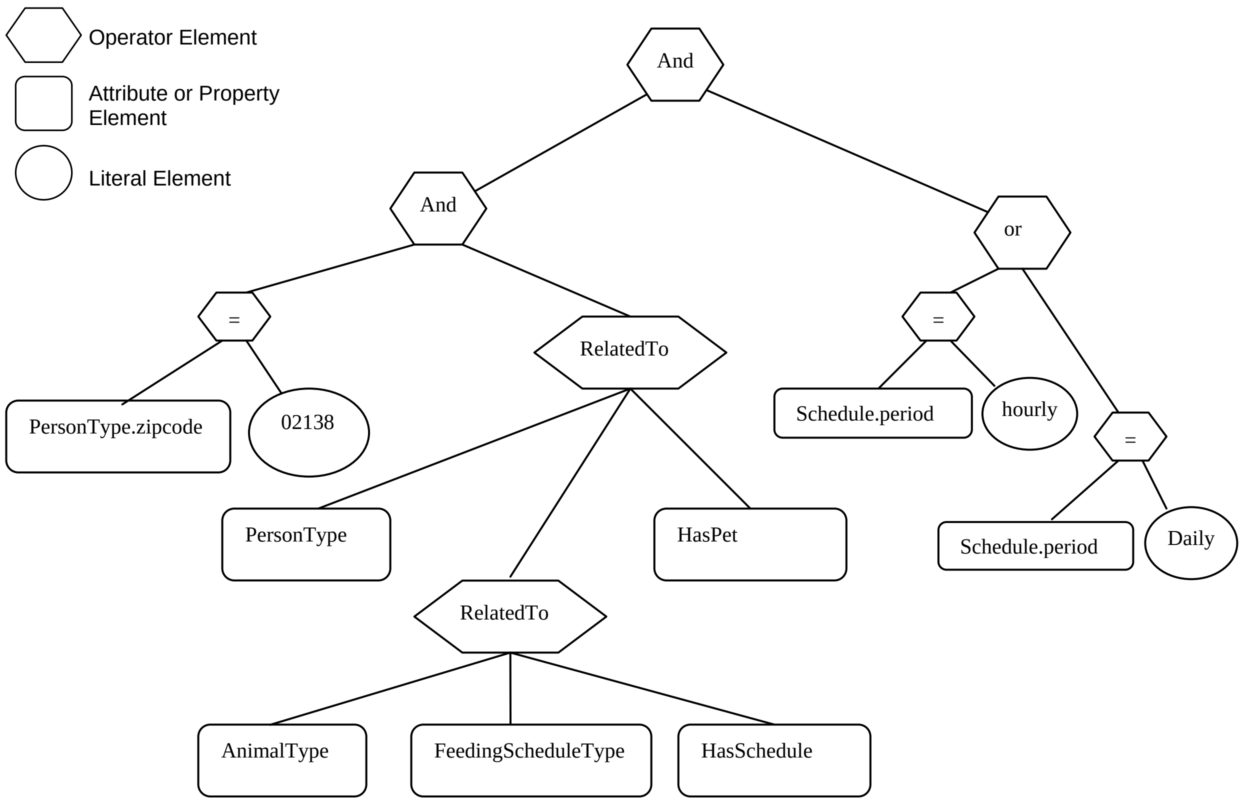

**Figure 18 – Example 3 Filter Logic Tree**

Table 53 describes the elements, operators and operands used in the example.

**Table 53 – Example 3 ContentFilter**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | And | Element Operand= 1 | ElementOperand = 2 | | |
| 1 | And | ElementOperand = 3 | ElementOperand = 5 | | |
| 2 | Or | ElementOperand = 6 | ElementOperand = 7 | | |
| 3 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | ElementOperand = 4 | AttributeOperand = NodeId: HasPet, AttributeId: NodeId | LiteralOperand = ‘1’ |
| 4 | RelatedTo | AttributeOperand = Node: AnilmalType, AttributeId: NodeId | AttributeOperand = NodeId: FeedingScheduleType, AttributeId: NodeId | AttributeOperand = NodeId: HasSchedule, AttributeId: NodeId | LiteralOperand = ‘1’ |
| 5 | Equals | PropertyOperand = NodeId: PersonType Property: zipcode | LiteralOperand = ‘02138’ | | |
| 6 | Equals | PropertyOperand = NodeId: ScheduleType Property: Period | LiteralOperand = ‘Daily’ | | |
| 7 | Equals | PropertyOperand = NodeId: ScheduleType Property: Period | LiteralOperand = ‘Hourly’ | | |

The results from this query would contain the *QueryDataSets* shown in Table 54.

**Table 54 – Example 3 QueryDataSets**

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

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table].

### 5.9.5.7 Example 4

The fourth example provides an illustration of the Hop parameter that is part of the RelatedTo. Operator.

**Example 4: Get PersonType.lastName where a person has a child who has a child who has a pet.**

Table 55 describes the NodeTypeDescription parameters used in the example.

**Table 55 – Example 4 NodeTypeDescription**

| Type Definition Node | Include Subtypes | Relative Path | Attribute Id | Index Range |
|---|---|---|---|---|
| PersonType | FALSE | “.lastName” | value | N/A |

The corresponding *ContentFilter* is illustrated in Figure 19.

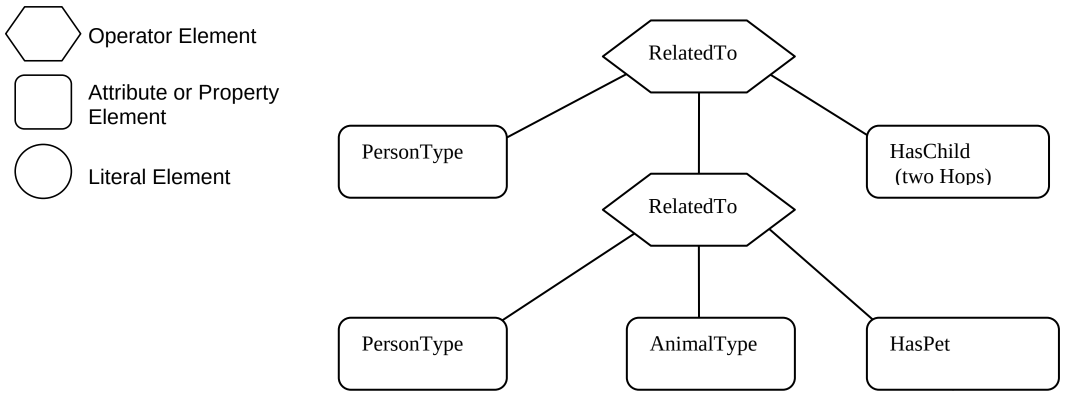

**Figure 19 – Example 4 Filter Logic Tree**

Table 56 describes the elements, operators and operands used in the example.

**Table 56 – Example 4 ContentFilter**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | Element Operand = 1 | AttributeOperand = NodeId: HasChild, AttributeId: NodeId | LiteralOperand = '2' |
| 1 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | AttributeOperand = NodeId: AnimalType, AttributeId: NodeId | AttributeOperand = NodeId: HasPet, AttributeId: NodeId | LiteralOperand = '1' |

The results from this query would contain the *QueryDataSets* shown in Table 57. It is worth noting that the pig "Pig1" is referenced as a pet by Sara, but is referenced as a farmanimal by Sara's parent Paul.

**Table 57 – Example 4 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 42 (HFamily1) | PersonType | ".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].

#### 5.9.5.8 Example 5

The fifth example provides an illustration of the use of alias.

**Example 5: Get the last names of children that have the same first name as a parent of theirs**

Table 58 describes the NodeTypeDescription parameters used in the example.

**Table 58 – Example 5 NodeTypeDescription**

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

The corresponding *ContentFilter* is illustrated in Figure 20.

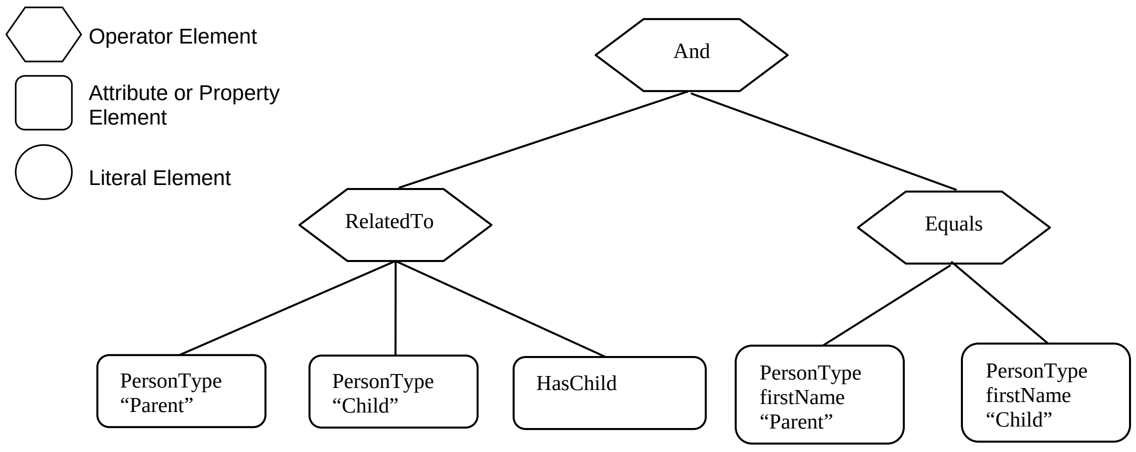

**Figure 20 – Example 5 Filter Logic Tree**

In this example, one *Reference* to PersonType is aliased to "Parent" and another *Reference* to PersonType is aliased to "Child". The value of Parent.firstName and Child.firstName are then compared. Table 59 describes the elements, operators and operands used in the example.

**Table 59 – Example 5 ContentFilter**

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

The results from this query would contain the *QueryDataSets* shown in Table 60.

**Table 60 – Example 5 QueryDataSets**

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

### 5.9.5.9 Example 6

The sixth example provides an illustration a different type of request, one in which the *Client* is interested in displaying part of the address space of the server. This request includes listing a *Reference* as something that is to be returned.

**Example 6: Get PersonType.NodeId, AnimalType.NodeId, PersonType.HasChild Reference, PersonType.HasAnimal Reference where a person has a child who has a Animal.**

Table 61 describes the NodeTypeDescription parameters used in the example.

**Table 61 – Example 6 NodeTypeDescription**

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

The corresponding *ContentFilter* is illustrated in Figure 21.

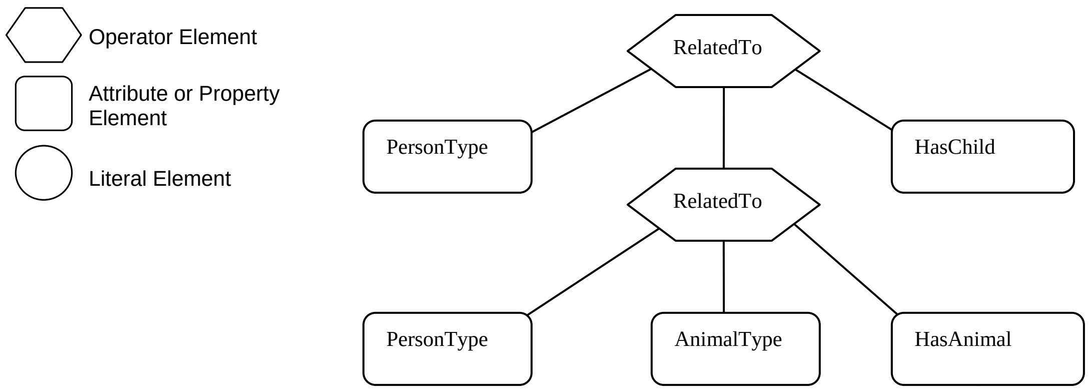

**Figure 21 – Example 6 Filter Logic Tree**

Table 62 describes the elements, operators and operands used in the example.

**Table 62 – Example 6 ContentFilter**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | ElementOperand = 1 | AttributeOperand = Node:HasChild, Attr:NodeId | LiteralOperand = '1' |
| 1 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | AttributeOperand = NodeId: AnimalType, AttributeId: NodeId | AttributeOperand = NodeId: HasAnimal, AttributeId: NodeId | LiteralOperand = '1' |

The results from this query would contain the *QueryDataSets* shown in Table 63.

**Table 63 – Example 6 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 42 (HFamily1) | PersonType | ".NodeId" | 42 (HFamily1) |
| | | <HasChild>PersonType<HasAnimal>AnimalType.NodeId | 91 (Pig1) |
| | | <HasChild> | HasChild *ReferenceDescription* |
| | | <HasChild>PersonType<HasAnimal> | HasFarmAnimal *ReferenceDescription* |
| 48 (HFamily2) | PersonType | ".NodeId" | 48 (HFamily2) |
| | | <HasChild>PersonType<HasAnimal>AnimalType.NodeId | 91 (Pig1) |
| | | <HasChild> | HasChild *ReferenceDescription* |
| | | <HasChild>PersonType<HasAnimal> | HasPet *ReferenceDescription* |

[Note: that the relative path and browse name (in parantheses) 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 63, the value would be a *ReferenceDescription* which are described in Clause 7.17].

Table 64 provides an example of the same QueryDataSet as shown in Table 63 without any additional fields and minimal symbolic Ids. There is an entry for each requested *Attribute*, in the cases where an *Attribute* would return multiple entries the entries are separated by 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 15.

**Table 64 – Example 6 QueryDataSets without Additional Information**

| NodeId | TypeDefinition NodeId | Value |
|---|---|---|
| 42 | PersonType | 42 |
| | | 91 |
| | | [HasChild,TRUE,[48,HFamily2,HFamily2,PersonType]], |
| | | [HasFarmAnimal,TRUE[91,Pig1,Pig1,PigType] |
| 48 | PersonType | 54 |
| | | 91 |
| | | [HasChild,TRUE,[ 54,HFamily3,HFamily3,PersonType]] |
| | | [HasPet, TRUE,[ 91,Pig1,Pig1,PigType]] |

The PersonType, HasChild, PigType, HasPet, HasFarmAnimal identifiers used in the above table would be translated to actual *ExtendedNodeIds*.

### 5.9.5.10 Example 7

The seventh example provides an illustration a request in which a *Client* wants to display part of the address space based on a starting point that was obtained via standard browsing. This request includes listing *References* as something that is to be returned. In this case the Person Browsed to Area2 and wanted to *Query* for information below this starting point.

**Example 7: Get PersonType.NodeId, AnimalType.NodeId, PersonType.HasChild Reference, PersonType.HasAnimal Reference where the person is in Area2 (Cleveland nodes) and the person has a child.**

Table 65 describes the NodeTypeDescription parameters used in the example.

**Table 65 – Example 7 NodeTypeDescription**

| Type Definition Node | Include Subtypes | Relative Path | Attribute Id | Index Range |
|---|---|---|---|---|
| PersonType | FALSE | “.NodeId” | value | N/A |
| | | <HasChild> | value | N/A |
| | | <HasAnimal>NodeId | value | N/A |
| | | <HasAnimal> | value | N/A |

The corresponding *ContentFilter* is illustrated in Figure 22. 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.

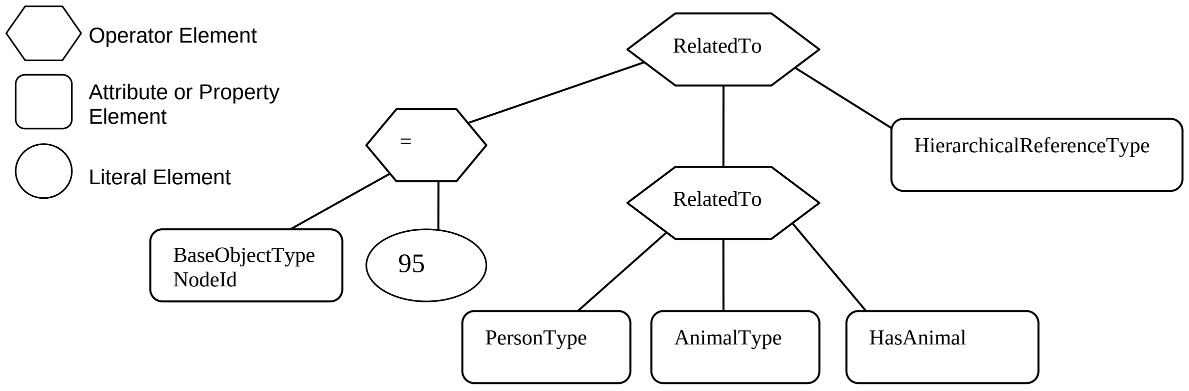

**Figure 22 – Example 7 Filter Logic Tree**

Table 66 describes the elements, operators and operands used in the example.

**Table 66 – Example 7 ContentFilter**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | RelatedTo | AttributeOperand = NodeId: BaseObjectType, AttributeId: NodeId | ElementOperand = 1 | AttributeOperand = Node:HierachicalReference, Attr:NodeId | LiteralOperand = '1' |
| 1 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | AttributeOperand = NodeId: PersonTypreType, AttributeId: NodeId | AttributeOperand = NodeId: HasChild, AttributeId: NodeId | LiteralOperand = '1' |
| 2 | Equals | PropertyOperand = NodeId: BaseObjectType, Property: NodeId, | LiteralOperand = '95 | | |

The results from this *Query* would contain the *QueryDataSets* shown in Table 67.

**Table 67 – Example 7 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 42 (HFamily1) | PersonType | ".NodeId" | 42 (HFamily1) |
| | | <HasChild> | HasChild *ReferenceDescription* |
| | | <HasAnimal>AnimalType.NodeId | NULL |
| | | <HasAnimal> | HasFarmAnimal *ReferenceDescription* |
| 48 (HFamily2) | PersonType | ".NodeId" | 48 (HFamily2) |
| | | <HasChild> | HasChild *ReferenceDescription* |
| | | <HasAnimal>AnimalType.NodeId | 91 (Pig1) |
| | | <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 67, the value would be a *ReferenceDescription* which are described in Clause 7.17].

### 5.9.5.11 Example 8

The eighth example provides an illustration of a request in which the address space is restricted by a *Server* defined *View*. This request is the same as in the second example which illustrates receiving a list of disjoint *Nodes* and also illustrates that an array of results can be received. It is **important** to note that all of the parameters and the *contentFilter* are the same, only the View description would be specified as "View1"

**Example 8: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule) limited by the address space in View1.**

The NodeTypeDescription parameters used in the example are described in Table 68

**Table 68 – Example 8 NodeTypeDescription**

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

The corresponding ContentFilter is illustrated in Figure 23.

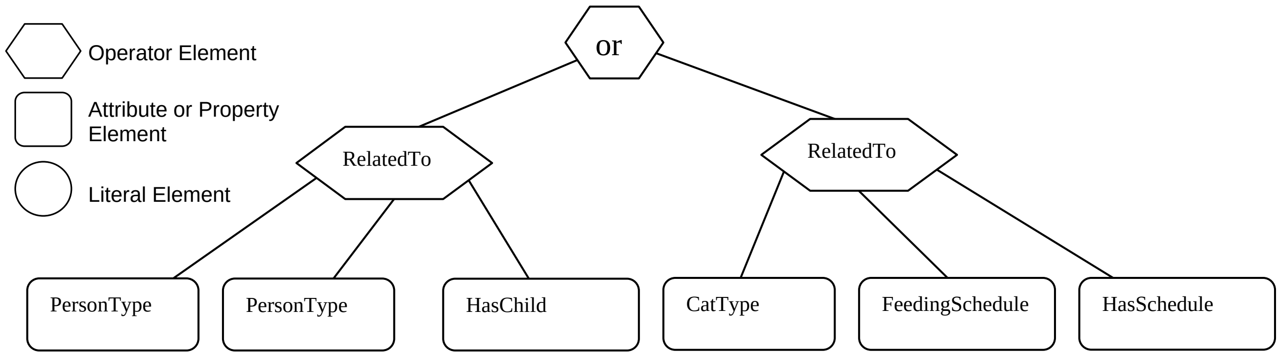

**Figure 23 – Example 8 Filter Logic Tree**

Table 69 describes the elements, operators and operands used in the example. It is worth noting that a CatType is a subtype of AnimalType.

**Table 69 – Example 8 ContentFilter**

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

The results from this query would contain the *QueryDataSets* shown in Table 70. If this is compared to the result set from example 2, the only difference is the omission of the Cat *Nodes*. These *Nodes* are not in the *View* and thus are not include in the result set

**Table 70 – Example 8 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 30 (Jfamily1) | Persontype | .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].

### 5.9.5.12 Example 9

The ninth example provides a further illustration for a request in which the address space is restricted by a *Server* defined *View*. This request is similar to the second example except that some of the requested nodes are expressed in terms of a relative path. It is **important** to note that the *contentFilter* is the same, only the View description would be specified as “View1”.

**Example 9: Get PersonType.lastName, AnimalType.name where a person has a child or (a pet is of type cat and has a feeding schedule) limited by the address space in View1.**

Table 71 describes the NodeTypeDescription parameters used in the example.

**Table 71 – Example 9 NodeTypeDescription**

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

The corresponding ContentFilter is illustrated in Figure 24.

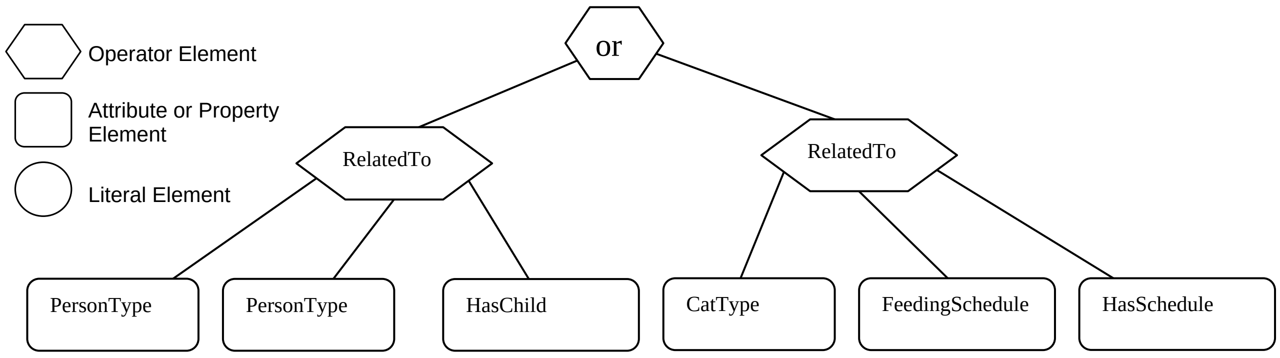

**Figure 24 – Example 9 Filter Logic Tree**

Table 72 describes the elements, operators and operands used in the example.

**Table 72 – Example 9 ContentFilter**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | Or | ElementOperand=1 | ElementOperand = 2 | | |
| 1 | RelatedTo | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | AttributeOperand = NodeId: PersonType, AttributeId: NodeId | AttributeOperand = NodeId: HasChild, AttributeId: NodeId | LiteralOperand = ‘1’ |
| 2 | RelatedTo | AttributeOperand = NodeId: CatType, AttributeId: NodeId | AttributeOperand = NodeId: FeedingScheduleType, AttributeId: NodeId | AttributeOperand = NodeId: HasSchedule, AttributeId: NodeId | LiteralOperand = ‘1’ |

The results from this *Query* would contain the *QueryDataSets* shown in Table 73. If this is compared to the result set from example 2, the Pet *Nodes* are included in the list, even though they are outside of the *View*. This is possible since the name referenced via the relative path and the root *Node* is in the *View*.

**Table 73 – Example 9 QueryDataSets**

| NodeId | TypeDefinition NodeId | RelativePath | Value |
|---|---|---|---|
| 30 (Jfamily1) | Persontype | .LastName | Jones |
| | | <HasAnimal>AnimalType.Name | Rosemary |
| | | <HasAnimal>AnimalType.Name | Basil |

[Note: that the relative path column and browse name (in parentheses in the *NodeId* column) are not in the QueryDataSet and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table].

## 5.10 Attribute Service Set

### 5.10.1 Overview

This *Service Set* provides *Services* to access *Attributes* that are part of *Nodes*.

### 5.10.2 Read

#### 5.10.2.1 Description

This *Service* is used to read one or more *Attributes* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this *Service* allows *Clients* to read the entire set of indexed values as a composite, to read individual elements or to read ranges of elements of the composite.

The maxAge parameter is used to direct the *Server* to access the value from the underlying data source, such as a device, if its copy of the data is older than the maxAge specifies. If the *Server* cannot meet the requested max age, it returns its "best effort" value rather than rejecting the request.

### 5.10.2.2 Parameters

Table 74 defines the parameters for the *Service*.

**Table 74 – Read Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| maxAge | Duration | Maximum age of the value to be read in milliseconds (see Clause 7.6 for *Duration* definition). The age of the value is based on the *Server* timestamp 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.<br>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.<br>If the *Server* does not have a value that is within the maximum age, it must attempt to read a new value from the data source.<br>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.<br>If *maxAge* is set to 0, the *Server* must attempt to read a new value from the data source.<br>If *maxAge* is set to the max Int32 value, the *Server* must attempt to get a cached value if a *MonitoredItem* is defined for the *Attribute*.<br>Negative values are invalid for *maxAge*. |
| timestampsTo Return | enum TimestampsTo Return | An enumeration that specifies the *Timestamp Attributes* to be returned for each requested *Variable Value Attribute*. The *TimestampsToReturn* enumeration is defined in Clause 7.29. |
| 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 Clause 7.16. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | DataValue | List of *Attribute* values (see Clause 7.4 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 Clause 7.5 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. |

### 5.10.2.3 Service results

Table 75 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 75 – Read Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_MaxAgeInvalid | The max age parameter is invalid. |
| Bad_TimestampsToReturnInvalid | See Table 156 for the description of this result code. |

### 5.10.2.4 StatusCodes

Table 76 defines values for the operation level *statusCode* contained in the *DataValue* structure of each *values* element. Common *StatusCodes* are defined in Table 157.

**Table 76 – Read Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_AttributeIdInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeNoData | See Table 157 for the description of this result code. |
| Bad_DataEncodingInvalid | See Table 157 for the description of this result code. |
| Bad_DataEncodingUnsupported | See Table 157 for the description of this result code. |
| Bad_NoReadRights | See Table 157 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |

### 5.10.3 HistoryRead

#### 5.10.3.1 Description

This *Service* is used to read historical values or *Events* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this *Service* allows *Clients* to read the entire set of indexed values as a composite, to read individual elements or to read ranges of elements of the composite. *Servers* may make historical values available to *Clients* using this *Service*, although the historical values themselves are not visible in the *AddressSpace*.

The *AccessLevel Attribute* defined in [UA Part 3] indicates a *Node*'s support for historical values. Several request parameters indicate how the *Server* is to access values from the underlying history data source. The *EventNotifier Attribute* defined in [UA Part 3] indicates a *Node*'s support for historical *Events*.

The *continuationPoint* parameter in the *HistoryRead* is used to mark a point from which to continue the read if not all values could be returned in one response. The value is opaque for the *Client* and is only used to maintain the state information for the *Server* to continue from. A *Server* may use the timestamp of the last returned data item if the timestamp is unique. This can reduce the need in the *Server* to store state information for the continuation point.

For additional details on reading historical data and historical *Events* see [UA Part 11].

#### 5.10.3.2 Parameters

Table 77 defines the parameters for the *Service*.

**Table 77 – HistoryRead ServiceParameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 [UA Part 11]. The *ExtensibleParameter* type is defined in Clause 8.2. |
| timestampsToReturn | enum TimestampsTo Return | An enumeration that specifies the timestamp *Attributes* to be returned for each requested *Variable Value Attribute*. The *TimestampsToReturn* enumeration is defined in Clause 7.29.<br>Specifying a *TimestampsToReturn* of NEITHER is not valid. A *Server* must return a *Bad_InvalidTimestampArgument StatusCode* in this case. If the requested timestamp is not stored for a *Node*, the operation for the *Node* must return the *StatusCode Bad_NoTimestamp*. When reading *Events* this only applies to *Event* fields that are of type *DataValue*. |
| releaseContinuation Points | Boolean | A *Boolean* parameter with the following values :<br>TRUE passed *continuationPoints* will be reset to free resources for the continuation point in the *Server*.<br>FALSE passed *continuationPoints* will be used to get the next set of history information.<br>A *Client* must 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 history information, *HistoryRead* must be called with this parameter set to TRUE. |
| nodesToRead [] | HistoryReadValueId | This parameter contains the list of items upon which the historical retrieval is to be performed. |
| nodeId | NodeId | If the *parameterTypeId of HistoryReadDetails* has the value RAW, PROCESSED, MODIFIED or ATTIME:<br>The *nodeId* of the *Nodes* whose historical values are to be read. The value returned must always include a timestamp.<br>If the *parameterTypeId of HistoryReadDetails* has the value EVENTS:<br>The *NodeId* of the *Node* whose *Event* history is to be read.<br>If the *Node* does not support the requested access for historical values or historical *Events* the appropriate error response for the given *Node* will be generated. |
| dataEncoding | QualifiedName | A *QualifiedName* that specifies the data encoding to be returned for the *Node* to be read (see Clause 7.16 for definition how to specify the data encoding). |
| 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.<br>The *HistoryRead* is used to select an ordered sequence of historical values. A continuation point marks a point in that ordered sequence, such that the *Server* returns the subset of the sequence that follows that point.<br>A null value indicates that this parameter is not used.<br>This continuation point is described in more detail in [UA Part 11]. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 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 Clause 7.28 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. In this case the *StatusCode* of the read result is set to *Good_MoreData*.<br>When this parameter is not used, its value is null.<br>*Servers* must support at least one continuation point per *Session*. *Servers* specify a max continuation points per *Session* in the *Server* capabilities *Object* defined in [UA Part 5]. A continuation point will 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 will be reset. |
| historyData | Extensible Parameter HistoryData | The history data returned for the *Node*.<br>The *HistoryData* parameter type is an extensible parameter type formally defined in [UA Part 11]. It specifies the types of history data that can be returned. The *ExtensibleParameter* base type is defined in Clause 8.2. |
| diagnosticInfos [] | Diagnostic Info | List of diagnostic information. The size and order of the list matches the size and order of the *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. |

#### 5.10.3.3 Service results

Table 78 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 78 – HistoryRead Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_TimestampsToReturnInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |

#### 5.10.3.4 StatusCodes

Table 79 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 79 – HistoryRead Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Good_MoreData | More data is available in the time range beyond the number of values requested.<br>This is an indication to reissue the read request using the continuationPoint parameter. |
| Good_NoData | No data was found in the specified time range. |
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_DataEncodingInvalid | See Table 157 for the description of this result code. |
| Bad_DataEncodingUnsupported | See Table 157 for the description of this result code. |
| Bad_NoTimestamp | The requested timestamp is not available for the *Node*. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_ContinuationPointInvalid | See Table 156 for the description of this result code. |

### 5.10.4 Write

#### 5.10.4.1 Description

This *Service* is used to write values to one or more *Attributes* of one or more *Nodes*. For constructed *Attribute* values whose elements are indexed, such as an array, this *Service* allows *Clients* to write the entire set of indexed values as a composite, to write individual elements or to write ranges of elements of the composite.

The sequence number in the *Service* request header is used by this *Service* to detect duplicate requests. *Servers* are responsible for tracking the sequence numbers used by the *Service* and for discarding requests that contain a sequence number that has already been used. *Servers* are expected to manage the rollover appropriately.

The values are written to the data source, such as a device, and the *Service* does not return until it writes the values or determines that the value cannot be written. In certain cases, the *Server* will successfully write to an intermediate system or *Server*, and will not know if the data source was updated properly. In these cases, the *Server* will report a success code that indicates that the write was not verified. In the cases where the *Server* is able to verify that it has successfully written to the data source, it reports an unconditional success.

It is possible that the *Server* may successfully write some *Attributes*, but not others. Rollback is the responsibility of the *Client*.

### 5.10.4.2 Parameters

Table 80 defines the parameters for the *Service*.

**Table 80 – Write Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 must be a valid *Attribute* id. The *IntegerId* is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA Part 6]. |
| indexRange | NumericRange | This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. The first element is identified by index 0 (zero). The *NumericRange* type is defined in Clause 7.14.<br>This parameter is not used 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 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 Clause 7.4 for *DataValue* definition).<br>If the *nodeAttributeId* parameter specifies a structure, an array or a range of array elements, then this parameter contains a composite value.<br>If a null timestamp is supplied for the source timestamp or the *Server* timestamp with the value, the *Server* replaces these nulls with the time that the request was received.<br>A Server must reject values that are not of the same type as the *Attribute*'s value type. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of results for the *Nodes* to write (see Clause 7.28 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 Clause 7.5 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. |

### 5.10.4.3 Service results

Table 81 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 81 – Write Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.10.4.4 StatusCodes

Table 82 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 82 – Write Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Good_CompletesAsynchronously | See Table 156 for the description of this result code.<br>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 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_AttributeIdInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeNoData | See Table 157 for the description of this result code. |
| Bad_WriteNotSupported | The requested write operation is not supported.<br>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* will not perform any write on this *Node* and will return this *StatusCode* for this *Node*. |
| Bad_NoWriteRights | See Table 157 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code.<br>The current user does not have permission to write the attribute. |
| Bad_OutOfRange | See Table 157 for the description of this result code. |
| Bad_TypeMismatch | The value supplied for the attribute is not of the same type as the attribute's value. |

### 5.10.5 HistoryUpdate

#### 5.10.5.1 Description

This *Service* is used to update historical values or *Events* of one or more *Nodes*. Several request parameters indicate how the *Server* is to update the historical value or *Event*. Valid actions are Insert, Replace or Delete.

#### 5.10.5.2 Parameters

Table 83 defines the parameters for the *Service*.

**Table 83 – HistoryUpdate Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 [UA Part 11]. It specifies the types of history updates that can be performed. The *ExtensibleParameter* type is defined in Clause 8.2 |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 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 Clause 7.28 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 Clause 7.5 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. |
| 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. |

#### 5.10.5.3 Service results

Table 84 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 84 – HistoryUpdate Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.10.5.4 StatusCodes

Table 85 defines values for the *statusCode* and *operationResult* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 85 – HistoryUpdate Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Good_ValueInserted | The value successfully inserted into history |
| Good_ValueReplaced | The value successfully replaced an existing value in history |
| Bad_ValueExists | A value already exists at the specified timestamp. |
| Bad_NoWriteRights | See Table 157 for the description of this result code. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |

## 5.11 Method Service Set

### 5.11.1 Overview

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

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

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

### 5.11.2 Call

#### 5.11.2.1 Description

This *Service* is used to call (invoke) a *method*. Each *method* call is invoked within the context of an existing *Session*. If the *Session* is terminated, the results of the *method's* execution cannot be returned to the *Client* and are discarded. This is independent of the task actually performed at the *Server*.

This *Service* provides for passing input and output arguments to/from a method. These arguments are defined by *Properties* of the method.

The sequence number in the *Service* request header is used by this *Service* to detect duplicate requests. *Servers* are responsible for tracking the sequence numbers used by this *Service* and for discarding requests that contain a sequence number that has already been used. *Clients* and *Servers* are expected to manage the rollover appropriately.

#### 5.11.2.2 Parameters

Table 86 defines the parameters for the *Service*.

**Table 86 – Call Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| objectId | NodeId | *NodeId* of the *Object* that defines the *Method*. See [UA 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*.<br>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 Clause 7.20 for *ResponseHeader* definition). |
| callResult | CallResult | Result of the *Method* call. |
| 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).<br>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*. |
| diagnosticInfo | DiagnosticInfo | Diagnostic information for the *StatusCode* of the CallResult. |
| 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*.<br>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*. |

#### 5.11.2.3 Service results

Table 87 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 87 – Call Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_InvalidArgument | See Table 156 for the description of this result code. |
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_MethodInvalid | The method id does not refer to a method for the specified object. |

## 5.12 MonitoredItem Service Set

### 5.12.1 MonitoredItem model

#### 5.12.1.1 Overview

*Clients* define *MonitoredItems* to subscribe to data and *Events*. Each *MonitoredItem* identifies the item to be monitored and the *Subscription* to use to send *Notifications*. The item to be monitored may be an arbitrary *Node Attribute*.

*Notifications* are data structures that describe the occurrence of data changes and *Events*. They are packaged into *NotificationMessages* for transfer to the *Client*. The *Subscription* periodically sends *NotificationMessages* at a user-specified publishing interval, and the cycle during which these messages are sent is called a publishing cycle.

Four primary *Attributes* are defined for *MonitoredItems* that tell the *Server* how the item is to be sampled, evaluated and reported. These *Attributes* are the sampling interval, the monitoring mode, the filter and the queue *Attributes*. Figure 25 illustrates these concepts.

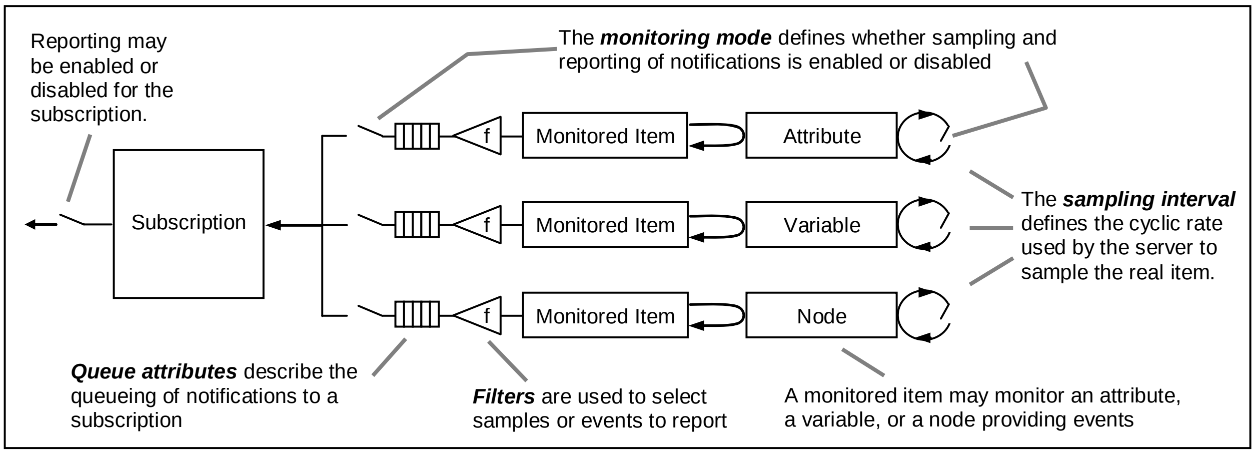
Reporting may be enabled or disabled for the subscription.
The **monitoring mode** defines whether sampling and reporting of notifications is enabled or disabled
Subscription
f
Monitored Item
Attribute
Variable
Node
The **sampling interval** defines the cyclic rate used by the server to sample the real 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 variable, or a node providing events

**Figure 25 – MonitoredItem Model**

This specification describes the monitoring of *Attributes* and *Variables* for value or status changes, including the caching of values and the monitoring of Nodes for Events.

*Attributes*, other than the *Value Attribute*, are monitored for a change in value. The filter is not used for these *Attributes*. Any change in value for these *Attributes* causes a *Notification* to be generated.

The *Value Attribute* is used when monitoring *Variables*. *Variable* values are monitored for a change in value or a change in their status. The filters defined in this specification (see Clause 8.3.2) and in [UA Part 8] are used to determine if the value change is large enough to cause a *Notification* to be generated for the *Variable*.

*Objects* and views can be used to monitor *Events*. *Events* are only available from *Nodes* where the *SubscribeToEvents* bit of the *EventNotifier Attribute* is set. The filter defined in this specification (see Clause 8.3.3) is used to determine if an *Event* received from the *Node* is sent to the *Client*. The filter also allows selecting *Properties* of the *EventType* that will be contained in the *Event* such as *EventId*, *EventType*, *SourceNode*, *Time* and *Description*.

[UA Part 3] describes the *Event* model and the base *EventTypes*.

The *Properties* of the base *EventTypes* and the representation of the base *EventTypes* in the *AddressSpace* are specified in [UA Part 5].

#### 5.12.1.2 Sampling interval

Each *MonitoredItem* created by the *Client* is assigned a sampling interval that is either inherited from the publishing interval of the *Subscription* or that is defined specifically to override that rate. The sampling interval indicates the fastest rate at which the *Server* should sample its underlying source for data changes.

The assigned sampling interval defines a “best effort” cyclic rate that the *Server* uses to sample the item from its source. “Best effort” in this context means that the *Server* does its best to sample at this rate. Sampling at rates faster than this rate is acceptable, but not necessary to meet the needs of the *Client*. How the *Server* deals with the sampling rate and how often it actually polls its data source internally is a *Server* implementation detail.

The *Client* may also specify 0 for the sampling interval, which indicates that the *Server* should use the fastest practical rate. It is expected that *Servers* will support only a limited set of sampling intervals to optimize their operation. If the exact interval requested by the *Client* is not supported by the *Server*, then the *Server* assigns to the *MonitoredItem* the most appropriate interval as determined by the *Server*. It returns this assigned interval to the *Client*. The *Server* Capabilities *Object* defined in [UA Part 5] identifies the sampling intervals supported by the *Server*.

The *Server* may support data that is collected based on a sampling model or generated based on an exception-based model. The fastest supported sampling interval may be equal to 0, which indicates that the data item is exception-based rather than being sampled at some period. Exception-based means that the underlying system reports changes of the data.

The *Client* may use the revised sampling interval values as a hint for setting the publishing interval as well as the keep alive count of a *Subscription*. If, for example, the smallest revised sampling interval of the *MonitoredItems* is 5 seconds, then the time before a keep-alive is sent should be longer than 5 seconds.

Note that, in many cases, the UA *Server* provides access to a decoupled system and therefore has no knowledge of the data update logic. In this case, even though the UA *Server* samples at the negotiated rate, the data might be updated by the underlying system at a much slower rate. In this case, changes can only be detected at this slower rate.

If the behaviour by which the underlying system updates the item is known, it will be available via the *MinSamplingInterval Attribute* defined in [UA Part 3].

*Clients* should also be aware that the sampling by the UA *Server* and the update cycle of the underlying system are usually not synchronized. This can cause additional delays in change detection, as illustrated in Figure 26.

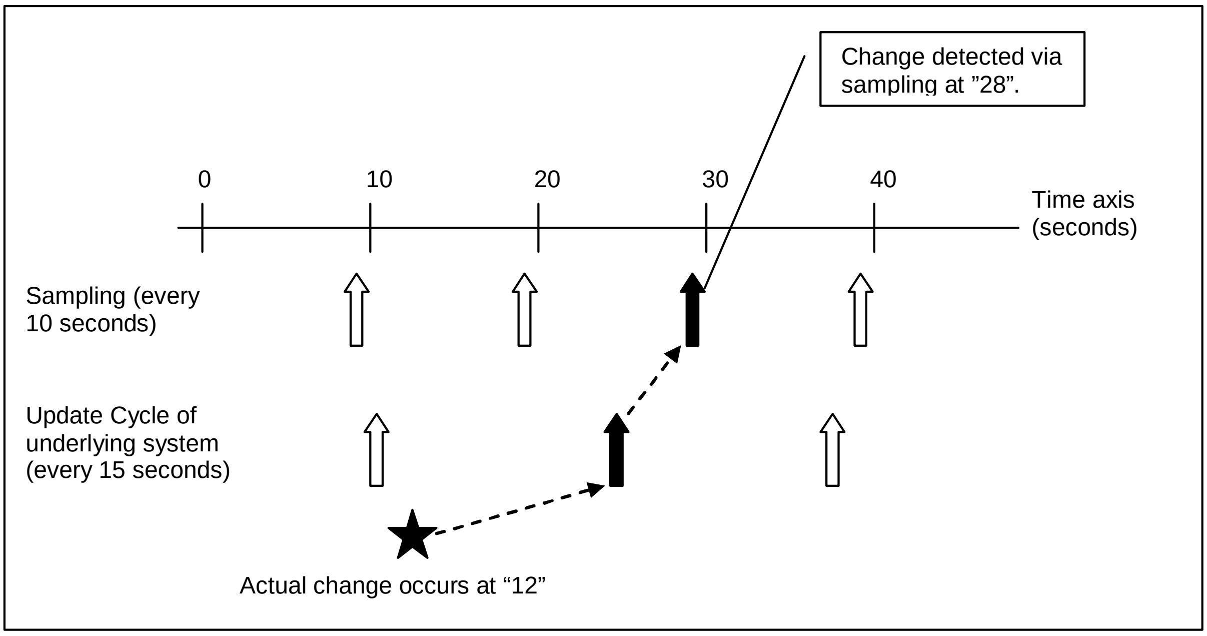
Change detected via sampling at ”28”.
0
10
20
30
40
Time axis (seconds)
Sampling (every 10 seconds)
Update Cycle of underlying system (every 15 seconds)
Actual change occurs at “12”

**Figure 26 – Typical delay in change detection.**

### 5.12.1.3 Monitoring mode

The monitoring mode *Attribute* is used to enable and disable the sampling of a *MonitoredItem*, and also to provide for independently enabling and disabling the reporting of *Notifications*. This capability allows a *MonitoredItem* to be configured to sample, sample and report, or neither.

Disabling sampling does not change the values of any of the other *MonitoredItem Attribute*s, such as its sampling interval.

#### 5.12.1.4 Filter

Each time a *MonitoredItem* is sampled, the *Server* evaluates the sample using the filter defined for the *MonitoredItem*. The filter *Attribute* defines the criteria that the *Server* uses to determine if a *Notification* should be generated for the sample. The type of filter is dependent on the type of the item that is being monitored. For example, the *Deadband* filter is used when monitoring *Variables* and the *EventFilter* is used when monitoring *Events*. Sampling and evaluation, including the use of filters, are described in this specification. Additional filters may be defined in other parts of this multi-part specification.

#### 5.12.1.5 Queue Attributes

If the sample passes the filter criteria, a *Notification* is generated and queued for transfer by the *Subscription*. The size of the queue is defined when the *MonitoredItem* is created. When the queue is full and a new *Notification* is received, the *Server* either discards the oldest *Notification* and queues the new one, or it simply discards the new one. The *MonitoredItem* is configured for one of these discard policies when the *MonitoredItem* is created. If a Notification is discarded for a *DataValue*, the *Overflow* bit in the *InfoBits* portion of the *DataValue statusCode* is set.

If the queue size is one and if the discard policy is to discard the oldest, the queue becomes a buffer that always contains the newest *Notification*. In this case, if the sampling interval of the *MonitoredItem* is faster than the publishing interval of the *Subscription*, the *MonitoredItem* will be over sampling and the *Client* will always receive the most up-to-date value.

On the other hand, the *Client* may want to subscribe to a continuous stream of *Notifications* without any gaps, but does not want them reported at the sampling interval. In this case, the *MonitoredItem* would be created with a queue size large enough to hold all *Notifications* generated between two consecutive publishing cycles. Then, at each publishing cycle, the *Subscription* would send all *Notifications* queued for the *MonitoredItem* to the *Client*. The *Server* is required to return values for any particular item in chronological order.

The *Server* may be sampling at a faster rate than the sampling interval to support other *Clients*; the *Client* should only expect values at the negotiated sampling interval. The *Server* may deliver fewer values than dictated by the sampling interval, based on the filter and implementation constraints. If a *Deadband* filter is configured for a *MonitoredItem*, it is always applied to the newest value in the queue compared to the current sample.

If, for example, the *AbsoluteDeadband* in the *DataChangeFilter* is “10”, the queue could consist of values in the following order:

- 100
- 111
- 101
- 89
- 100

Queuing of data may result in unexpected behaviour when using a *Deadband* filter and the number of encountered changes is larger than the number of values that can be maintained. It is realistically possible that, due to the discard policy “`discardOldest=TRUE`”, the new first value in the queue will not exceed the *Deadband* limit of the previous value sent to the *Client*.

The queue size is the maximum value supported by the *Server* when monitoring *Events*. In this case, the *Server* is responsible for the *Event* buffer. If *Events* are lost, an *Event* of the type *EventQueueOverflow* is generated.

### 5.12.1.6 Triggering model

The *MonitoredItems Service* allows adding items that are reported only when some other item (the triggering item) triggers. This is done by creating links between the triggered items and the items to report. The monitoring mode of the items to report is set to sampling-only so that it will sample and queue *Notifications* without reporting them. Figure 27 illustrates this concept.

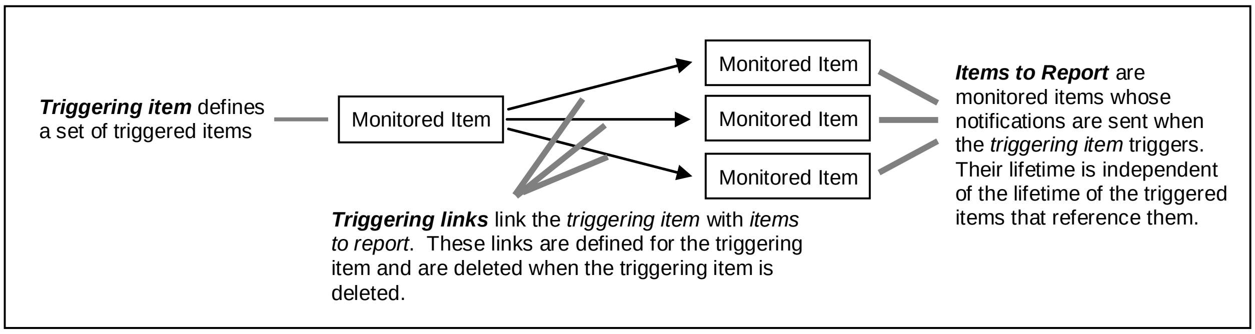
***Triggering item*** defines a set of triggered items
Monitored Item
Monitored Item
Monitored Item
Monitored Item
***Items to Report*** are monitored items whose notifications are sent when the *triggering item* triggers. Their lifetime is independent of the lifetime of the triggered items that reference 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 27 – Triggering Model**

The triggering mechanism is a useful feature that allows *Clients* to reduce the data volume on the wire by configuring some items to sample frequently but only report when some other *Event* happens.

The following triggering behaviours are specified:

a) 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.

b) 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.

c) 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.

d) The monitoring mode of the item to report indicates that reporting is enabled. In this case, the item to report is reported only once (when the item to report triggers), effectively causing the triggering item to be ignored.

*Clients* create and delete triggering links between a triggering item and a set of items to report. If the *MonitoredItem* that represents an item to report is deleted before its associated triggering link is deleted, the triggering link is also deleted, but the triggering item is otherwise unaffected.

Deletion of a *MonitoredItem* should not be confused with the removal of the *Attribute* that it monitors. If the *Node* that contains the *Attribute* being monitored is deleted, the *MonitoredItem* generates a *Notification* with a *StatusCode Bad_UnknownNodeId* that indicates the deletion, but the *MonitoredItem* is not deleted.

### 5.12.2 CreateMonitoredItems

#### 5.12.2.1 Description

This *Service* is used to create and add one or more *MonitoredItems* to a *Subscription*. A *MonitoredItem* is deleted automatically by the *Server* when the *Subscription* is deleted. Deleting a *MonitoredItem* causes its entire set of triggered item links to be deleted, but has no effect on the *MonitoredItems* referenced by the triggered items.

Calling the *CreateMonitoredItems Service* repetitively to add a small number of *MonitoredItems* each time may adversely affect the performance of the *Server*. *Servers* may introduce delays between repetitive calls to this *Service* or enforce other measures to discourage this behaviour. *Clients* are cautioned to not use this *Service* in this manner. Instead, *Clients* should add a complete set of *MonitoredItems* to a *Subscription* whenever possible.

When a *MonitoredItem* is added, the *Server* performs initialization processing for it. The initialization processing is defined by the *Notification* type of the item being monitored. *Notification* types are specified in this specification and in the Access Types Parts of this multi-part specification, such as [UA Part 8]. See Clause 4 of [UA Part 1] for a description of the Access Type Parts.

When a user adds a monitored item that the user is denied read access to, the add operation for the item must succeed and the bad status Bad_NoReadRights or Bad_UserAccessDenied will be returned in the Publish response. This is the same behaviour for the case where the access rights are changed after the call to *CreateMonitoredItem*. If the access rights change to read rights, the *Server* must start sending data for the *MonitoredItem*.

### 5.12.2.2 Parameters

Table 88 defines the parameters for the *Service*.

**Table 88 – CreateMonitoredItems Service Parameters**

| **Name** | **Type** | **Description** |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* that will report *Notifications* for this *MonitoredItem* (see Clause 7.9 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 Clause 7.29.<br>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 Clause 7.16. |
| monitoringMode | enum MonitoringMode | The monitoring mode to be set for the *MonitoredItem*. The *MonitoringMode* enumeration is defined in Clause 7.12. |
| requestedAttributes | Monitoring Attributes | The requested monitoring *Attributes*. *Servers* negotiate the values of these *Attributes* based on the override policy of the *Subscription* and the capabilities of the *Server*. The *MonitoringAttributes* type is defined in Clause 7.10. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 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 Clause 7.28 for *StatusCode* definition). |
| monitoredItemId | IntegerId | *Server*-assigned id for the *MonitoredItem* (see Clause 7.9 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 (see Clause 7.6 for *Duration* definition).<br>This value is based on a number of factors, including capabilities of the underlying system. |
| revisedQueueSize | Counter | The actual queue size that the *Server* will use. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the *MonitoredItems* to create (see Clause 7.5 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. |

### 5.12.2.3 Service results

Table 89 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 89 – CreateMonitoredItems Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_TimestampsToReturnInvalid | See Table 156 for the description of this result code. |
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

#### 5.12.2.4 StatusCodes

Table 90 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 90 – CreateMonitoredItems Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_MonitoringModeInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdInvalid | See Table 157 for the description of this result code. |
| Bad_NodeIdUnknown | See Table 157 for the description of this result code. |
| Bad_AttributeIdInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeInvalid | See Table 157 for the description of this result code. |
| Bad_IndexRangeNoData | See Table 157 for the description of this result code. |
| Bad_DataEncodingInvalid | See Table 157 for the description of this result code. |
| Bad_DataEncodingUnsupported | See Table 157 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |
| Bad_FilterNotAllowed | See Table 156 for the description of this result code. |

### 5.12.3 ModifyMonitoredItems

#### 5.12.3.1 Description

This *Service* is used to modify *MonitoredItems* of a *Subscription*. Changes to the sampling interval and filter take effect at the beginning of the next sampling interval (the next time the sampling timer expires).

#### 5.12.3.2 Parameters

Table 91 defines the parameters for the *Service*.

**Table 91 – ModifyMonitoredItems Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* used to qualify the *monitoredItemId* (see Clause 7.9 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 Clause 7.29. When monitoring *Events*, this applies only to *Event* fields that are of type *DataValue*. |
| itemsToModify [] | MonitoredItemMo difyRequest | The list of *MonitoredItems* to modify. |
| monitoredItemId | IntegerId | *Server*-assigned id for the *MonitoredItem*. |
| requested Attributes | Monitoring Attributes | The requested values for the monitoring *Attributes*. The *MonitoringAttributes* type is defined in Clause 7.10. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | MonitoredItemMo difyResult | 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 Clause 7.28 for *StatusCode* definition). |
| revisedSampling Interval | Duration | The actual sampling interval that the *Server* will use (see Clause 7.6 for *Duration* definition). The *Server* returns the value it will actually use for the sampling interval. This value is based on a number of factors, including capabilities of the underlying system. |
| revisedQueueSize | Counter | The actual queue size that the *Server* will use. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the *MonitoredItems* to modify (see Clause 7.5 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. |

#### 5.12.3.3 Service results

Table 92 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 92 – ModifyMonitoredItems Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_TimestampsToReturnInvalid | See Table 156 for the description of this result code. |
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

#### 5.12.3.4 StatusCodes

Table 93 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 93 – ModifyMonitoredItems Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_MonitoredItemIdInvalid | See Table 157 for the description of this result code. |
| Bad_ExtensibleParameterInvalid | See Table 156 for the description of this result code. |
| Bad_ExtensibleParameterUnsupported | See Table 156 for the description of this result code. |
| Bad_FilterNotAllowed | See Table 156 for the description of this result code. |

### 5.12.4 SetMonitoringMode

#### 5.12.4.1 Description

This *Service* is used to set the monitoring mode for one or more *MonitoredItems* of a *Subscription*. Setting the mode to DISABLED or SAMPLING causes all queued *Notifications* to be deleted.

#### 5.12.4.2 Parameters

Table 94 defines the parameters for the *Service*.

**Table 94 – SetMonitoringMode Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* used to qualify the *monitoredItemIds* (see Clause 7.9 for *IntegerId* definition). |
| monitoringMode | enum MonitoringMode | The monitoring mode to be set for the *MonitoredItems*. The *MonitoringMode* enumeration is defined in Clause 7.12. |
| monitoredItemIds [] | IntegerId | List of *Server*-assigned ids for the *MonitoredItems*. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *MonitoredItems* to enable/disable (see Clause 7.28 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 Clause 7.5 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. |

#### 5.12.4.3 Service results

Table 95 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 95 – SetMonitoringMode Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |
| Bad_MonitoringModeInvalid | See Table 157 for the description of this result code. |

#### 5.12.4.4 StatusCodes

Table 96 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 96 – SetMonitoringMode Operation Level Result Codes**

| Value | Description |
|---|---|
| Bad_MonitoredItemIdInvalid | See Table 157 for the description of this result code. |

### 5.12.5 SetTriggering

#### 5.12.5.1 Description

This *Service* is used to create and delete triggering links for a triggering item. The triggering item and the items to report must belong to the same *Subscription*.

Each triggering link links a triggering item to an item to report. Each link is represented by the *MonitoredItem* id for the item to report. An error code is returned if this id is invalid.

### 5.12.5.2 Parameters

Table 97 defines the parameters for the *Service*.

**Table 97 – SetTriggering Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* that contains the triggering item and the items to report (see Clause 7.9 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 Clause 7.20 for *ResponseHeader* definition). |
| addResults [] | StatusCode | List of *StatusCodes* for the items to add (see Clause 7.28 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 Clause 7.5 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. |
| 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. |

### 5.12.5.3 Service results

Table 98 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Clause 7.28.

**Table 98 – SetTriggering Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |
| Bad_MonitoredItemIdInvalid | See Table 157 for the description of this result code. |

### 5.12.5.4 StatusCodes

Table 99 defines values for the results parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 99 – SetTriggering Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_MonitoredItemIdInvalid | See Table 157 for the description of this result code. |

### 5.12.6 DeleteMonitoredItems

#### 5.12.6.1 Description

This *Service* is used to remove one or more *MonitoredItems* of a *Subscription*. When a *MonitoredItem* is deleted, its triggered item links are also deleted.

Successful removal of a *MonitoredItem*, however, might not remove *Notifications* for the *MonitoredItem* that are in the process of being sent by the *Subscription*. Therefore, *Clients* may receive *Notifications* for the *MonitoredItem* after they have received a positive response that the *MonitoredItem* has been deleted.

#### 5.12.6.2 Parameters

Table 100 defines the parameters for the *Service*.

**Table 100 – DeleteMonitoredItems Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* that contains the *MonitoredItems* to be deleted (see Clause 7.9 for *IntegerId* definition). |
| monitoredItemIds [] | IntegerId | List of *Server*-assigned ids for the *MonitoredItems* to be deleted. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *MonitoredItems* to delete (see Clause 7.28 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 Clause 7.5 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. |

#### 5.12.6.3 Service results

Table 101 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 101 – DeleteMonitoredItems Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

#### 5.12.6.4 StatusCodes

Table 102 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 102 – DeleteMonitoredItems Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_MonitoredItemIdInvalid | See Table 157 for the description of this result code. |

## 5.13 Subscription Service Set

### 5.13.1 Subscription model

#### 5.13.1.1 Description

*Subscriptions* are used to report *Notifications* to the *Client*. Their general behaviour is summarized below. Their precise behaviour is described in Clause 5.13.1.2.

a) *Subscriptions* have a set of *MonitoredItems* assigned to them by the *Client*. *MonitoredItems* generate *Notifications* that are to be reported to the *Client* by the *Subscription* (see Clause 5.12.1 for a description of *MonitoredItems*).

b) *Subscriptions* have a publishing interval. The publishing interval of a *Subscription* defines the cyclic rate at which the *Subscription* executes. Each time it executes, it attempts to send a *NotificationMessage* to the *Client*. *NotificationMessages* contain *Notifications* that have not yet been reported to *Client*.

c) *NotificationMessages* are sent to the *Client* in response to *Publish* requests. *Publish* requests are normally queued to the *Session* as they are received, and one is dequeued and processed by a subscription related to this *Session* each publishing cycle, if there are *Notifications* to report. When there are not, the *Publish* request is not dequeued from the *Session*, and the *Server* waits until the next cycle and checks again for *Notifications*.

d) At the beginning of a cycle, if there are *Notifications* to send but there are no *Publish* requests queued, the *Server* enters a wait state for a *Publish* request to be received. When one is received, it is processed immediately without waiting for the next publishing cycle.

e) *NotificationMessages* are uniquely identified by sequence numbers that enable *Clients* to detect missed *Messages*. The publishing interval also defines the default sampling interval for its *MonitoredItems*.

f) *Subscriptions* have a keep-alive counter that counts the number of consecutive publishing cycles in which there have been no *Notifications* to report to the *Client*. When the maximum keep-alive count is reached, a *Publish* request is dequeued and used to return a keep-alive *Message*. This keep-alive *Message* informs the *Client* that the *Subscription* is still active. Each keep-alive *Message* is a response to a Publish request in which the *notificationMessage* parameter does not contain any Notifications and that contains the sequence number of the next *NotificationMessage* that is to be sent. In the sections that follow, the term *NotificationMessage* refers to a response to a Publish request in which the *notificationMessage* parameter actually contains one or more *Notifications*, as opposed to a *keep-alive Message* in which this parameter contains no *Notifications*. The maximum keep-alive count is set by the *Client* during *Subscription* creation and may be subsequently modified using the *ModifySubscription Service*. Similar to *Notification* processing described in (c) above, if there are no *Publish* requests queued, the *Server* waits for the next one to be received and sends the keep-alive immediately without waiting for the next publishing cycle.

g) Publishing by a *Subscription* may be enabled or disabled by the *Client* when created, or subsequently using the *SetPublishingMode Service*. Disabling causes the *Subscription* to cease sending *NotificationMessages* to the *Client*. However, the *Subscription* continues to execute cyclically and continues to send keep-alive *Messages* to the *Client*.

h) *Subscriptions* have a lifetime counter that counts the number of consecutive publishing cycles in which there have been no *Publish* requests received from the *Client*. When this counter reaches the value calculated for the lifetime of a *Subscription* based on the MaxKeepAliveCount parameter in the *CreateSubscription Service* (Clause 5.13.2), the *Subscription* is closed. Closing the *Subscription* causes its *MonitoredItems* to be deleted.

i) *Subscriptions* maintain a retransmission queue of sent *NotificationMessages*. *NotificationMessages* are retained in this queue until they are acknowledged or until they have been in the queue for a minimum of one keep-alive interval. *Clients* are required to acknowledge *NotificationMessages* as they are received.

The sequence number is an unsigned 32-bit integer that is incremented by one for each *NotificationMessage* sent. The value 0 is never used for the sequence number. The first *NotificationMessage* sent on a *Subscription* has a sequence number of 1. If the sequence number rolls over, it rolls over to 1.

When a *Subscription* is created, the first *Message* is sent at the end of the first publishing cycle to inform the *Client* that the *Subscription* is operational. A *NotificationMessage* is sent if there are *Notifications* ready to be reported. If there are none, a keep-alive *Message* is sent instead that contains a sequence number of 1, indicating that the first *NotificationMessage* has not yet been sent. This is the only time a keep-alive *Message* is sent without waiting for the maximum keep-alive count to be reached, as specified in (f) above.

The value of the sequence number is never reset during the lifetime of a *Subscription*. Therefore, the same sequence number will not be reused on a *Subscription* until over four billion *NotificationMessages* have been sent. At a continuous rate of one thousand *NotificationMessages* per second on a given *Subscription*, it would take roughly fifty days for the same sequence number to be reused. This allows *Clients* to safely treat sequence numbers as unique.

Sequence numbers are also used by *Clients* to acknowledge the receipt of *NotificationMessages*. *Publish* requests allow the *Client* to acknowledge all *Notifications* up to a specific sequence number and to acknowledge the sequence number of the last *NotificationMessage* received. One or more gaps may exist in between. Acknowledgements allow the *Server* to delete *NotificationMessages* from its retransmission queue.

*Clients* may ask for retransmission of selected *NotificationMessages* using the Republish *Service*. This *Service* returns the requested *Message*.

#### 5.13.1.2 State table

The state table formally describes the operation of the *Subscription*. The following model of operations is described by this state table. This description applies when publishing is enabled or disabled for the *Subscription*.

After creation of the *Subscription*, the *Server* starts the publishing timer and restarts it whenever it expires. If the timer expires the number of times defined for the *Subscription* lifetime without having received a *Subscription Service* request from the *Client*, the *Subscription* assumes that the *Client* is no longer present, and terminates.

*Clients* send *Publish* requests to *Servers* to receive *Notifications*. *Publish* requests are not directed to any one *Subscription* and, therefore, may be used by any *Subscription*. Each contains acknowledgements for one or more *Subscriptions*. These acknowledgements are processed when the *Publish* request is received. The *Server* then queues the request in a queue shared by all *Subscriptions*, except in the following cases:

a) The previous *Publish* response indicated that there were still more *Notifications* ready to be transferred and there were no more *Publish* requests queued to transfer them.
b) The publishing timer of a *Subscription* expired and there were either *Notifications* to be sent or a keep-alive *Message* to be sent.

In these cases, the newly received *Publish* request is processed immediately by the first *Subscription* to encounter either case (a) or case (b).

Each time the publishing timer expires, it is immediately reset. If there are *Notifications* or a keep-alive *Message* to be sent, it dequeues and processes a *Publish* request. When a *Subscription* processes a *Publish* request, it accesses the queues of its *MonitoredItems* and dequeues its *Notifications*, if any. It returns these *Notifications* in the response, setting the *moreNotifications* flag if it was not able to return all available *Notifications* in the response.

If there were *Notifications* or a keep-alive *Message* to be sent but there were no *Publish* requests queued, the *Subscription* assumes that the *Publish* request is late and waits for the next *Publish* request to be received, as described in case (b).

If the *Subscription* is disabled when the publishing timer expires or if there are no *Notifications* available, it enters the keep-alive state and sets the keep-alive counter to its maximum value as defined for the *Subscription*.

While in the keep-alive state, it checks for *Notifications* each time the publishing timer expires. If one or more have been generated, a *Publish* request is dequeued and a *NotificationMessage* is returned in the response. However, if the publishing timer reaches the maximum keep-alive count without a *Notification* becoming available, a *Publish* request is dequeued and a keep-alive *Message* is returned in the response. The *Subscription* then returns to the normal state of waiting for the publishing timer to expire again. If, in either of these cases, there are no *Publish* requests queued, the *Subscription* waits for the next *Publish* request to be received, as described in case (b).

The *Subscription* states are defined in Table 103.

**Table 103 – Subscription States**

| State | Description |
|---|---|
| CLOSED | The *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 104. The following rules and conventions apply:

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

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

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

d) *Service* request and internal *Events* may be accompanied by conditions represented by functions whose return value is tested. Functions are identified by "()" after their name. They are described in Clause 5.13.1.4.

e) When an *Event* is received, the first transition for the current state is located and the transitions are searched sequentially for the first transition that meets the *Event* or conditions criteria. If none are found, the *Event* is ignored.

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

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

**Table 104 – Subscription State Table**

| # | Current State | Event/Conditions | Action | Next State |
|---|---|---|---|---|
| 1 | CLOSED | Receive CreateSubscription Request | CreateSubscription() | CREATING |
| 2 | CREATING | CreateSubscription fails | ReturnNegativeResponse() | CLOSED |
| 3 | CREATING | CreateSubscription succeeds | InitializeSubscription()<br>MessageSent = FALSE<br>ReturnResponse() | NORMAL |
| 4 | NORMAL | Receive *Publish* Request<br>&&<br>(<br>PublishingEnabled == FALSE<br>\|\|<br>(PublishingEnabled == TRUE<br>&& MoreNotifications == FALSE)<br>) | ResetLifetimeCounter()<br>DeleteAckedNotificationMsgs()<br>EnqueuePublishingReq() | NORMAL |
| 5 | NORMAL | Receive *Publish* Request<br>&& PublishingEnabled == TRUE<br>&& MoreNotifications == TRUE | ResetLifetimeCounter()<br>DeleteAckedNotificationMsgs()<br>ReturnNotifications()<br>MessageSent = TRUE | NORMAL |
| 6 | NORMAL | PublishingTimer Expires<br>&& PublishingReqQueued == TRUE<br>&& PublishingEnabled == TRUE<br>&& NotificationsAvailable == TRUE | StartPublishingTimer()<br>DequeuePublishReq()<br>ReturnNotifications()<br>MessageSent == TRUE | NORMAL |
| 7 | NORMAL | PublishingTimer Expires<br>&& PublishingReqQueued == TRUE<br>&& MessageSent == FALSE<br>&&<br>(<br>PublishingEnabled == FALSE<br>\|\|<br>(PublishingEnabled == TRUE<br>&& NotificationsAvailable == FALSE)<br>) | StartPublishingTimer()<br>DequeuePublishReq()<br>ReturnKeepAlive()<br>MessageSent == TRUE | NORMAL |
| 8 | NORMAL | PublishingTimer Expires<br>&& PublishingReqQueued == FALSE<br>&&<br>(<br>MessageSent == FALSE<br>\|\|<br>(PublishingEnabled == TRUE<br>&& NotificationsAvailable == TRUE)<br>) | StartPublishingTimer() | LATE |
| 9 | NORMAL | PublishingTimer Expires<br>&& MessageSent == TRUE<br>&&<br>(<br>PublishingEnabled == FALSE<br>\|\|<br>(PublishingEnabled == TRUE<br>&& NotificationsAvailable == FALSE)<br>) | StartPublishingTimer()<br>ResetKeepAliveCounter() | KEEPALIVE |
| 10 | LATE | Receive *Publish* Request<br>&& PublishingEnabled == TRUE<br>&& (NotificationsAvailable == TRUE<br>\|\| MoreNotifications == TRUE) | ResetLifetimeCounter()<br>DeleteAckedNotificationMsgs()<br>ReturnNotifications()<br>MessageSent = TRUE | NORMAL |
| 11 | LATE | Receive *Publish* Request<br>&&<br>(<br>PublishingEnabled == FALSE<br>\|\|<br>(PublishingEnabled == TRUE<br>&& NotificationsAvailable == FALSE<br>&& MoreNotifications == FALSE)<br>) | ResetLifetimeCounter()<br>DeleteAckedNotificationMsgs()<br>ReturnKeepAlive()<br>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<br>\|\| LATE<br>\|\| KEEPALIVE | Receive ModifySubscription Request | ResetLifetimeCounter()<br>UpdateSubscriptionParams()<br>ReturnResponse() | SAME |
| 19 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive SetPublishingMode Request | ResetLifetimeCounter()<br>SetPublishingEnabled()<br>MoreNotifications = FALSE<br>ReturnResponse() | SAME |
| 20 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive Republish Request<br>&& RequestedMessageFound == TRUE | ResetLifetimeCounter()<br>ReturnResponse() | SAME |
| 21 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive Republish Request<br>&& RequestedMessageFound == FALSE | ResetLifetimeCounter()<br>ReturnNegativeResponse() | SAME |
| 22 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive TransferSubscriptions Request<br>&& SessionChanged() == FALSE | ResetLifetimeCounter()<br>ReturnNegativeResponse () | SAME |
| 23 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive TransferSubscriptions Request<br>&& SessionChanged() == TRUE<br>&& ClientValidated() ==TRUE | SetSession()<br>ResetLifetimeCounter()<br>DeleteAckedNotificationMsgs()<br>ReturnResponse() | SAME |
| 24 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive TransferSubscriptions Request<br>&& SessionChanged() == TRUE<br>&& ClientValidated() == FALSE | ReturnNegativeResponse() | SAME |
| 25 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive DeleteSubscriptions Request<br>&& SubscriptionAssignedToClient ==TRUE | DeleteMonitoredItems()<br>DeleteClientPublReqQueue() | CLOSED |
| 26 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | Receive DeleteSubscriptions Request<br>&& SubscriptionAssignedToClient ==FALSE | ResetLifetimeCounter()<br>ReturnNegativeResponse() | SAME |
| 27 | NORMAL<br>\|\| LATE<br>\|\| KEEPALIVE | LifetimeTimer Expires | DeleteMonitoredItems() | CLOSED |

### 5.13.1.3 State Variables and parameters

The state *Variables* are defined alphabetically in Table 105.

**Table 105 – State variables and parameters**

| State Variable | Description |
|---|---|
| MoreNotifications | A boolean *Variable* that is set to TRUE only by the CreateNotificationMsg() when there were too many *Notifications* for a single *NotificationMessage*. |
| LatePublishRequest | A boolean *Variable* that is set to TRUE to reflect that, the last time the publishing timer expired, there were no *Publish* requests queued. |
| LifetimeCounter | A *Variable* that contains the number of consecutive publishing timer expirations without *Client* activity before the *Subscription* is terminated. |
| MessageSent | A boolean *Variable* 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 *Variable* 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 *Variable* that is set to TRUE only when there is a *Publish* request *Message* enqueued to the *Subscription*. |
| RequestedMessageFound | A boolean *Variable* that is set to TRUE only when the *Message* requested to be retransmitted was found in the retransmission queue. |
| SeqNum | The *Variable* that records the value of the sequence number used in *NotificationMessages* |
| SubscriptionAssignedToClient | A boolean *Variable* 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 106.

**Table 106 – 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*.<br>Save the newly-created *NotificationMessage* in the retransmission queue.<br>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. |
| EnqueuePublishingReq() | Enqueue the publishing request |
| InitializeSubscription() | ResetLifetimeCounter()<br>MoreNotifications = FALSE<br>PublishRateChange = FALSE<br>PublishingEnabled = value of publishingEnabled parameter in the CreateSubscription request<br>PublishingReqQueued = FALSE<br>SeqNum = 0<br>SetSession()<br>StartPublishingTimer() |
| 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* (Clause 5.13.2). |
| ReturnKeepAlive() | CreateKeepAliveMsg()<br>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()<br>ReturnResponse()<br>If (MoreNotifications == TRUE) && (PublishingReqQueued == TRUE)<br>{<br>   DequeuePublishReq()<br>   Loop through this function again<br>} |
| 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 107 defines the parameters for the *Service*.

**Table 107 – CreateSubscription Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | Request Header | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| requestedPublishing Interval | Duration | This interval defines the cyclic rate that the *Subscription* is being requested to return *Notifications* to the *Client* (see Clause 7.6 for *Duration* definition). This interval is expressed in milliseconds. This interval is represented by the publishing timer in the *Subscription* state table (see Clause 5.13.1.2).<br>The negotiated value for this parameter returned in the response is used as the default sample interval for *MonitoredItems* assigned to this *Subscription*. |
| requestedLifetimeCount | Counter | Requested lifetime count (see Clause 7.3 for *Counter* definition). The lifetime count must be a mimimum of three times the keep keep-alive count.<br>When the publishing timer has expired this number of times without a *NotificationMessage* being sent, the *Subscription* will be deleted by the *Server*. |
| requestedMaxKeepAlive Count | Counter | Requested maximum keep-alive count (see Clause 7.3 for *Counter* definition). When the publishing timer has expired this number of times without a *NotificationMessage* being sent, the *Subscription* sends a keep-alive *Message* to the *Client*. |
| publishingEnabled | Boolean | A *Boolean* parameter with the following values :<br>TRUE publishing is enabled for the *Subscription*.<br>FALSE publishing is disabled for the *Subscription*.<br>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. Any *Subscription* that needs to send a keep-alive *Message* must take precedence regardless of its *priority*, in order to prevent the *Subscription* from expiring.<br>A Client that does not require special priority settings should set this value to zero. |
| | | |
| **Response** | | |
| responseHeader | Response Header | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* (see Clause 7.9 for *IntegerId* definition). This identifier must 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 (see Clause 7.6 for *Duration* 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. |
| revisedLifetimeCount | Counter | The lifetime of the *Subscription* must be a minimum of three times the keep-alive interval negotiated by the *Server.* |
| revisedMaxKeepAliveCount | Counter | The actual maximum keep-alive count (see Clause 7.3 for *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 108 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 108 – CreateSubscription Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_TooManySubscriptions | The *Server* has reached its maximum number of subscriptions. |

### 5.13.3 ModifySubscription

#### 5.13.3.1 Description

This *Service* is used to modify a *Subscription*.

#### 5.13.3.2 Parameters

Table 109 defines the parameters for the *Service*. Changes to the publishing interval become effective the next time the publishing timer expires.

**Table 109 – ModifySubscription Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* (see Clause 7.9 for *IntegerId* definition). |
| requestedPublishingInterval | Duration | This interval defines the cyclic rate that the *Subscription* is being requested to return *Notifications* to the *Client* (see Clause 7.6 for *Duration* definition). This interval is expressed in milliseconds. This interval is represented by the publishing timer in the *Subscription* state table (see Clause 5.13.1.2).<br>The negotiated value for this parameter returned in the response is used as the default sample interval for *MonitoredItems* assigned to this *Subscription*. |
| requestedLifetimeCount | Counter | Requested lifetime count (see Clause 7.3 for *Counter* definition). The lifetime count must be a mimimum of three times the keep keep-alive count.<br>When the publishing timer has expired this number of times without a *NotificationMessage* being sent, the *Subscription* will be deleted by the *Server*. |
| requestedMaxKeepAliveCount | Counter | Requested maximum keep-alive count (see Clause 7.3 for *Counter* definition). When the publishing timer has expired this number of times without a *NotificationMessage* being sent, the *Subscription* sends a keep-alive *Message* to the *Client*. |
| 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* must take precedence regardless of its *priority*, in order to prevent the *Subscription* from expiring.<br>A Client that does not require special priority settings should set this value to zero. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| revisedPublishingInterval | Duration | The actual publishing interval that the *Server* will use, expressed in milliseconds (see Clause 7.6 for *Duration* 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. |
| revisedLifetimeCount | Counter | The lifetime of the *Subscription* must be a minimum of three times the keep-alive interval negotiated by the *Server.* |
| revisedMaxKeepAliveCount | Counter | The actual maximum keep-alive count (see Clause 7.3 for *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 110 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 110 – ModifySubscription Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

### 5.13.4 SetPublishingMode

#### 5.13.4.1 Description

This *Service* is used to enable sending of *Notifications* on one or more *Subscriptions*.

#### 5.13.4.2 Parameters

Table 111 defines the parameters for the *Service*.

**Table 111 – SetPublishingMode Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| publishingEnabled | Boolean | A *Boolean* parameter with the following values :<br>TRUE publishing of *NotificationMessages* is enabled for the *Subscription*.<br>FALSE publishing of *NotificationMessages* is disabled for the *Subscription*.<br>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 Clause 7.9 for *IntegerId* definition). |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *Subscriptions* to enable/disable (see Clause 7.28 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 Clause 7.5 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. |

#### 5.13.4.3 Service results

Table 112 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 112 – SetPublishingMode Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.13.4.4 StatusCodes

Table 113 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 113 – SetPublishingMode Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

### 5.13.5 Publish

#### 5.13.5.1 Description

This *Service* is used for two purposes. First, it is used to acknowledge the receipt of *NotificationMessages* for one or more *Subscriptions*. Second, it is used to request the *Server* to return a *NotificationMessage* or a keep-alive *Message*. Since *Publish* requests are not directed to a specific *Subscription*, they may be used by any *Subscription*. Clause 5.13.1.2 describes the use of the *Publish Service*.

*Client* strategies for issuing *Publish* requests may vary depending on the networking delays between the *Client* and the *Server*. In many cases, the *Client* may wish to issue a *Publish* request immediately after creating a *Subscription*, and thereafter, immediately after receiving a *Publish* response.

In other cases, especially in high latency networks, the *Client* may wish to pipeline *Publish* requests to ensure cyclic reporting from the *Server*. Pipelining involves sending more than one *Publish* request for each *Subscription* before receiving a response. For example, if the network introduces a delay between the *Client* and the *Server* of 5 seconds and the publishing interval for a *Subscription* is one second, then the *Client* will have to issue *Publish* requests every second instead of waiting for a response to be received before sending the next request.

A server should limit the number of active *Publish* requests to avoid an infinite number since it is expected that the *Publish* requests are queued in the *Server*. But a Server must accept more queued *Publish* requests than created Subscriptions. It is expected that a *Server* supports several *Publish* requests per *Subscription*. The *Server* must return the *Service* result Bad_TooManyPublishRequests if the number of *Publish* requests exceeds the limit. If a *Client* receives this *Service* result for a *Publish* request it must not issue another *Publish* request before one of its outstanding *Publish* requests is returned from the Server.

### 5.13.5.2 Parameters

Table 114 defines the parameters for the *Service*.

**Table 114 – Publish Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 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 Clause 7.9 for *IntegerId* definition). |
| sequenceNumber | Counter | The sequence number being acknowledged (see Clause 7.3 for *Counter* definition). The *Server* may delete the *Message* with this sequence number from its retransmission queue. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| subscriptionId | IntegerId | The *Server*-assigned identifier for the *Subscription* for which *Notifications* are being returned (see Clause 7.9 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 Clause 7.3 for *Counter* definition). |
| moreNotifications | Boolean | A *Boolean* parameter with the following values :<br>TRUE the number of *Notifications* that were ready to be sent could not be sent in a single response.<br>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 Clause 7.13. |
| results [] | StatusCode | List of results for the acknowledgements (see Clause 7.28 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 Clause 7.5 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. |

### 5.13.5.3 Service results

Table 115 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 115 – Publish Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_TooManyPublishRequests | The server has reached the maximum number of queued publish requests. |
| Bad_NoSubscription | There is no subscription available for this session. |

### 5.13.5.4 StatusCodes

Table 116 defines values for the *acknowledgeResults* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 116 – Publish Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

### 5.13.6 Republish

#### 5.13.6.1 Description

This *Service* requests the *Subscription* to republish a *NotificationMessage* from its retransmission queue. If the *Server* does not have the requested *Message* in its retransmission queue, it returns an error response.

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

#### 5.13.6.2 Parameters

Table 117 defines the parameters for the *Service*.

**Table 117 – Republish Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionId | IntegerId | The *Server* assigned identifier for the *Subscription* to be republished (see Clause 7.9 for *IntegerId* definition). |
| retransmitSequence Number | Counter | The sequence number of a specific *NotificationMessage* to be republished (see Clause 7.3 for *Counter* definition). |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| notificationMessage | Notification Message | The requested *NotificationMessage*. The *NotificationMessage* parameter type is specified in Clause 7.13. |

#### 5.13.6.3 Service results

Table 118 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 118 – Republish Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |
| Bad_MessageNotAvailable | The requested message is no longer available. |

### 5.13.7 TransferSubscriptions

#### 5.13.7.1 Description

This *Service* is used to transfer a *Subscription* and its *MonitoredItems* from one *Session* to another. For example, a *Client* may need to reopen a *Session* and then transfer its *Subscriptions* to that *Session*. It may also be used by one *Client* to take over a *Subscription* from another *Client* by transferring the *Subscription* to its *Session*.

The *sessionId* contained in the request header identifies the *Session* to which the *Subscription* and *MonitoredItems* will be transferred. The *Server* must validate that the *Client* of that *Session* is operating on behalf of the same user and that the potentially new *Client* supports the *Profiles* that are necessary for the *Subscription*. If the *Server* transfers the *Subscription*, it returns the sequence numbers of the *NotificationMessages* that are available for retransmission. The *Client* should acknowledge all *Messages* in this list for which it will not request retransmission.

If the *Server* transfers the *Subscription* to the new *Session*, the *Server* must return the service result Good_SubscriptionTransfered with the *subscriptionId* and an empty *notificationMessage* for the next *Publish* request in the context of the old *Session*.

#### 5.13.7.2 Parameters

Table 119 defines the parameters for the *Service*.

**Table 119 – TransferSubscriptions Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionIds [] | IntegerId | List of identifiers for the *Subscriptions* to be transferred to the new *Client* (see Clause 7.9 for *IntegerId* definition). These identifiers are transferred from the primary *Client* to a backup *Client* via external mechanisms. |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 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 Clause 7.28 for *StatusCode* definition). |
| availableSequence NumbersRanges [] | NumericRange | 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 *NumericRange* type is defined in Clause 7.14. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information for the *Subscriptions* to transfer (see Clause 7.5 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. |

#### 5.13.7.3 Service results

Table 120 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 120 – TransferSubscriptions Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |
| Bad_UserAccessDenied | See Table 156 for the description of this result code.<br>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 121 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 121 – TransferSubscriptions Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

### 5.13.8 DeleteSubscriptions

#### 5.13.8.1 Description

This *Service* is invoked by the *Client* to delete one or more *Subscriptions* that it has created and that have not been transferred to another *Client* or that have been transferred to it.

Successful completion of this *Service* causes all *MonitoredItems* that use the *Subscription* to be deleted. If this is the last *Subscription* assigned to the *Client* issuing the request, then all *Publish* requests queued by that *Client* are dequeued and a negative response is returned for each.

#### 5.13.8.2 Parameters

Table 122 defines the parameters for the *Service*.

**Table 122 – DeleteSubscriptions Service Parameters**

| Name | Type | Description |
|---|---|---|
| **Request** | | |
| requestHeader | RequestHeader | Common request parameters (see Clause 7.19 for *RequestHeader* definition). |
| subscriptionIds [] | IntegerId | The *Server*-assigned identifier for the *Subscription* (see Clause 7.9 for *IntegerId* definition). |
| | | |
| **Response** | | |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |
| results [] | StatusCode | List of *StatusCodes* for the *Subscriptions* to delete (see Clause 7.28 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 Clause 7.5 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. |

#### 5.13.8.3 Service results

Table 123 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 156.

**Table 123 – DeleteSubscriptions Service Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_NothingToDo | See Table 156 for the description of this result code. |

#### 5.13.8.4 StatusCodes

Table 124 defines values for the *results* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 157.

**Table 124 – DeleteSubscriptions Operation Level Result Codes**

| Symbolic Id | Description |
|---|---|
| Bad_SubscriptionIdInvalid | See Table 156 for the description of this result code. |

# 6 Service behaviours

## 6.1 Security

### 6.1.1 Overview

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

### 6.1.2 Obtaining and Installing an Application Instance Certificate

All *UA Applications* require an application instance certificate which should contain the following information:

- The network name or address of the computer where the application runs;
- The name of the organisation that administers or owns the application;
- The name of the application;
- The name of the certificate authority that issued the certificate;
- The issue and expiry date for the certificate;
- The public key issued to the application by the certificate authority (CA);
- A digital signature created by the certificate authority (CA).

In addition, each application instance certificate has a private key which should be stored in a location that can only be accessed by the application. If this private key is compromised, the administrator must assign a new application instance certificate and private key to the application.

This certificate may be generated automatically when the application is installed. In this situation the private key assigned to the certificate must be used to create the certificate signature. Certificates created in this way are called self-signed certificates.

If the administrator responsible for the application decides that a self-signed certificate does not meet the security requirements of the organisation, then the administrator should install a certificate issued by a certification authority. The steps involved in requesting an application instance certificate from a certificate authority are shown in Figure 28.

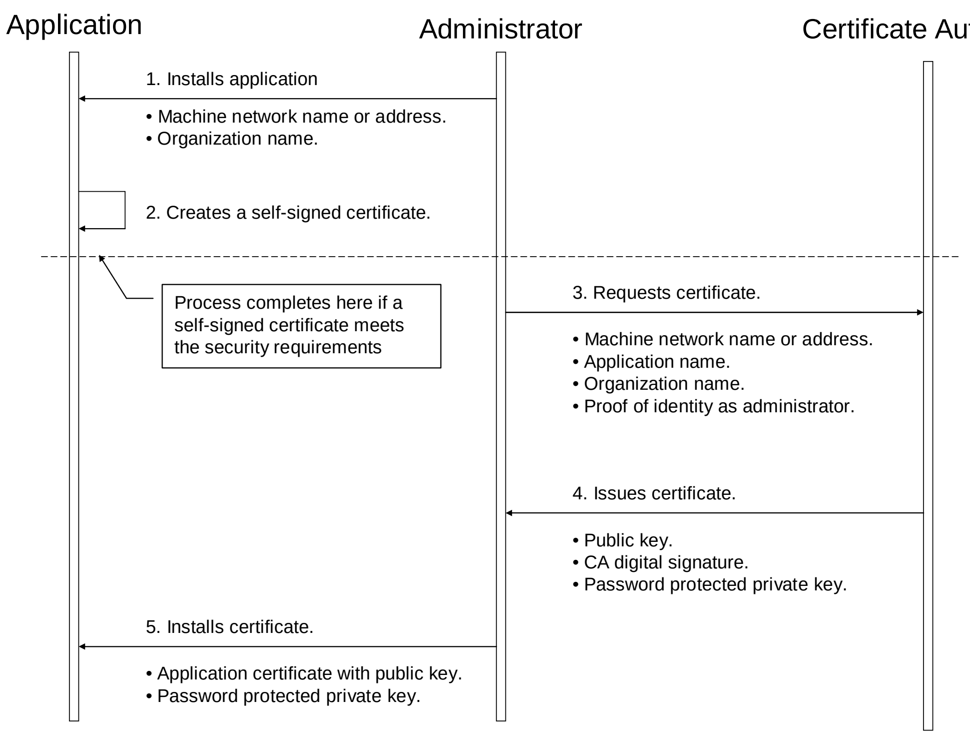
Application
Administrator
Certificate Authority
1. Installs application
• Machine network name or address.
• Organization name.
2. Creates a self-signed certificate.
Process completes here if a self-signed certificate meets the security requirements
3. Requests certificate.
• Machine network name or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

**Figure 28 – Obtaining and Installing an Application Instance Certificate**

The figure above illustrates the interactions between the *Application*, the *Administrator* and the *CertificateAuthority*. The *Application* is a *UA Application* installed on a single machine. The *Administrator* is the person responsible for managing the machine and the *UA Application*. The *CertificateAuthority* is an entity that can issue digital certificates that meet the requirements of the organisation deploying the *UA Application*.

If the *Administrator* decides that a self-signed certificate meets the security requirements for the organisation, then the *Administrator* may skip Steps 3 through 5. Application vendors must always create a default self-signed certificate during the installation process. Every *UA Application* must allow the *Administrators* to replace application instance certificates with certificates that meet their requirements.

When the *Administrator* requests a new certificate from a certificate authority, the certificate authority may require that the *Administrator* provide proof of authorization to request certificates for the organisation that will own the certificate. The exact mechanism used to provide this proof depends on the certificate authority.

Vendors may choose to automate the process of acquiring certificates from an authority. If this is the case, the *Administrator* would still go through the steps illustrated in the figure, however, the application install program would do them automatically and only prompt the *Administrator* to provide information about the application instance being installed.

### 6.1.3 Obtaining and Installing an Software Certificate

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

- The name of the organisation that developed the application;
- The name of the application;
- The software version and build number;
- A list of profiles supported by the application;
- The certification testing status for each supported profile;
- The name of the testing authority that issued the certificate;
- The issue and expiry date for the certificate;
- The public key issued to the application by the testing authority;
- A digital signature created by the testing authority.

The application vendor is responsible for completing the testing process and requesting software certificates from the testing authorities. The vendor should install the software certificates when an application is installed on machine. When distributing these certificates, the application vendors should take precautions to prevent unauthorized users from acquiring their software certificates and using them for applications that the vendor did not develop. Misused software certificates are not a security risk, but vendors could find that they are blamed for interoperability problems caused by use by unauthorized applications.

The steps involved in acquiring and installing a software certificate from a certificate authority are shown in Figure 29.

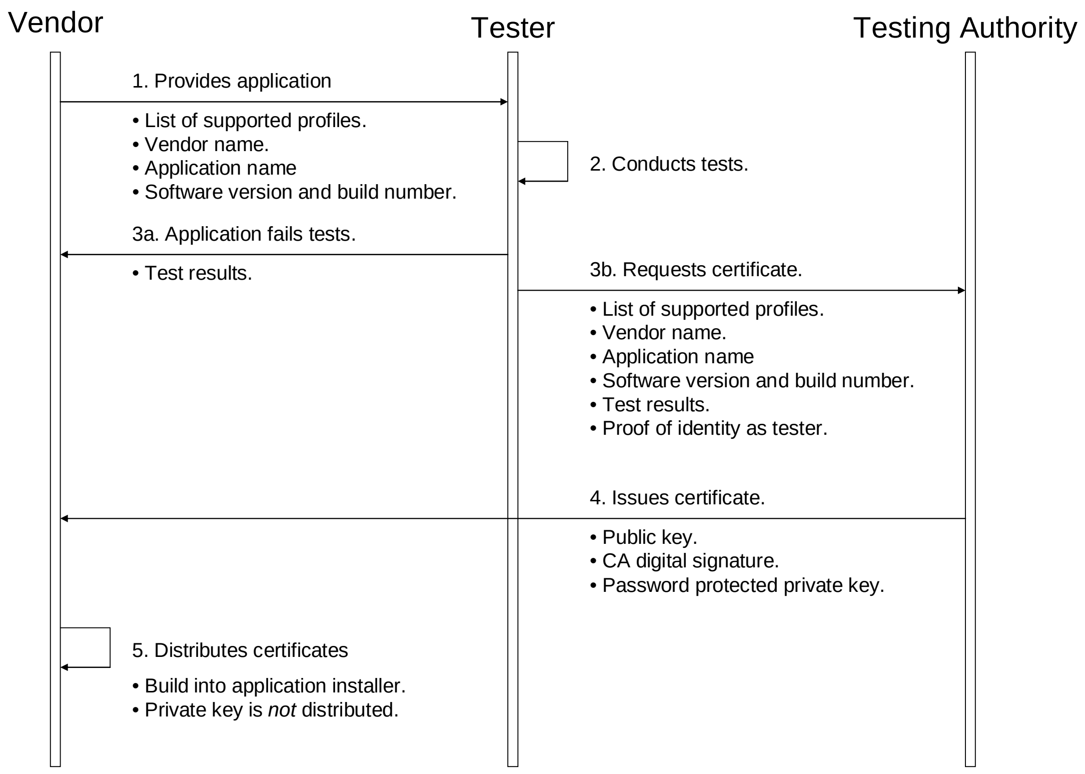
Vendor
Tester
Testing Authority
1. Provides application
• List of supported profiles.
• Vendor name.
• Application name
• Software version and build number.
2. Conducts tests.
3a. Application fails tests.
• Test results.
3b. Requests certificate.
• List of supported profiles.
• Vendor name.
• Application name
• Software version and build number.
• Test results.
• Proof of identity as tester.
4. Issues certificate.
• Public key.
• CA digital signature.
• Password protected private key.
5. Distributes certificates
• Build into application installer.
• Private key is *not* distributed.

**Figure 29 – Obtaining and Installing a Software Certificate**

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

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

### 6.1.4 Creating a SecureChannel

All *UA Applications* must establish a *SecureChannel* before creating a *Session*. This *SecureChannel* requires that both applications have access to certificates that can be used to encrypt and sign *Messages* exchange. The application instance certificates installed by following the process described in Clause 6.1.2 may be used for this purpose.

The steps involved in establishing a *SecureChannel* are shown in Figure 30.

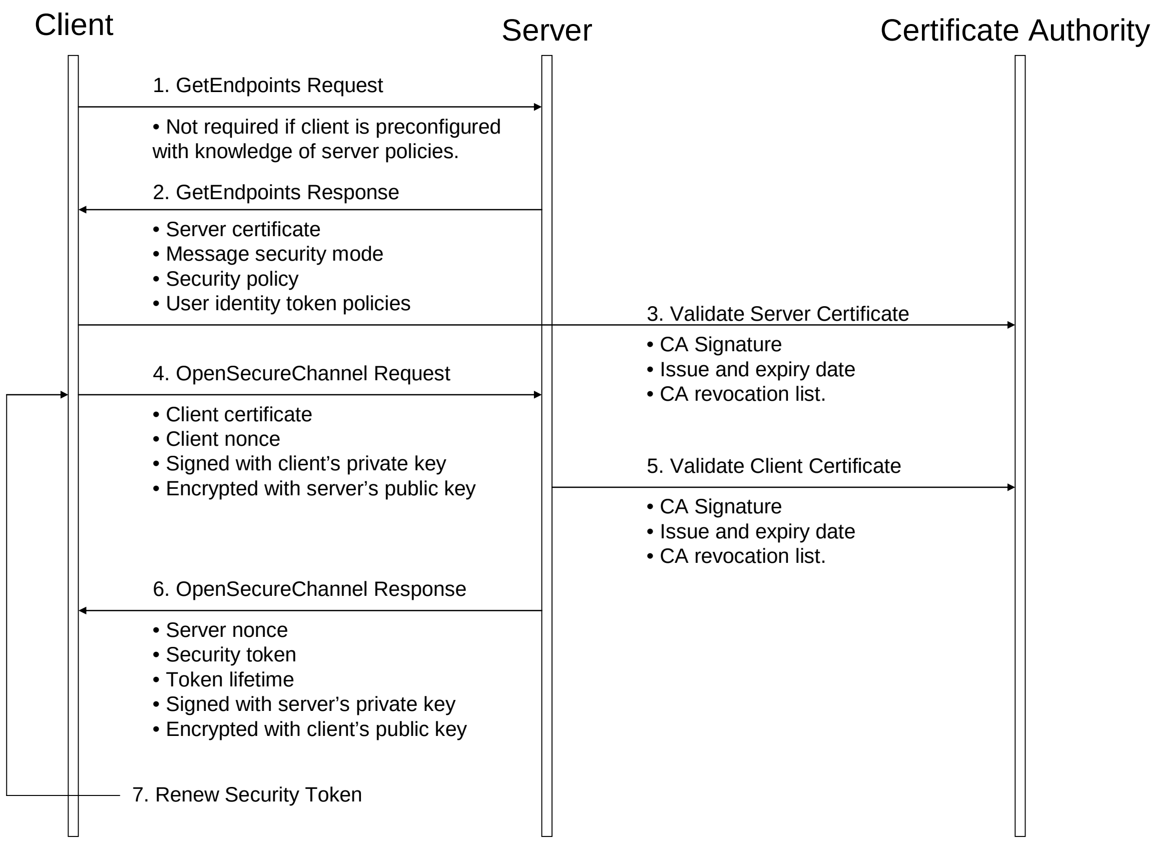
Client
Server
Certificate Authority
1. GetEndpoints Request
• Not required if client is preconfigured with knowledge of server policies.
2. GetEndpoints Response
• Server certificate
• Message security mode
• Security policy
• User identity token policies
3. Validate Server Certificate
• CA Signature
• 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 30 – Establishing a SecureChannel**

The figure above assumes *Client* and *Server* have online access to a certificate authority (CA). If online access is not available and if the administrator has installed the CA public key on the local machine, then the *Client* and *Server* must still validate the application certificates using that key. The figure shows only one CA, however, there is no requirement that the *Client* and *Server* certificates be issued by the same authority. A self-signed application instance certificate does not need to be verified with a CA.

Both the *Client* and *Server* will have a list of certificates that they have been configured to trust (sometimes called the Certificate Trust List or CTL). These trusted certificates may be certificates for certificate authorities or they may be *UA Application* instance certificates. *UA Applications* may be configured to reject connections with applications that do not have a trusted certificate.

Certificates can be compromised, which means they should no longer be trusted. Administrators can revoke a certificate by removing it from the trust list for all applications or they can add the certificate to the Certificate Revocation List (CRL) for the certificate authority that issued the certificate. Administrators may save a local copy of the CRL for each certificate authority when online access is not available.

A *Client* does not need to call *GetEndpoints* each time it connects to the *Server*. This information should change rarely and the *Client* can cache it locally. If the *Server* rejects the *OpenSecureChannel* request the *Client* should call *GetEndpoints* and make sure the *Server* configuration has not changed.

The exact mechanisms for using the security token to sign and encrypt *Messages* exchanged over the *SecureChannel* are described in [UA Part 6]. The process for renewing tokens is also described in detail in [UA Part 6].

In many cases, the certificates used to establish the *SecureChannel* will be the application instance certificates. However, some communication stacks might not support certificates that are specific to a single application. Instead, they expect all communication to be secured with a certificate specific to a user or the entire machine. For this reason, *UA Applications* will need to exchange their application instance certificates when creating a *Session*.

### 6.1.5 Creating a Session

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

The steps involved in establishing a *Session* are shown in Figure 31.

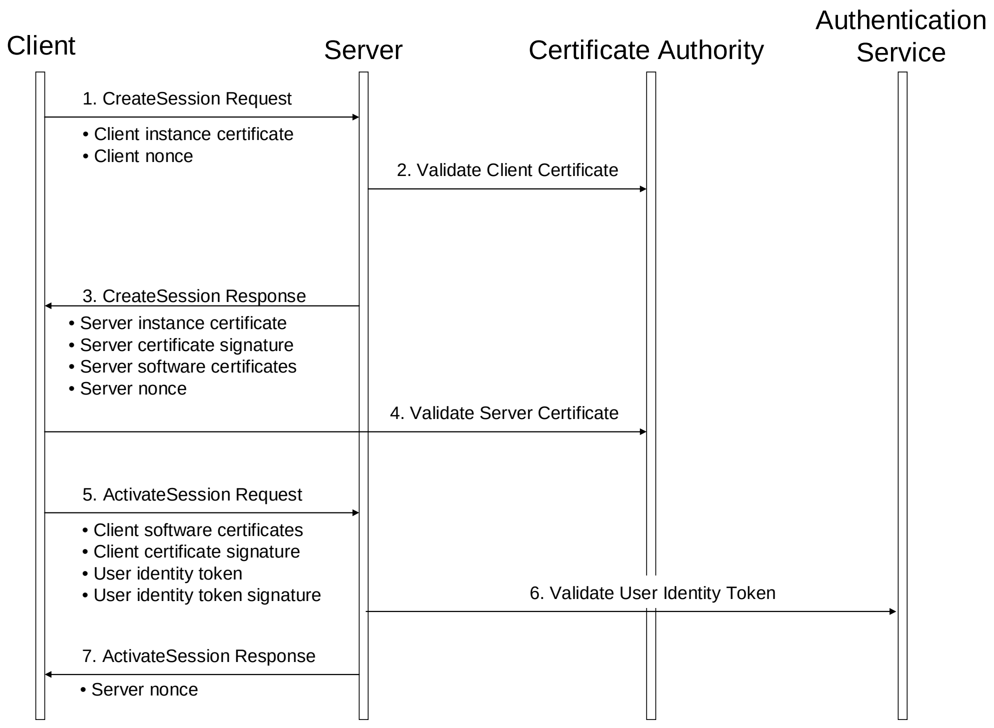
Client
Server
Certificate Authority
Authentication Service
1. CreateSession Request
• Client instance certificate
• Client nonce
2. Validate Client Certificate
3. CreateSession Response
• Server instance certificate
• Server certificate signature
• Server software certificates
• Server nonce
4. Validate 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 31 – Establishing a Session**

The figure above illustrates the interactions between a *Client*, a *Server*, a certificate authority (CA) and an authentication service. The CA is responsible for issuing the application certificates. If the *Client* or *Server* do not have online access to the CA, then they must validate the application certificates using the CA public key that the administrator must install on the local machine.

The authentication service is a central database that can verify that user token provided by the *Client*. This authentication service may also tell the *Server* what access rights the user has. The authentication service depends on the user identity token. It could be a certificate authority, a Kerberos ticket granting service, a WS-Trust *Server* or a proprietary database of some sort.

The *Client* and *Server* must prove possession of their application certificates by signing the certificates with a nonce appended. The exact mechanism used to create the proof of possession signatures is described in [UA Part 6]. Similarly, the *Client* must prove possession of some types of user identity tokens by creating signatures with the secret associated with the token.

The application instance certificates exchanged while creating a *Session* may be the same certificates that are exchanged when the *SecureChannel* was opened. If this is the case, the applications do not need to re-validate the certificate. UA requires that the application instance certificates be provided twice because the UA application might not be able to discover what certificates were used to establish the *SecureChannel* and the UA application may need these certificates to apply access restrictions that are based on an application instance rather than a specific user.

### 6.1.6 Impersonating a User

Once a UA *Client* has established a *Session* with a *Server* it can change the user identity associated with the *Session* by calling the *ActivateSession* service.

The steps involved in impersonating a user are shown in Figure 32.

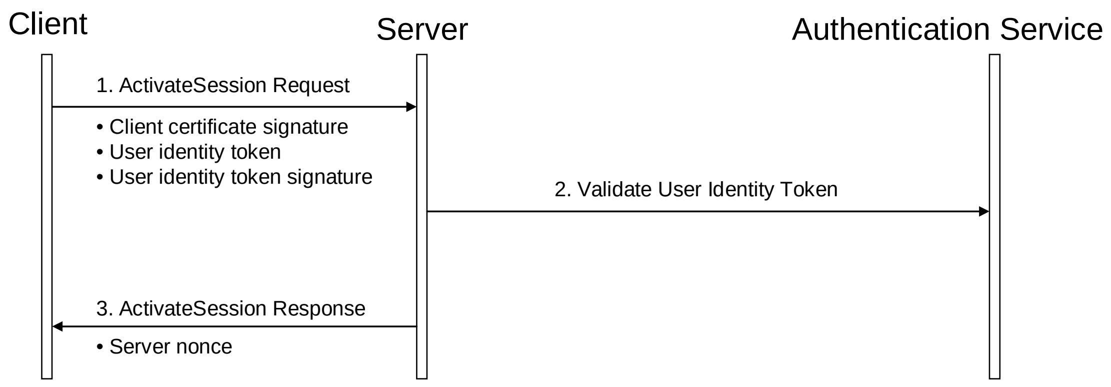
Client
Server
Authentication Service
1. ActivateSession Request
• Client certificate signature
• User identity token
• User identity token signature
2. Validate User Identity Token
3. ActivateSession Response
• Server nonce

**Figure 32 – Impersonating a User**

## 6.2 UA Auditing

### 6.2.1 Overview

Auditing is a requirement in many systems. It provides a means of tracking activities that occur as part of normal operation of the system. It also provides a means of tracking abnormal behaviour. It is also a requirement from a security standpoint. For more information on the security aspects of auditing see [UA Part 2]. This section describes what is expected of a UA *Server* and *Client* with respect to auditing and it details the audit recommendation for each service set.

### 6.2.2 General audit logs

Each UA service request contains a string parameter that is used to carry an audit record id. A *Client* or any *Server* operating as a *Client*, such as an aggregating *Server*, can create a local audit log entry for a request that it submits. This parameter allows this *Client* to pass the identifier for this entry with the request. If this *Server* also maintains an audit log, it should include this id in its audit log entry that it writes. When this log is examined and that entry is found, the examiner will be able to relate it directly to the audit log entry created by the *Client*. This capability allows for traceability across audit logs within a system.

### 6.2.3 General audit Events

A *Server* that maintains an audit log must provide the audit log entries via standard *Event Messages*. The *AuditEventType* and its sub-types are defined in [UA Part 3]. An audit *Event Message* also includes the audit record Id. The details of the *AuditEventType* and its sub types are defined in [UA Part 5]. A *Server* that is an aggregating *Server* that supports auditing must also subscribe for audit events for all of the *Servers* that it is aggregating (assuming they provide auditing). The combined stream should be available from the aggregating *Server*.

### 6.2.4 Auditing for SecureChannel Service Set

All *Services* in this *Service Set* for *Servers* that support auditing must generate audit entries and audit *Events* for failed service invocations and for successful invocation of the *OpenSecureChannel* and *CloseSecureChannel* Services. The audit entries should be setup prior to the actual call, allowing the correct audit record Id to be provided. The *OpenSecureChannel* Service must generate an audit *Event* of type *AuditOpenSecureChannelEventType* or a subtype of it. The *CloseSecureChannel* service must generate an audit *Event* of type *AuditCloseSecureChannelEventType* or a subtype of it. Both of these *Event* types are sub-types of the *AuditChannelEventType*. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the service failed. The additional parameters should include the details of the request. It is understood that these events may be generated by the underlining stack in many cases, but they must be made available to the *Server* and the *Server* must report them.

### 6.2.5 Auditing for Session Service Set

All *Services* in this *Service Set* for *Servers* that support auditing must generate audit entries and audit *Events* for both successful and failed *Service* invocations. These *Services* must generate an audit *Event* of type *AuditSessionEventType* or a sub-type of it. In particular, they must generate the base *EventType* or the appropriate sub-type, depending on the service that was invoked. The *CreateSession* service must generate *AuditCreateSessionEventType* events or sub-types of it. The *ActivateSession* service must generate *AuditActivateSessionType* events or sub-types of it. When the *ActivateSession Service* is called to change the user identity then the server must generate *AuditImpersonateUserEventType* events or sub-types of it. The CloseSession service must generate the base *EventType* of *AuditSessionEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the *Service* failed. The additional parameters should include the details of the request.

For *Clients*, that support auditing, accessing the services in the *Session Service Set* must generate audit entries for both successful and failed invocations of the *Service*. These audit entries should be setup prior to the actual *Service* invocation, allowing the invocation to contain the correct audit record id.

### 6.2.6 Auditing for NodeManagement Service Set

All Services in this *Service Set* for *Servers* that support auditing must generate audit entries and audit *Events* for both successful and failed *Service* invocations. These *Services* must generate an audit *Event* of type *AuditNodeManagementEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for *Events* of this type should include a description of why the service failed. The additional parameters should include the details of the request.

For *Clients* that support auditing, accessing the *Services* in the *NodeManagement Service Set* must generate audit entries for both successful and failed invocations of the *Service*. All audit entries should be setup prior to the actual *Service* invocation, allowing the invocation to contain the correct audit record id.

### 6.2.7 Auditing for Attribute Service Set

The *Write* or *HistoryUpdate* Services in this *Service Set* for *Servers* that support auditing must generate audit entries and audit *Events* for both successful and failed *Service* invocations. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the *Service* failed. The additional parameters should include the details of the request.

The *Read* and *HistoryRead Services* may generate audit entries and audit *Events* for failed *Service* invocations. These *Services* should generate an audit *Event* of type *AuditEventType*. The *SourceNode* for *Events* of this type should be assigned to the *NodeId of the Node* that reports the error. The *SourceName* for *Events* of this type should be “Attribute/” and the service that generates the event (*Read*, *HistoryRead*). The *Message* for *Events* of this type should include a description of why the *Service* failed.

For *Clients* that support auditing, accessing the *Write* or *HistoryUpdate* services in the *Attribute Service Set* must generate audit entries for both successful and failed invocations of the *Service*. Invocations of the other *Services* in this *Service Set* may generate audit entries. All audit entries should be setup prior to the actual *Service* invocation, allowing the invocation to contain the correct audit record id.

### 6.2.8 Auditing for Method Service Set

All *Services* in this *Service Set* for *Servers* that support auditing must generate audit entries and audit *Events* for both successful and failed service invocations if the invocation modifies the address space, writes a value or modifies the state of the system (alarm acknowledge, batch sequencing or other system changes). Methods that do not modify the address space, write values or modify the state of the system may generate events. These method calls should generate the most appropriate *EventType* for the call. The *SourceNode* for Events of this type should be the *Session* that the call was issued on. The *SourceName* for *Events* of this type should be “Call/” the *Method* name that was invoked.

For *Clients* that support auditing, accessing the *Method Service Set* must generate audit entries for both successful and failed invocations of the *Service*, if the invocation modifies the address space, writes a value or modifies the state of the system (alarm acknowledge, batch sequencing or other system changes). Invocations of the other *Methods* may generate audit entries. All audit entries should be setup prior to the actual *Service* invocation, allowing the invocation to contain the correct audit record id.

### 6.2.9 Auditing for View, Query, MonitoredItem and Subscription Service Set

All of the *Services* in these four *Service Sets* only provide the *Client* with information, with the exception of the *TransferSubscriptions Service* in the *Subscription Service Set*. In general, these services will not generate audit entries or audit Event *Messages*. The *TransferSubscriptions Service* must generate an audit *Event* of type *AuditSessionEventType* or sub-types of it. See [UA Part 5] for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for Events of this type should include a description of why the service failed.

For *Clients* that support auditing, accessing the *TransferSubscriptions Service* in the *Subscription Service Set* must generate audit entries for both successful and failed invocations of the *Service*. Invocations of the other *Services* in this *Service Set* do not require audit entries. All audit entries should be setup prior to the actual *Service* invocation, allowing the invocation to contain the correct audit record id.

## 6.3 Redundancy

### 6.3.1 Redundancy overview

Redundancy in OPC UA ensures that both Clients and Server can be redundant. OPC UA does not provide redundancy, it provides the data structures and services by which redundancy may be achieved in a standard manner.

### 6.3.2 Server redundancy overview

Server redundancy comes in two modes, transparent and non-transparent. By definition, in transparent redundancy the failover of *Server* responsibilities from one *Server* to another is transparent to the *Client*: the *Client* does care or even know that failover has occurred; the *Client* does not need to do anything at all to keep data flowing. In contrast, non-transparent failover requires some activity on the part of the *Client*.

The two areas where redundancy creates specific needs are in keeping the *Server* and *Client* information synchronised across *Servers*, and in controlling the failover of data flow from one *Server* to another.

#### 6.3.2.1 Transparent redundancy

For transparent redundancy, all OPC UA provides is the data structures to allow the *Client* to identify what *Servers* are available in the redundant set, what the service level of each *Server* is and which *Server* is currently supporting a specified *Session*. All OPC UA interactions within a given session will be supported by one *Server* and the *Client* is able to identify which *Server* that is, allowing a complete audit trail for the data. It is the responsibility of the *Servers* to ensure that information is synchronised between the *Servers* and to effect the switching of the address from one *Server* to another upon failover.

Figure 33 shows a typical transparent redundancy setup.

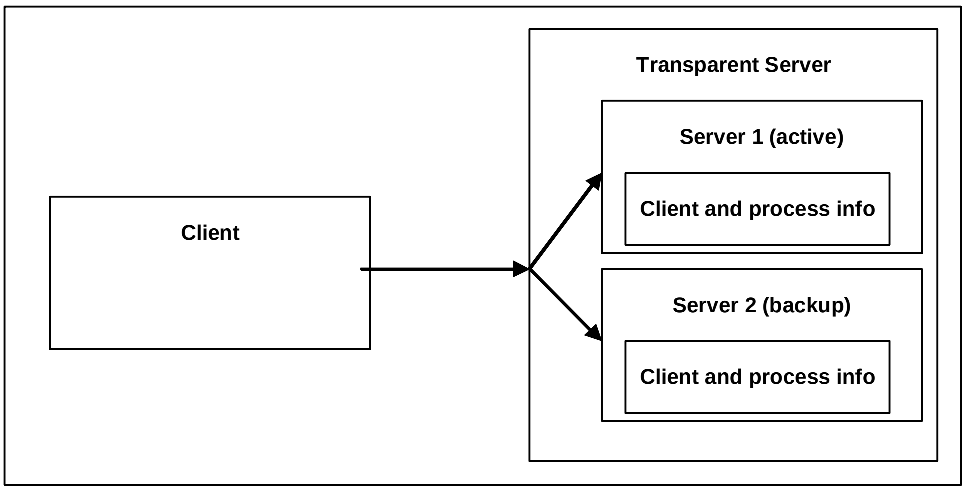

**Figure 33 – Transparent Redundancy setup**

#### 6.3.2.2 Non-transparent redundancy

For non-transparent redundancy, OPC UA provides the same data structures and also *Server* information which tells the *Client* what modes of failover the *Server* supports. This information allows the *Client* to determine what actions it may need to take in order to accomplish failover.

Figure 34 shows a typical non-transparent redundancy setup.

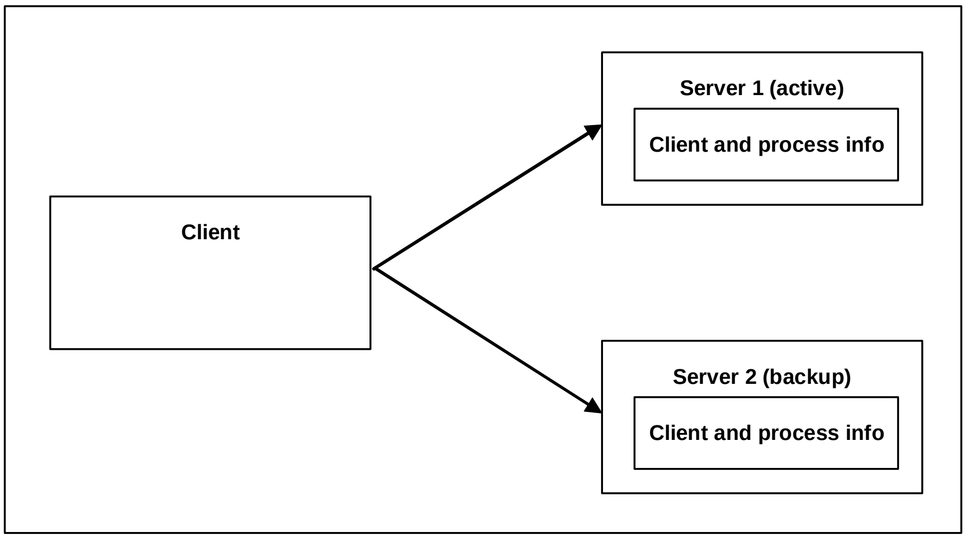

**Figure 34 – Non-Transparent Redundancy setup**

For non-transparent redundancy the *Server* has additional concepts of cold, warm and hot failover. Cold failovers are for *Servers* where only one *Server* can be active at a time. Warm failovers are for *Servers* where the backup *Servers* can be active, but cannot connect to actual data points (typically a system where the underlying devices are limited to a single connection). Hot failovers are for *Servers* where more than one *Server* can be active and fully operational

Table 125 defines the list of failover actions.

**Table 125 – Redundancy failover actions**

| **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 must be designed into the *Client* and to enable simpler *Clients* to take advantage of non-transparent redundancy. By using the *TransferSubscriptions Service*, which allows a *Client* to request that a set of *Subscriptions* be moved from one *Session* to another, a *Server* vendor can effectively make transparent failover a part of a proxy stub that lives on the *Client*. There are two ways to do this, one requiring code in the *Server* to support this and the other doing it all from the *Client* proxy process.

When the *Client* proxy is used, the proxy simply duplicates *Subscriptions* and modifications to *Subscriptions*, by passing the calls on to both *Servers*, but only enabling publishing or sampling on one *Server*. When the proxy detects a failure, it enables publishing and/or sampling on the backup *Server*, just as the *Client* would if it were a redundancy-aware *Client*.

The other method also requires a *Client* stub, but in this case the stub is a much lighter-weight process. In this mode, it is the *Server* which mirrors all *Subscriptions* in the other *Server*, but the *Client* endpoint for these *Subscriptions* is the active *Server*. When the stub detects that the active *Server* has failed, it issues a *TransferSubscriptions* call to the backup *Server*, moving the *Subscriptions* from the *Session* owned by the failed *Server* to its own *Session*, and activating publishing.

Figure 35 shows the difference between *Client* proxy and *Server* proxy redundancy.

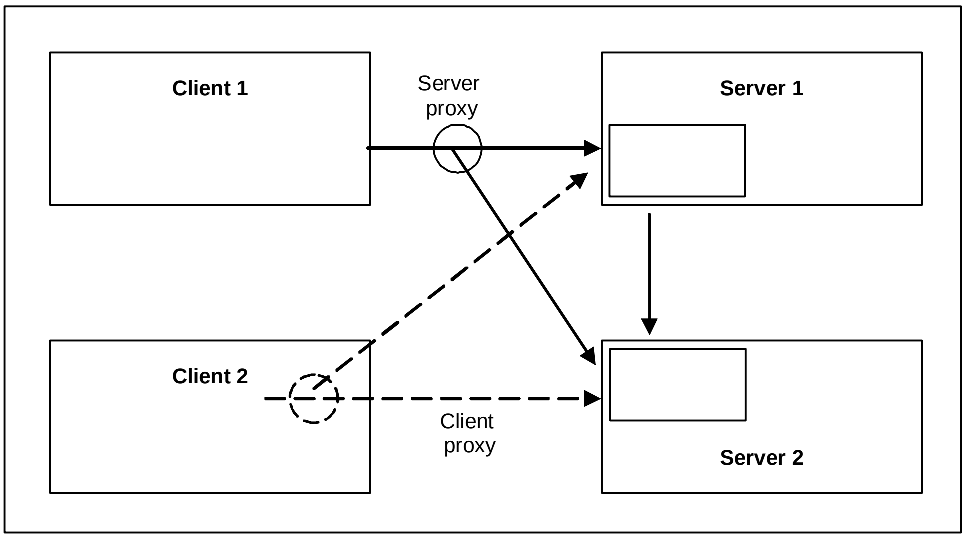

**Figure 35 – Redundancy mode**

### 6.3.3 Client redundancy

*Client* redundancy is supported in OPC UA by the *TransferSubscriptions* call and by exposing *Client* information in the *Server* information structures. Since *Subscription* lifetime is not tied to the *Session* in which it was created, backup *Clients* can monitor the active *Client's Session* with the *Server*, just as they would monitor any other data variable. If the active *Client* ceases to be active, the *Server* will send a data update to any *Client* which has that variable monitored. Upon receiving such notification, a backup *Client* would then instruct the *Server* to transfer the *Subscriptions* to its own session. If the *Subscription* is crafted carefully, with sufficient resources to buffer data during the change-over, there need be no data loss from a *Client* failover.

OPC UA does not provide a standardized mechanism for conveying the *SessionId* and *SubscriptionIds* from the active *Client* to the backup *Clients*, but as long as the backup *Clients* know the *Client* name of the active *Client*, this information is readily available using the *SessionDiagnostics* and *SubscriptionDiagnostics* portions of the *ServerDiagnostics* data.

# 7 Common parameter type definitions

## 7.1 BuildInfo

The components of this data type are defined in Table 126.

**Table 126 – BuildInfo Structure**

| Name | Type | Description |
|---|---|---|
| BuildInfo | structure | Information that describes the build of the software. |
| applicationUri | String | URI that identifies the software |
| manufacturerName | String | Name of the software manufacturer. |
| applicationName | String | Name of the software. |
| softwareVersion | String | Software version |
| buildNumber | String | Build number |
| buildDate | UtcTime | Data and time of the build. |

## 7.2 ContentFilter

The *ContentFilter* structure defines a collection of elements that make up a filtering criteria. Each element in the collection describes an operator and an array of operands to be used by the operator. The operators that can be used in a ContentFilter are described in Table 127. The filter is evaluated by evaluating the first entry in the element array starting with the first operand in the operand array. The operands of an element may contain *References* to sub-elements resulting in the evaluation continuing to the referenced elements in the element array.

Table 127 defines the ContentFilter structure.

**Table 127 – ContentFilter Structure**

| Name | Type | Description |
|---|---|---|
| ContentFilter | structure | |
| elements [] | ContentFilterElement | List of operators and their operands that compose the filter criteria. The filter is evaluated by starting with the first entry in this array. |
| filterOperator | enum FilterOperator | Filter operator to be evaluated.<br>The *FilterOperator* enumeration is defined in Table 128. |
| filterOperands [] | Extensible Parameter FilterOperand | Operands used by the selected operator. The number and use depend on the operands defined in Table 128. This array needs at least one entry.<br>This extensible parameter type is the *FilterOperand* parameter type specified in Clause 8.4. It specifies the list of valid *FilterOperand* values. |

Table 128 defines the valid operators that can be used in a ContentFilter.

**Table 128 –FilterOperator Definition**

| Operator | Number of Operands | Description |
|---|---|---|
| Equals | 2 | TRUE if operand[0] is equal to operand[1]. |
| IsNull | 1 | TRUE if operand[0] is a null value. |
| GreaterThan | 2 | TRUE if operand[0] is greater than operand[1]. |
| LessThan | 2 | TRUE if operand[0] is less than operand[1]. |
| GreaterThanOrEqual | 2 | TRUE if operand[0] is greater than or equal to operand[1]. |
| LessThanOrEqual | 2 | TRUE if operand[0] is less than or equal to operand[1]. |
| Like | 2 | TRUE if operand[0] matches a pattern defined by operand[1]. See Table 129 for the definition of the pattern syntax. |
| Not | 1 | TRUE if operand[0] is FALSE. |
| Between | 3 | TRUE if operand[0] is greater or equal to operand[1] and less than or equal to operand[2]. |
| InList | 2..n | TRUE if operand[0] is equal to one or more of the remaining operands. |
| And | 2 | TRUE if operand[0] and operand[1] are TRUE. |
| Or | 2 | TRUE if operand[0] or operand[1] are TRUE. |
| InView | 1 | TRUE if contained in the *View* defined by operand[0]. Operand[0] should be of an AttributeOperand type where the attribute is the NodeId |
| OfType | 1 | TRUE if of type operand[0] or of a subtype of operand[0]. Operand[0] should be of an AttributeOperand type where the attribute is the NodeId |
| RelatedTo | 4 | TRUE if the *Object* 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 joining and chaining of relationships. In this case, the referred to element returns a list of *NodeIds* instead of TRUE or FALSE. Operand[3] defines the number of hops the relationship should be followed. If Operand[3] is 1, then objects must be directly related. If a hop is greater than 1, then a *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[0], [1],[2] should be of an AttributeOperand type where the attribute is the NodeId. |

The *Like* operator can be used to perform wildcard comparisons. Several special characters can be included in the second operand of the *Like* operator. The valid characters are defined in Table 129.

The RelatedTo operator can be used to identify if a given type, set as operand[1], is a subtype of another type set as operand[0] by setting operand[2] to the *HasSubtype ReferenceType* and operand[3] to 0.

**Table 129 – Wildcard characters**

| Special Character | Description |
|---|---|
| % | Match any character or group of characters |
| _ | Match any single character |
| \ | Escape character allows literal interpretation (i.e. \\ is \, \% is %, _ is _) |
| [] | Match any single character in a list (i.e. [1,3-6,8] would match 1,3,4,5,6, and 8 |
| [!] | Not Matching any single character in a list (i.e. [!1,3-5] would not match 1,3,4, and 5) |

For example the logic describe by '(((AType.A = 5) or InList(BType.B, 3,5,7)) and CType.displayName LIKE "Main%")' would result in a logic tree as shown in Figure 36 and a ContentFilter as shown in Table 130.

**Ex: (((AType.A = 5) or Inlist(BType.B,3,5,7)) and CType.displayName LIKE “Main%”)**

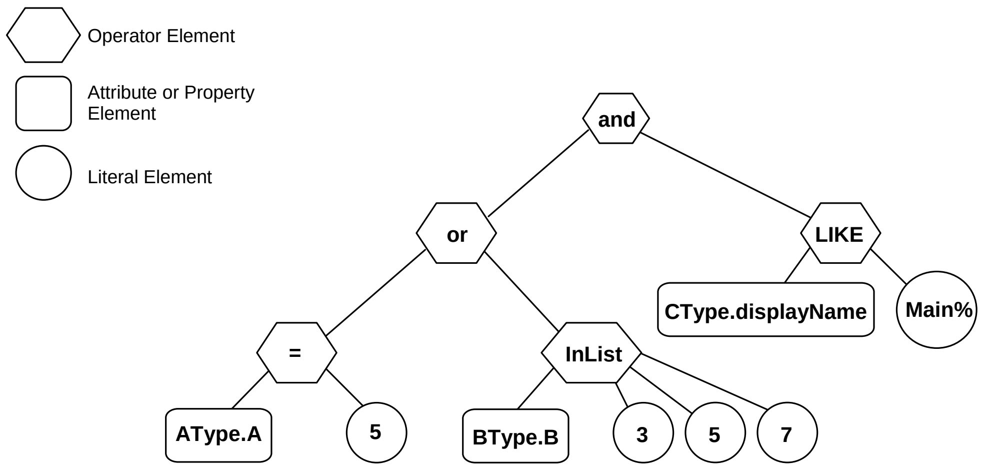

**Figure 36 – Filter Logic Tree Example**

Table 130 describes the elements, operators and operands used in the example.

**Table 130 – ContentFilter Example**

| Element[] | Operator | Operand[0] | Operand[1] | Operand[2] | Operand[3] |
|---|---|---|---|---|---|
| 0 | And | ElementOperand = 1 | Element Operand = 4 | | |
| 1 | Or | ElementOperand = 2 | Element Operand = 3 | | |
| 2 | Equals | PropertyOperand = NodeId: AType, Property: A | LiteralOperand = ‘5’ | | |
| 3 | InList | PropertyOperand = NodeId: BType, Property: B | LiteralOperand = ‘3’ | LiteralOperand = ‘5’ | LiteralOperand = ‘7’ |
| 4 | Like | AttributeOperand = NodeId: CType, AttributeId: displayName | LiteralOperand = “Main%” | | |

As another example a filter to select all *SystemEvents* (including derived types) that are contained in the Area1 *View* or the Area2 *View* would result in a logic tree as shown in Figure 37 and a ContentFilter as shown in Table 131.

**Ex: (InView(Area1) or InView(Area2)) and OfType(SytemEventType)**

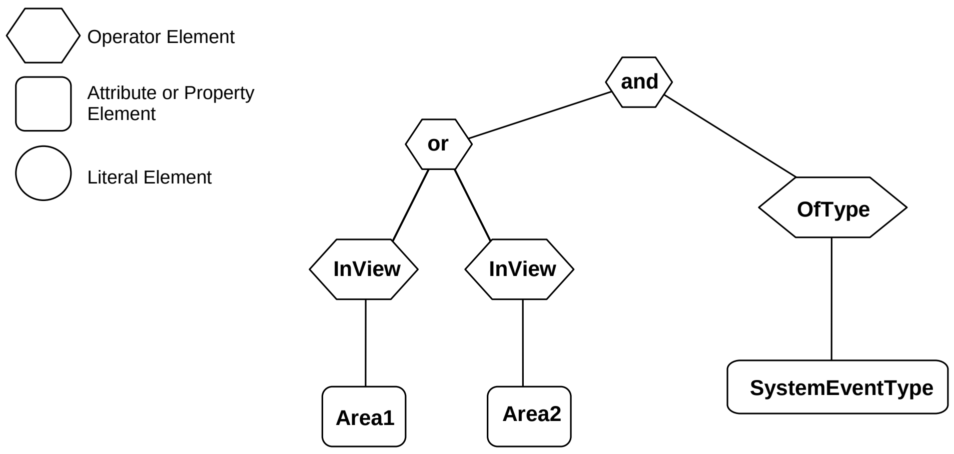

**Figure 37 – Filter Logic Tree Example**

Table 131 describes the elements, operators and operands used in the example.

**Table 131 – ContentFilter Example**

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

Table 132 defines the casting rules for the operand values. The source types in the table are automatically converted to the target types listed in the table. The types used in the table are defined in [UA Part 3]. An overflow during conversion should result in a bad *StatusCode*.

**Table 132 – Casting rules**

| Source Type | Target Type |
|---|---|
| String | SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Guid, XmlElement |
| Double | Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| Float | Double, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| SByte | String, Double, Float, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| Int16 | String, Double, Float, SByte, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| Int32 | String, Double, Float, SByte, Int16, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| Int64 | String, Double, Float, SByte, Int16, Int32, Byte, UInt16, UInt32, UInt64, Boolean |
| Byte | String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64, Boolean |
| UInt16 | String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt32, UInt64, Boolean |
| UInt32 | String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt64, Boolean |
| UInt64 | String, Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, Boolean |
| Boolean | Double, Float, SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, UInt64 |
| Guid | String |

## 7.3 Counter

This primitive data type is a UInt32 that represents the value of a counter. The initial value of a counter is specified by its use. Modulus arithmetic is used for all calculations, where the modulus is max value + 1. Therefore,

x + y = (x + y)mod(max value + 1)

For example:

max value + 1 = 0

max value + 2 = 1

## 7.4 DataValue

The components of this parameter are defined in Table 133.

**Table 133 – DataValue**

| Name | Type | Description |
|---|---|---|
| DataValue | structure | The value and associated information. |
| value | BaseDataType | The data value. |
| statusCode | StatusCode | The *StatusCode* that defines with the *Server*'s ability to access/provide the value. The *StatusCode* type is defined in Clause 7.28 |
| sourceTimestamp | UtcTime | The source timestamp for the value. |
| serverTimestamp | UtcTime | The *Server* timestamp for the value. |

<u>SourceTimestamp</u>

The *sourceTimestamp* is used to reflect the timestamp that was applied to a *Variable* value by the data source. Once a value has been assigned a source timestamp, the source timestamp for that value instance never changes. In this context, “value instance” refers to the value received, independent of its actual value.

The *sourceTimestamp* must be UTC time and should indicate the time of the last change of the *value* or *statusCode*.

The *sourceTimestamp* should be generated as close as possible to the source of the value but the timestamp needs to be set always by the same physical clock. In the case of redundant sources, the clocks of the sources should be synchronised.

If the OPC UA *Server* receives the *Variable* value from another OPC UA *Server*, then the OPC UA *Server* must always pass the source timestamp without changes. If the source that applies the timestamp is not available, the source time stamp is set to null. For example if a value could not be read because of some error during processing like invalid arguments passed in the request then the sourceTimestamp must be null.

In the case of a bad or uncertain status *sourceTimestamp* is used to reflect the time that the source recognized the non-good status or the time the *Server* last tried to recover from the bad or uncertain status.

The *sourceTimestamp* is only returned with a *Value Attribute*. For all other *Attributes* the returned *sourceTimestamp* is set to null.

<u>ServerTimestamp</u>

The *serverTimestamp* is used to reflect the time that the *Server* received a *Variable* value or knew it to be accurate.

In the case of a bad or uncertain status, *serverTimestamp* is used to reflect the time that the *Server* received the status or that the *Server* last tried to recover from the bad or uncertain status.

In the case where the OPC UA *Server* subscribes to a value from another OPC UA *Server*, each *Server* applies its own *serverTimestamp*. This is in contrast to the *sourceTimestamp* in which only the originator of the data is allowed to apply the *sourceTimestamp*.

If the *Server* is subscribing to the value from another *Server* every ten seconds and the value changes, then the *serverTimestamp* is updated each time a new value is received. If the value does not change, then new values will not be received on the *Subscription*. However, in the absence of errors, the receiving *Server* applies a new *serverTimestamp* every ten seconds because not receiving a value means that the value has not changed. Thus, the *serverTimestamp* reflects the time at which the *Server* knew the value to be accurate.

This concept also applies to OPC UA *Servers* that receive values from exception-based data sources. For example, suppose that a *Server* is receiving values from an exception-based device, and that

a) the device is checking values every 0.5 second,

b) the connection to the device is good,

c) the device sent an update three minutes ago with a value of 1.234.

In this case, the *Server* value would be 1.234 and the *serverTimestamp* would be updated every 0.5 seconds after the receipt of the value.

*StatusCode* assigned to a value

The *StatusCode* is used to indicate the conditions under which a *Variable* value was generated, and thereby can be used as an indicator of the usability of the value. The *StatusCode* is defined in Cause 7.28.

Overall condition (severity):

- A *StatusCode* with severity `Good` means that the value is of good quality.
- A *StatusCode* with severity `Uncertain` means that the quality of the value is uncertain for reasons indicated by the Substatus.
- A *StatusCode* with severity `Bad` means that the value is not usable for reasons indicated by the Substatus.

Rules:

- The *StatusCode* indicates the usability of the value. Therefore, It is required that *Clients* minimally check the *StatusCode Severity* of all results - even if they do not check the other fields - before accessing and using the value.
- A *Server*, which does not support status information, must return a severity code of `Good`. It is also acceptable for a *Server* to simply return a severity and a non-specific (0) Substatus.
- If the *Server* has no known value - in particular when *Severity* is BAD - it must return a NULL value.

## 7.5 DiagnosticInfo

The components of this parameter are defined in Table 134.

**Table 134 – DiagnosticInfo**

| Name | Type | Description |
|---|---|---|
| DiagnosticInfo | structure | Vendor-specific diagnostic information. |
| identifier | structure | The vendor-specific identifier of an error or condition. |
| namespaceIndex | Index | 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 Clause 7.20. The *namespaceIndex* parameter contains the index into the *stringTable* for this string. The index type is defined in Clause 7.8. |
| symbolicIdIndex | Index | A vendor-specific symbolic identifier string identifies an error or condition. The maximum length of this string is 32 characters. *Servers* wishing to return a numeric return code should convert the return code into a string and return the string in this identifier.<br>This symbolic identifier string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in Clause 7.20. The *symbolicIdIndex* parameter contains the index into the *stringTable* for this string. The index type is defined in Clause 7.8. |
| localizedTextIndex | Index | A vendor-specific localized text string describes the symbolic id. The maximum length of this text string is 256 characters.<br>This localized text string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in Clause 7.20. The *localizedTextIndex* parameter contains the index into the *stringTable* for this string. The index type is defined in Clause 7.8. |
| additionalInfo | String | Vendor-specific diagnostic information. |
| innerStatusCode | StatusCode | The *StatusCode* from the inner operation.<br>Many applications will make calls into underlying systems during UA request processing. A UA *Server* has the option of reporting the status from the underlying system in the diagnostic info. |
| innerDiagnosticInfo | DiagnosticInfo | The diagnostic info associated with the inner *StatusCode*. |

## 7.6 Duration

This primitive data type is an Int32 that defines an interval of time in milliseconds. Negative values are generally invalid but may have special meanings for the parameter where the Duration is used.

## 7.7 ExpandedNodeId

The components of this parameter are defined in Table 135. *ExpandedNodeIds* allow the namespace to be specified explicitly as a string or with an index in the *Server*'s namespace table.

**Table 135 – ExpandedNodeId**

| Name | Type | Description |
|---|---|---|
| ExpandedNodeId | structure | The *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*.<br>This index is the index of that *Server* in the local *Server*'s *Server* table. The index of the local *Server* in the *Server* table is always 0. All remote *Servers* have indexes greater than 0. The *Server* table is contained in the *Server Object* in the *AddressSpace* (see [UA Part 3] and [UA Part 5]).<br>The *Client* may read the *Server* table *Variable* to access the description of the target *Server* |
| namespaceUri | String | The URI of the namespace.<br>If this parameter is specified then the namespace index is ignored.<br>[UA Part 6] describes discovery mechanism that can be used to resolve URIs into URLs. |
| namespaceIndex | Index | The index in the *Server*'s namespace table.<br>This parameter must 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 a UA *Server*. (see *NodeId* definition in [UA Part 3]). |

## 7.8 Index

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

## 7.9 IntegerId

This primitive data type is an UInt32 that is used as an identifier, such as a handle. All values, except for 0, are valid.

## 7.10 MessageSecurityMode

The *MessageSecurityMode* is an enumeration that specifies what security should be applied to messages exchanges during a Session. The possible values are described in Table 136.

**Table 136 – MessageSecurityMode Values**

| Value | Description |
|---|---|
| NONE_0 | No security is applied. |
| SIGN_1 | All messages are signed but not encrypted. |
| SIGNANDENCYPT_2 | All messages are signed and encrypted. |

## 7.11 MonitoringAttributes

The components of this parameter are defined in Table 137.

**Table 137 – MonitoringAttributes**

| Name | Type | Description |
|---|---|---|
| MonitoringAttributes | structure | *Attributes* 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 Clause 7.9. |
| 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.<br>The value 0 indicates that the *Server* should use the fastest practical rate.<br>The value -1 indicates that the default sampling interval defined by the publishing rate of the *Subscription* is used.<br>The *Server* uses this parameter to assign the *MonitoredItems* to a sampling interval that it supports. The Duration type is defined in Clause 7.6. |
| filter | Extensible Parameter MonitoringFilter | A filter used by the *Server* to determine if the *MonitoredItem* should generate a *Notification*. If not used, this parameter is null. The *MonitoringFilter* parameter type is an extensible parameter type specified in Clause 8.3. It specifies the types of filters that can be used. |
| queueSize | Counter | The requested size of the *MonitoredItem* queue. If *Events* are lost an *Event* of the type *EventQueueOverflow* is generated.<br>The following values have special meaning:<br><u>Value</u> <u>Meaning</u><br>1 the queue has a single entry, effectively disabling queuing.<br>>1 a first-in-first-out queue is to be used.<br>Max Value the max size that the *Server* can support. This is used for *Event Notifications*. In this case the *Server* is responsible for the *Event* buffer. |
| discardOldest | Boolean | A boolean parameter that specifies the discard policy when the queue is full and a new *Notification* is to be enqueued. It has the following values:<br>TRUE the oldest (first) *Notification* in the queue is discarded. The new *Notification* is added to the end of the queue.<br>FALSE the new *Notification* is discarded. The queue is unchanged. |

## 7.12 MonitoringMode

The *MonitoringMode* is an enumeration that specifies whether sampling and reporting are enabled or disabled for a *MonitoredItem*. The value of the publishing enabled parameter for a *Subscription* does not affect the value of the monitoring mode for a *MonitoredItem* of the *Subscription*. The values of this parameter are defined in Table 138.

**Table 138 – MonitoringMode Values**

| Value | Description |
|---|---|
| DISABLED_0 | The item being monitored is not sampled or evaluated, and *Notifications* are not generated or queued. *Notification* reporting is disabled. |
| SAMPLING_1 | The item being monitored is sampled and evaluated, and *Notifications* are generated and queued. *Notification* reporting is disabled. |
| REPORTING_2 | The item being monitored is sampled and evaluated, and *Notifications* are generated and queued. *Notification* reporting is enabled. |

## 7.13 NotificationMessage

The components of this parameter are defined in Table 139.

**Table 139 – NotificationMessage**

| Name | Type | Description |
|---|---|---|
| NotificationMessage | structure | The *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*.<br>The *NotificationData* parameter type is an extensible parameter type specified in Clause 8.5. It specifies the types of *Notifications* that can be sent. The *ExtensibleParameter* type is specified in Clause 8.5.<br>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*. |

## 7.14 NumericRange

This parameter is defined in Table 140. A formal BNF definition of the numeric range can be found in Appendix A3.

**Table 140 – NumericRange**

| Name | Type | Description |
|---|---|---|
| NumericRange | String | A number or a numeric range. The syntax for the string contains one of the following two constructs.<br>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.<br>The second construct is a range represented by two integers separated by the colon (“:”) character. The first integer must always have a lower value than the second. For example, “5:7” is valid, while “7:5” and “5:5” are not. The minimum and maximum values that can be expressed by these integers are defined by the use of this parameter, and not by this parameter type definition.<br>No other characters, including white-space characters, are permitted.<br>A null string indicates that this parameter is not used. |

## 7.15 QueryDataSet

The components of this parameter are defined in Table 141.

**Table 141 – QueryDataSet**

| Name | Type | Description |
|---|---|---|
| QueryDataSet | structure | Data related to a *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. |

## 7.16 ReadValueId

The components of this parameter are defined in Table 142.

**Table 142 – ReadValueId**

| Name | Type | Description |
|---|---|---|
| ReadValueId | structure | Identifier for an item to read or to monitor. |
| nodeId | NodeId | *NodeId* of a *Node*. |
| attributeId | IntegerId | Id of the *Attribute*. This must be a valid *Attribute* id. The *IntegerId* is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA Part 6]. |
| indexRange | NumericRange | This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. If a range of elements is specified, the values are returned as a composite. The first element is identified by index 0 (zero). The *NumericRange* type is defined in Clause 7.14.<br>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. |
| 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*.<br>A *Client* can discover what *DataTypeEncoding*s are available by following the *HasEncoding Reference* from the *DataType Node* for a *Variable*.<br>UA defines standard *BrowseNames* which *Servers* must recognize even if the *DataType Nodes* are not visible in the *Server* address space. These *BrowseNames* are:<br><br>Default Binary The default or native binary (or non-XML) encoding.<br>Default XML The default XML encoding.<br><br>Each *DataType* must support at least one of these standard encodings. *DataTypes* that do not have a true binary encoding (e.g. they only have a non-XML text encoding) should use the *Default Binary* name to identify the encoding that is considered to be the default non-XML encoding. *DataTypes* that support at least one XML-based encoding must identify one of the encodings as the Default XML encoding. Other standards bodies may define other well-known data encodings that could be supported.<br>If this parameter is not specified then the *Server* must choose either the Default Binary or *Default XML* encoding according to what *Message* encoding (see [UA Part 6]) is used for the *Session*. If the *Server* does not support the encoding that matches the *Message* encoding then the *Server* must choose the default encoding that it does support.<br>If this parameter is specified for a *MonitoredItem*, the *Server* must set the *StructureChanged* bit in the *StatusCode* (see Clause 7.28) if the *DataTypeEncoding* changes. The *DataTypeEncoding* changes if the *DataTypeVersion* of the *DataTypeDescription* or the *DataTypeDictionary* associated with the *DataTypeEncoding* changes. |

## 7.17 ReferenceDescription

The components of this parameter are defined in Table 143.

**Table 143 – ReferenceDescription**

| **Name** | **Type** | **Description** |
|---|---|---|
| ReferenceDescription | structure | Reference parameters returned for the browse *Service* and exported by the *ExportNodeService*. |
| 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 a inverse *Reference*. |
| targetNodeInfo | structure | Information that describes the *TargetNode* of the *Reference* |
| nodeId | Expanded NodeId | *NodeId* of the *TargetNode* as assigned by the *Server* identified by the *Server* index. The *ExpandedNodeId* type is defined in Clause 7.7.<br>If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *nodeId* must contain the absolute namespace URI. If the *TargetNode* is a local *Node* the *nodeId* must 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*. |
| Notes:<br>1 If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *TargetNode browseName*, nodeClass and typeDefinition may be null or empty. If they are not, they might not be up to date because the local *Server* might not continuously monitor the remote *Server* for changes. | | |

## 7.18 RelativePath

A *RelativePath* is a string that describes a sequence of *References* and *Nodes* to follow. The components of a *RelativePath* are specified in Table 144. A formal BNF definition of the *RelativePath* can be found in Appendix A2.

**Table 144 – RelativePath**

| Symbol | Meaning |
|---|---|
| / | The forward slash character indicates that the *Server* is to follow any subtype of *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. A '!' in front of the BrowseName is used to indicate that the inverse *Reference* should be followed. If the *BrowseName* of a *ReferenceType* with subtypes is specified then any *References* of the subtypes are followed as well.<br>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.<br>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:<br>Received browse path name — Resolves to<br>"&/Name_1" — "/Name_1"<br>"&.Name_2" — ".Name_2"<br>"&:Name_3" — ":Name_3"<br>"&&Name_4" — "&Name_4" |

Table 145 provides examples of *RelativePaths*.

**Table 145 – RelativePath Examples**

| Browse Path | Description |
|---|---|
| "/Block&.Output" | Follows any hierarchical *Reference* with target *BrowseName* = "Block.Output". |
| "/Truck.NodeVersion" | Follows any hierarchical *Reference* with target *BrowseName* = "Truck" and from there a *HasProperty* or *HasComponent Reference* to a target with *BrowseName* "NodeVersion". |
| "<ConnectedTo>Boiler/HeatSensor" | Follows any Reference with a *BrowseName* or *InverseName* = 'ConnectedTo' and finds targets with *BrowseName* = 'Boiler'. From there follows any hierarchical *Reference* and find targets with *BrowseName* = 'HeatSensor'. |
| "<ConnectedTo>Boiler/" | Follows any Reference with a *BrowseName* or *InverseName* = 'ConnectedTo' and finds targets with *BrowseName* = 'Boiler'. From there it finds all targets of hierarchical *References* . |
| "<0:HasChild>2:Wheel" | Follows any Reference with a *BrowseName* = 'HasChild' and qualified with the default 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 ignored. |
| "<0:HasChild>" | Finds all targets of *References* with a *BrowseName* = 'HasChild' and qualified with the default UA namespace. |

## 7.19 RequestHeader

The components of this parameter are defined in Table 146.

**Table 146 – RequestHeader**

| **Name** | **Type** | **Description** |
|---|---|---|
| RequestHeader | structure | Common parameters for all requests submitted on a *Session*. |
| securityHeader | Security Header | Common security parameter. This parameter assigns the SecureChannel related security settings to the *Message*.<br>The *SecurityHeader* type is defined in Clause 7.21. |
| sessionId | IntegerId | *Server*-unique identifier for the *Session*. The *IntegerId* type is defined in Clause 7.9. |
| timestamp | UtcTime | The time the *Client* sent the request. |
| sequenceNumber | Counter | The sequence number of the request. The initial sequence number to use is one, and when the sequence number rolls over, it rolls over to one. Zero is never used. The *Counter* type is defined in Clause 7.3 |
| returnDiagnostics | UInt32 | A bit mask that identifies the types of vendor-specific diagnostics to be returned in *diagnosticInfo r*esponse parameters.<br>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.<br>Bit Value — Diagnostics to return<br>0x0000 0001 ServiceLevel / SymbolicId<br>0x0000 0002 ServiceLevel / LocalizedText<br>0x0000 0004 ServiceLevel / AdditionalInfo<br>0x0000 0008 ServiceLevel / Inner *StatusCode*<br>0x0000 0010 ServiceLevel / Inner Diagnostics<br>0x0000 0020 OperationLevel / SymbolicId<br>0x0000 0040 OperationLevel / LocalizedText<br>0x0000 0080 OperationLevel / AdditionalInfo<br>0x0000 0100 OperationLevel / Inner *StatusCode*<br>0x0000 0200 OperationLevel / Inner Diagnostics<br>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, then diagnostics information is not returned.<br>*Level*:<br>ServiceLevel return diagnostics in the *diagnosticInfo* of the *Service*.<br>OperationLevel return diagnostics in the *diagnosticInfo* defined for individual operations requested in the *Service*.<br>*Type:*:<br>SymbolicId return a namespace-qualified, symbolic identifier for an error or condition. The maximum length of this identifier is 32 characters.<br>LocalizedText return up to 256 bytes of localized text that describes the symbolic id<br>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.<br>InnerStatusCode return the inner *StatusCode* associated with the operation or *Service*.<br>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. |
| auditLogEntryId | 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. |
| timeoutHint | Duration | This timeout is used in the *Client* side *Communication Stack* to set the timeout on a per-call base.<br>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.<br>The value of 0 indicates no timeout. |
| additionalHeader | Extensible Parameter AdditionalHeader | Reserved for future use.<br>Applications that do not understand the header should ignore it. |

## 7.20 ResponseHeader

The components of this parameter are defined in Table 147.

**Table 147 – ResponseHeader**

| Name | Type | Description |
|---|---|---|
| ResponseHeader | structure | Common parameters for all responses. |
| securityHeader | Security Header | Common security parameter. This parameter assigns the SecureChannel-related security settings to the *Message*.<br>The *SecurityHeader* type is defined in Clause 7.21. |
| sessionId | IntegerId | *Server*-unique identifier for the *Session*. The *IntegerId* type is defined in Clause 7.9. |
| timestamp | UtcTime | The time the *Server* sent the response. |
| sequenceNumber | Counter | The sequence number given by the *Client* to the request. |
| serviceResult | StatusCode | Standard, OPC UA-defined result of the *Service* invocation. The *StatusCode* type is defined in Clause 7.28. |
| diagnosticInfo | 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 Clause 7.5. |
| stringTable [] | String | There is one string in this list for each unique namespace, symbolic identifier, and localized text string contained in all of the diagnostics information parameters contained in the response (see Clause 7.5). Each is identified within this table by its zero-based index. |
| additionalHeader | Extensible Parameter AdditionalHeader | Reserved for future use.<br>Applications that do not understand the header should ignore it. |

## 7.21 SecurityHeader

The signing and encryption parameters for all *Messages* in the context of a *SecureChannel* are passed in the *securityHeader* parameter of the *RequestHeader* or *ResponseHeader* of the *Messages*. This parameter may be passed automatically by the *Communication Stack* in some of the mappings defined in [UA Part 6]. If this is the case, then this parameter is not part of the *RequestHeader* or *ResponseHeader* parameters.

[UA Part 6] defines the structure of the *SecurityHeader* for the different mappings.

## 7.22 SecurityPolicy

Security policies specify which encryption and hashing functions are used for which functions and which protocols will perform the various functions. The security policies are specified in more detail in [UA Part 2].

The components of this parameter are defined in Table 148.

**Table 148 – SecurityPolicy**

| Name | Type | Description |
|---|---|---|
| SecurityPolicy | structure | Specifies what security policies the *Server* requires. |
| uri | String | A URI that identifies the *SecurityPolicy*.<br>UA defines the standard *SecurityPolicies* in [UA Part 7]. |
| digest | String | The URI of the digest algorithm to use. |
| symmetricSignature | String | The URI of the symmetric signature algorithm to use. |
| symmetricKeyWrap | String | The URI of the symmetric key wrap algorithm to use. |
| symmetricKeyEncryption | String | The URI of the symmetric key encryption algorithm to use. |
| symmetricKeyLength | Int32 | The length in bits for the symmetric key. |
| asymmetricSignature | String | The URI of the asymmetric signature algorithm to use. |
| asymmetricKeyWrap | String | The URI of the asymmetric key wrap algorithm to use. |
| asymmetricKeyEncryption | String | The URI of the asymmetric key encryption algorithm to use. |
| asymmetricKeyMinLength | Int32 | The minimum length in bits for the asymmetric key. |
| asymmetricKeyMaxLength | Int32 | The maximum length in bits for the asymmetric key. |
| derivedKey | String | The key derivation algorithm to use. |
| derivedEncyptionKeyLength | Int32 | The length of the derived key used for encryption. |
| derivedSignatureKeyLength | Int32 | The length of the derived key used for signatures. |

## 7.23 ServerDescription

The components of this parameter are defined in Table 149.

**Table 149 – ServerDescription**

| Name | Type | Description |
|---|---|---|
| ServerDescription | structure | Specifies a *Server* that is available. |
| serverUri | String | The globally unique identifier for the *Server*. |
| serverName | LocalizedText | A localized descriptive name for the *Server*. |
| serverType | Enum<br>ServerType | The type of server.<br>This value is an enumeration with one of the following values:<br>SERVER_0 The server is a regular *Server*.<br>DISCOVERY_1 The server is a *DiscoveryServer.* |

## 7.24 ServiceFault

The components of this parameter are defined in Table 150.

The *ServiceFault* parameter is returned instead of the Service response message when a service level error occurs. The *sessionId* and *sequenceNumber* in the *ResponseHeader* should be set to what was provided in the *RequestHeader* even if these values were not valid. The level of diagnostics returned in the *ResponseHeader* is specified by the *returnDiagnostics* parameter in the *RequestHeader*.

The exact use of this parameter depends on the mappings defined in [UA Part 6].

**Table 150 – ServiceFault**

| Name | Type | Description |
|---|---|---|
| ServiceFault | structure | An error response sent when a service level error occurs. |
| responseHeader | ResponseHeader | Common response parameters (see Clause 7.20 for *ResponseHeader* definition). |

## 7.25 SignatureData

The components of this parameter are defined in Table 151.

**Table 151 – SignatureData**

| Name | Type | Description |
|---|---|---|
| SignatureData | structure | Contains a digital signature created with a *Certificate*. |
| signature | ByteString | This is a signature generated with the private key associated with a *Certificate*. |
| signatureAlgorithm | String | A string containing the URI of the *signatureAlgorithm*.<br>The list of UA-defined names that may be used is specified in [UA Part 7]. |

## 7.26 SignedSoftwareCertificate

The components of this parameter are defined in Table 152. See [UA Part 6] for the details on the signing of *Certificates*.

**Table 152 – SignedSoftwareCertificate**

| Name | Type | Description |
|---|---|---|
| SignedSoftwareCertificate | structure | A *Certificate* that identifies the capabilities of a *Client* or a *Server*. |
| certificateData | ByteString | Contents of the *Certificate*. The fields of this parameter are defined in Table 153. These fields are serialized using the OPC Binary Encoding Rules and transferred in a byte string. The structure of the *softwareCertificate* type is specified in Clause 7.27. |
| issuerSignature | ByteString | Signature of the *SoftwareCertificate* body byte string. This signature is generated by the *Certificate* issuer using its *PrivateKey*.<br>To validate the *Certificate*, hashes the *Certificate* data using the issuer algorithm specified in the *Certificate* and then verifies the signature using the issuer's *PublicKey*.<br>The *PublicKey* of the *Certificate* issuer is obtained by the receiver through external means. |

## 7.27 SoftwareCertificate

The components of this parameter are defined in Table 153.

**Table 153 – SoftwareCertificate**

| Name | Type | Description |
|---|---|---|
| SoftwareCertificate | structure | *Certificate* signed by the issuer. |
| serverInfo | BuildInfo | Information that identifies the software build that this *Certificate* certifies. |
| issuedBy | String | URI of the certifying authority. |
| issueDate | UtcTime | Date and time that the *Certificate* was issued. |
| expirationDate | UtcTime | Date and time that the *Certificate* expires. |
| applicationCertificate | ByteString | The DER encoded form of the X.509 *Certificate* which is assigned to the application. |
| issuerCertificateThumbprint | ByteString | The thumbprint of the issuer's X.509 *Certificate* used to sign the *SoftwareCertificate*. |
| issuerSignatureAlgorithm | String | The algorithm used to create the issuer's signature. |
| supportedProfiles [] | structure | List of supported *Profiles* |
| profileUri | String | URI that identifies the *Profile* |
| profileName | String | Human-readable name for the *Profile*. This name does not have to be globally-unique. |
| complianceLevel | enum Compliance Level | An enumeration that specifies the compliance level of the *Profile*. It has the following values :<br>UNTESTED_0 the profiled capability has not been tested successfully<br>PARTIAL_1 the profiled capability has been tested and has passed critical tests, as defined by the certifying authority.<br>SELFTESTED_2 the profiled capability has been successfully tested using a self-test system authorized by the certifying authority.<br>CERTIFIED_3 the profiled capability has been successfully tested by a testing organisation authorized the certifying authority. |

### 7.28 StatusCode

A *StatusCode* in UA is numerical value that is used to report the outcome of an operation performed by a UA *Server*. This code may have associated diagnostic information that describes the status in more detail; however, the code by itself is intended to provide *Client* applications with enough information to make decisions on how to process the results of a UA *Service*.

The *StatusCode* is a 32-bit unsigned integer. The top 16 bits represent the numeric value of the code that must be used for detecting specific errors or conditions. The bottom 16 bits are bit flags that contain additional information but do not affect the meaning of the *StatusCode*.

All UA *Clients* must always check the *StatusCode* associated with a result before using it. Results that have an uncertain/warning status associated with them must be used with care since these results might not be valid in all situations. Results with a bad/failed status must never be used.

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

UA *Servers* should use uncertain/warning *StatusCodes* if they could not complete the operation in the manner requested by the *Client*, however, the operation did not fail entirely.

The exact bit assignments are shown in Table 154.

### Table 154 – StatusCode Bit Assignments

| Field | Bit Range | Description |
|---|---|---|
| Severity | 30:31 | Indicates whether the *StatusCode* represents a good, bad or uncertain condition. These bits have the following meanings:<br>Good Success — 00 — Indicates that the operation was successful and the associated results may be used.<br>Uncertain Warning — 01 — Indicates that the operation was partially successful and that associated results might not be suitable for some purposes.<br>Bad Failure — 10 — Indicates that the operation failed and any associated results cannot be used.<br>Reserved — 11 — Reserved for future use. All *Clients* should treat a *StatusCode* with this severity as “Bad”. |
| Reserved | 29:28 | Reserved for future use. Must always be zero. |
| SubCode | 16:27 | The code is a numeric value assigned to represent different conditions. Each code has a symbolic name and a numeric value. All descriptions in the UA specification refer to the symbolic name. [UA Part 6] maps the symbolic names onto a numeric value. |
| Structure Changed | 15:15 | 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.<br>*Servers* must set this bit if the *DataTypeEncoding* used for a *Variable* changes. Clause 7.16 describes how the *DataTypeEncoding* is specified for a *Variable*.<br>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.<br>*Servers* must also set this bit if the length of a fixed-length array *Variable* changes.<br>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.<br>This bit has meaning only for *StatusCodes* returned as part of a data change *Notification*. *StatusCodes* used in other contexts must always set this bit to zero. |
| Semantics Changed | 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*.<br>*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.<br>[UA Part 8] defines the conditions where a *Server* must 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.<br>This bit has meaning only for *StatusCodes* returned as part of a data change *Notification*. *StatusCodes* used in other contexts must always set this bit to zero. |
| Reserved | 12:13 | Reserved for future use. Must always be zero. |
| InfoType | 10:11 | The type of information contained in the info bits. These bits have the following meanings:<br>NotUsed — 00 — The info bits are not used and must be set to zero.<br>DataValue — 01 — The *StatusCode* and its info bits are associated with a data value returned from the *Server*.<br>Reserved — 1X — Reserved for future use. The info bits must be ignored. |
| InfoBits | 0:9 | Additional information bits that qualify the *StatusCode*.<br>The structure of these bits depends on the Info Type field. |

Table 155 describes the structure of the *InfoBits* when the Info Type is set to *DataValue* (01).

**Table 155 – DataValue InfoBits**

| Info Type | Bit Range | Description |
|---|---|---|
| LimitBits | 8:9 | The limit bits associated with the data value. The limits bits have the following meanings::<br>**Limit** **Bits** **Description**<br>None 00 The value is free to change.<br>Low 01 The value is at the lower limit for the data source.<br>High 10 The value is at the higher limit for the data source.<br>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. Must always be zero. |
| HistorianBits | 0:4 | These bits are set only when reading historical data. They indicate where the data value came from and provide information that affects how the *Client* uses the data value. The historian bits have the following meaning:<br>Raw XXX00 A raw data value.<br>Calculated XXX01 A data value which was calculated.<br>Interpolated XXX10 A data value which was interpolated.<br>Reserved XXX11 Undefined.<br>Partial XX1XX A data value which was calculated with an incomplete interval.<br>Extra Data X1XXX A raw data value that hides other data at the same timestamp.<br>Multi Value 1XXXX Multiple values match the aggregate criteria (i.e. multiple minimum values at different timestamps within the same interval)<br>[UA Part 11] describes how these bits are used in more detail. |

## Common *StatusCodes*

Table 156 defines the common *StatusCodes* for all *Service* results. [UA Part 6] maps the symbolic names to a numeric value.

**Table 156 – Common Service Result Codes**

| **Symbolic Id** | **Description** |
|---|---|
| Good | The operation was successful. |
| Good_CompletesAsynchronously | The processing will complete asynchronously. |
| Good_CommunicationEvent | The communication layer has raised an event |
| Good_ShutdownEvent | The system is shutting down. |
| Good_CallAgain | The operation is not finished and needs to be called again. |
| Good_NonCriticalTimeout | A non-critical timeout occurred. |
| Good_SubscriptionTransfered | The subscription was transferred to another session. |
| | |
| Bad_UnexpectedError | An unexpected error occurred. |
| Bad_InternalError | An internal error occurred as a result of a programming or configuration error. |
| Bad_OutOfMemory | Not enough memory to complete the operation. |
| Bad_ResourceUnavailable | An operating system resource is not available. |
| Bad_CommunicationError | A low level communication error occurred. |
| Bad_ConnectionRejected | Could not establish a network connection to remote server. |
| Bad_ConnectionClosed | The network connection has been closed. |
| Bad_InvalidState | The operation cannot be completed because the object is closed, uninitialized or in some other invalid state. |
| Bad_EncodingError | Encoding halted because of invalid data in the objects being serialized. |
| Bad_DecodingError | Decoding halted because of invalid data in the stream. |
| Bad_EndOfStream | Cannot move beyond end of the stream. |
| Bad_NoDataAvailable | No data is currently available for reading from a non-blocking stream. |
| Bad_ExpectedStreamToBlock | The stream did not return all data requested (possibly because it is a non-blocking stream). |
| Bad_WaitingForResponse | The asynchronous operation is waiting for a response. |
| Bad_OperationAbandoned | The asynchronous operation was abandoned by the caller. |
| Bad_WouldBlock | Non blocking behaviour is required and the operation would block. |
| Bad_SyntaxError | A value had an invalid syntax. |
| Bad_ConnectionIdInvalid | The specified connection id was not valid. |
| Bad_UnknownResponse | An unrecognized response was received from the server. |
| Bad_Timeout | The operation timed out. |
| Bad_MaxConnectionsReached | The operation could not be finished because all available connections are in use. |
| Bad_ServiceUnsupported | The server does not support the requested service. |
| Bad_Shutdown | The operation was cancelled because the application is shutting down |
| Bad_Disconnect | The operation was cancelled because the network connection with the server was closed. |
| Bad_ServerNotConnected | The operation could not complete because the client is not connected to the server. |
| Bad_ServerHalted | The server has stopped and cannot process any requests. |
| Bad_InvalidArgument | One or more arguments are invalid.<br>Each service defines parameter-specific *StatusCodes* and these *StatusCodes* must be used instead of this general error code. This error code must be used only by the communication stack and in services where it is defined in the list of valid *StatusCodes* for the service. |
| Bad_NothingToDo | There was nothing to do because the client passed a list of operations with no elements. |
| Bad_DataTypeIdUnknown | The extension object cannot be (de)serialized because the data type id is not recognized. |
| Bad_ExtensibleParameterInvalid | The extensible parameter provided is not a valid for the service. |
| Bad_ExtensibleParameterUnsupported | The extensible parameter provided is valid but the server does not support it. |
| Bad_CertificateInvalid | The certificate is not valid. |
| Bad_CertificateExpired | The certificate is expired or not yet valid. |
| Bad_CertificateRevoked | The certificate has been revoked by the certification authority. |
| Bad_CertificateUntrusted | The certificate is valid; however, the server does not recognize it as a trusted certificate. |
| Bad_UserAccessDenied | User does not have permission to perform the requested operation. |
| Bad_IdentityTokenInvalid | The user identity token is not valid. |
| Bad_IdentityTokenRejected | The user identity token is valid but the server has rejected it. |
| Bad_SecureChannelIdInvalid | The specified secure channel is not longer valid. |
| Bad_SequenceNumberInvalid | The sequence number is less than that of a previous update request. |

| | |
|---|---|
| Bad_InvalidTimestamp | The timestamp is outside the range allowed by the server. |
| Bad_SignatureInvalid | The message signature is invalid. |
| Bad_NonceInvalid | The nonce does appear to be not a random value or it is not the correct length. |
| Bad_SessionIdInvalid | The session id is not valid. |
| Bad_SessionClosed | The session was closed by the client. |
| Bad_SessionNotActivated | The session cannot be used because ActivateSession has not been called. |
| Bad_SubscriptionIdInvalid | The subscription id is not valid. |
| Bad_NoSubscription | There is no subscription available for this session. |
| Bad_RequestHeaderInvalid | The header for the request is missing or invalid. |
| Bad_ContinuationPointInvalid | The continuation point is no longer valid. |
| Bad_TimestampsToReturnInvalid | The timestamps to return parameter is invalid. |

Table 157 defines the common *StatusCodes* for all operation level results. [UA Part 6] maps the symbolic names to a numeric value. The common *Service* result codes can be also contained in the operation level.

**Table 157 – Common Operation Level Result Codes**

| **Symbolic Id** | **Description** |
|---|---|
| Good_Overload | Sampling has slowed down due to resource limitations. |
| Good_Clamped | The value written was accepted but was clamped. |
| | |
| Uncertain | The value is uncertain but no specific reason is known |
| | |
| Bad | The value is bad but no specific reason is known |
| Bad_NoCommunication | Communication with the data source is defined, but not established, and there is no last known value available.<br>This status/sub status is used for cached values before the first value is received. |
| Bad_WaitingForInitialData | Waiting for the server to obtain values from the underlying data source.<br>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. |
| Bad_NodeIdInvalid | The syntax of the node id is not valid. |
| Bad_NodeIdUnknown | The node id refers to a node that does not exist in the server address space. |
| Bad_AttributeIdInvalid | The attribute is not supported for the specified *Node*. |
| Bad_IndexRangeInvalid | The syntax of the index range parameter is invalid. |
| Bad_IndexRangeNoData | No data exists within the range of indexes specified. |
| Bad_DataEncodingInvalid | The data encoding is invalid. |
| Bad_DataEncodingUnsupported | The server does not support the requested data encoding for the node. |
| Bad_NoReadRights | The access level does not allow reading or subscribing to the *Node*. |
| Bad_NoWriteRights | The access level does not allow writing to the *Node*. |
| Bad_OutOfRange | The value was out of range. |
| Bad_NotSupported | The requested operation is not supported. |
| Bad_NotFound | A requested item was not found or a search operation ended without success. |
| Bad_ObjectDeleted | The object cannot be used because it has been deleted. |
| Bad_NotImplemented | Requested operation is not implemented. |
| Bad_MonitoringModeInvalid | The monitoring mode is invalid. |
| Bad_MonitoredItemIdInvalid | The monitoring item id does not refer to a valid monitored item. |
| Bad_FilterNotAllowed | A monitoring filter cannot be used in combination with the attribute specified. |
| Bad_StructureMissing | A mandadatory structured parameter was missing or null. |

## 7.29 TimestampsToReturn

The *TimestampsToReturn* is an enumeration that specifies the *Timestamp Attributes* to be transmitted for *MonitoredItems* or *Nodes* in *HistoryRead*. The values of this parameter are defined in Table 158.

**Table 158 – TimestampsToReturn Values**

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

## 7.30 UserTokenPolicy

The components of this parameter are defined in Table 159.

**Table 159 – UserTokenPolicy**

| Name | Type | Description |
|---|---|---|
| UserTokenPolicy | structure | Specifies a *UserIdentityToken* that a *Server* will accept. |
| tokenType | Enum UserIdentityTokenType | The type of user identity token required.<br>This value is an enumeration with one of the following values:<br>DEFAULT_0 No token is required.<br>USERNAME_1 A username/password token.<br>CERTIFICATE_2 An X509v3 certificate token.<br>ISSUEDTOKEN_3 Any WS-Security defined token. |
| issuerType | String | A URI indicating the type of token issuer.<br>[UA Part 7] defines URIs for common issuer types.<br>Vendors may specify their own issuer types that describe a vendor defined user database or identity token server.<br>Clients that do not recognize the IssuerType should still be able to attempt a connection if the *TokenType* is CERTIFICATE or USERNAME.<br>If the *TokenType* is ISSUEDTOKEN then the issuer type may be some version of WS-Trust. |
| issuerUrl | String | A URL of the token issuing service.<br>This meaning of this value depends on the *issuerType* |

### 7.31 ViewDescription

The components of this parameter are defined in Table 160.

**Table 160 – 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 must 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 [UA Part 3]. If *timestamp* is set this parameter must be 0. The current view is used if timestamp is null and viewVersion is 0. |

# 8 Extensible parameter type definition

## 8.1 Overview

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

## 8.2 ExtensibleParameter

The *ExtensibleParameter* defines a data structure with two elements. The *parameterTypeId* specifies the data type encoding of the second element. Therefore the second element is specified as "--". The *ExtensibleParameter* base type is defined in Table 161.

The following clauses define concrete extensible parameters that are common to OPC UA. Additional parts of this multi-part specification can define additional extensible parameter types.

**Table 161 – ExtensibleParameter Base Type**

| Name | Type | Description |
|---|---|---|
| ExtensibleParameter | structure | Specifies the details of an extensible parameter type. |
| parameterTypeId | NodeId | Identifies the data type of the parameter that follows. |
| parameterData | -- | The details for the extensible parameter type. |

## 8.3 MonitoringFilter parameters

### 8.3.1 Overview

The *CreateMonitoredItem Service* allows specifying a filter for each *MonitoredItem*. The *MonitoringFilter* is an extensible parameter whose structure depends on the type of item being monitored. The *parameterTypeIds* are defined in Table 162. Other types can be defined by additional parts of this multi-part specification or other standards based on OPC UA.

**Table 162 – MonitoringFilter parameterTypeIds**

| Symbolic Id | Description |
|---|---|
| DataChangeFilter | The change in a data value that will cause a *Notification* to be generated. |
| EventFilter | If a *Notification* conforms to the *EventFilter*, the *Notification* is sent to the *Client*. |

### 8.3.2 DataChangeFilter

The *DataChangeFilter* defines the conditions under which a data change notification should be reported and, optionally, a range or band for value changes where no *DataChange Notification* is generated. This range is called *Deadband*. The *DataChangeFilter* is defined in Table 163.

**Table 163 – DataChangeFilter**

| Name | Type | Description |
|---|---|---|
| DataChangeFilter | structure | |
| trigger | enum DataChangeTrigger | Specifies the conditions under which a data change notification should be reported. It has the following values :<br>STATUS_0 Report a notification ONLY if the *StatusCode* associated with the value changes. See Table 157 for *StatusCodes* defined in this Part. [UA Part 8] specifies additional *StatusCodes* that are valid in particular for device data<br>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.<br>**This is the default setting if no filter is set.**<br>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.<br><br>If the DataChangeFilter is not applied to the monitored item, STATUS_VALUE is the default reporting behaviour. |
| deadbandType | UInt32 | A bit mask that defines the *Deadband* type and behaviour.<br>Bit Value deadbandType<br>0x0000 0000 No *Deadband* calculation should be applied.<br>0x0000 0001 AbsoluteDeadband (see below)<br>0x0000 0002 PercentDeadband (This type is specified in [UA Part 8]). |
| deadbandValue | Double | The *Deadband* is applied only if<br>* the *trigger* includes value changes and<br>* the *deadbandType* is set appropriately.<br>Deadband is generally ignored if the status of the data item changes.<br><br>***DeadbandType = AbsoluteDeadband***:<br>For this type the *deadbandValue* contains the absolute change in a data value that will cause a *Notification* to be generated. This parameter applies only to *Variable*s with any integer or floating point data type.<br>An exception that causes a *DataChange Notification* based on an AbsoluteDeadband is determined as follows:<br>Exception **if (absolute value of (last cached value - current value) > *AbsoluteDeadband*)**<br>The last cached value is defined as the most recent value previously sent to the *Notification* channel.<br>If the item is an array of values, the entire array is returned if any array element exceeds the AbsoluteDeadband.<br><br>***DeadbandType = PercentDeadband:***<br>This type is specified in [UA Part 8] |

### 8.3.3 EventFilter

#### 8.3.3.1 General

The *EventFilter* provides for the filtering and content selection of *Event Subscriptions*.

If an *Event Notification* conforms to the filter defined by the *where* parameter of the *EventFilter*, then the *Notification* is sent to the *Client*.

Each *Event Notification* will include the field defined by the *select* array parameter of the *EventFilter*. [UA Part 3] describes the *Event* model and the base *EventTypes*. The fields of the base *EventTypes* and their representation in the *AddressSpace* are specified in [UA Part 5].

If an *EventType* does not support a selected field, a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* must be *Bad_NotSupported*.

If the selected array is empty or one of the NodeIds is invalid, the *Server* will return an error. At least one field must be selected.

Table 164 defines the EventFilter structure.

**Table 164 – EventFilter structure**

| Name | Type | Description |
|---|---|---|
| EventFilter | structure | |
| selectClauses [] | NodeId | List of NodeIds for the *EventType* field to return with each *Event* in a *Notification*. At least one *EventType* field must be selected.<br>If a field is not applicable to an *Event* being returned a null value will be returned for that field. |
| whereClause | ContentFilter | Limit the *Notifications* to those *Events* that match the criteria defined by this ContentFilter. The ContentFilter structure is described in Clause 7.2. |

### 8.4 FilterOperand parameters

#### 8.4.1 Overview

The *ContentFilter* structure specified in Clause 7.2 defines a collection of elements that makes up a filter criteria and contains different types of *FilterOperands*. The *FilterOperand* parameter is an extensible parameter. This parameter is defined in Table 165.

**Table 165 –FilterOperand parameterTypeIds**

| Symbolic Id | Description |
|---|---|
| Element | Contains an index into the array of elements. This type is use to build a logic tree of sub-elements by linking the operand of one element to a sub-element. |
| Literal | Contains a literal value. |
| Attribute | Contains a *NodeId Reference* to an *Attribute* of a *Variable* or *Property*. This must be a *NodeId* from the type system. |
| Property | Contains a *NodeId* and a name of a *Property* of the *Node*. This must be a *NodeId* from the type system. |

#### 8.4.2 ElementOperand

The *ElementOperand* provides the linking to sub-elements within a *ContentFilter*. The link is in the form of an integer that is used to index into the array of elements contained in the *ContentFilter*. An index is considered valid if its value is greater than the element index it is part of and it does not *Reference* a non-existent element. *Clients* must construct filters in this way to avoid circular and invalid *References*. *Servers* should protect against invalid indexes by verifying the index prior to using it.

Table 166 defines the *ElementOperand* type.

**Table 166 – ElementOperand**

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

#### 8.4.3 LiteralOperand

Table 167 defines the *LiteralOperand* type.

**Table 167 – LiteralOperand**

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

### 8.4.4 AttributeOperand

Table 168 defines the *AttributeOperand* type.

**Table 168 – AttributeOperand**

| Name | Type | Description |
|---|---|---|
| AttributeOperand | structure | Attribute of a *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. |
| attributeId | IntegerId | Id of the *Attribute*. This must be a valid *Attribute* id. The *IntegerId* is defined in Clause 7.9. The IntegerIds for the Attributes are defined in [UA Part 6]. |
| indexRange | NumericRange | This parameter is used to identify a single element of a structure, bit mask or an array, or a single range of indexes for arrays or bit masks. The first element is identified by index 0 (zero). The *NumericRange* type is defined in Clause 7.14.<br>This parameter is not used if the specified *Attribute* is not an array, a bit mask or a structure. However, if the specified *Attribute* is an array, a bit mask or a structure, and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used. |

### 8.4.5 PropertyOperand

Table 169 defines the *PropertyOperand* type.

**Table 169 – PropertyOperand**

| Name | Type | Description |
|---|---|---|
| PropertyOperand | structure | Property in the address space. |
| nodeId | NodeId | *NodeId* of a *Node* in the type system. |
| alias | String | An optional parameter used to identify or refer to an alias. An alias is a symbolic name that can be used to alias this operand and use it in other location in the filter structure. |
| property | QualifiedName | *Property. Implicit Reference to the value Attribute of the Property* |
| indexRange | NumericRange | This parameter is used to identify a single element of a structure, bit mask or an array, or a single range of indexes for arrays or bit masks. The first element is identified by index 0 (zero). The *NumericRange* type is defined in Clause 7.14.<br>This parameter is not used if the specified *Attribute* is not an array, a bit mask or a structure. However, if the specified *Attribute* is an array, a bit mask or a structure, and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used. |

## 8.5 NotificationData parameters

### 8.5.1 Overview

The *NotificationMessage* structure used in the *Subscription Service* set allows specifying different types of *NotificationData*. The *NotificationData* parameter is an extensible parameter whose structure depends on the type of *Notification* being sent. This parameter is defined in Table 170. Other types can be defined by additional parts of this multi-part specification or other standards based on OPC UA.

**Table 170 – NotificationData parameterTypeIds**

| Symbolic Id | Description |
|---|---|
| DataChange | *Notification* data parameter used for data change *Notifications*. |
| Event | *Notification* data parameter used for *Event Notifications*. |

## 8.5.2 DataChangeNotification parameter

Table 171 defines the *NotificationData* parameter used for data change notifications. This structure contains the monitored data items that are to be reported. Monitored data items are reported under two conditions:

a) If the *MonitoringMode* is set to REPORTING and a change in value or its status (represented by its *StatusCode*) is detected.

b) If the *MonitoringMode* is set to SAMPLING, the *MonitoredItem* is linked to a triggering item and the triggering item triggers.

See Clause 5.12 for a description of the *MonitoredItem Service* set, and in particular the *MonitoringItemModel* and the *TriggeringModel*.

After creating a *MonitoredItem* the current value or status of the monitored Attribute must be queued without applying the filter. If the current value is not available after the first sampling interval the first *Notification* must be queued after getting the initial value or status from the data source.

**Table 171 – DataChangeNotification**

| Name | Type | Description |
|---|---|---|
| DataChangeNotification | structure | Data change *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 Clause 7.9 |
| value | DataValue | The *StatusCode*, value and timestamp(s) of the monitored *Attribute* depending on the sampling and queuing configuration.<br>If the *StatusCode* indicates an error then the value and timestamp(s) are to be ignored.<br>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.<br>*DataValue* is a common type defined in Clause 7.4. |
| diagnosticInfos [] | DiagnosticInfo | List of diagnostic information. The size and order of this list matches the size and order of the *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 Clause 7.5. |

## 8.5.3 EventNotification parameter

Table 172 defines the *NotificationData* parameter used for *Event* notifications.

The EventNotification defines a table structure that is used to return *Event* fields to a *Client Subscription*. The structure is in the form of a table consisting of one or more *Events*, each containing an array of one or more fields. The selection and order of the fields returned for each *Event* is identical to the selected parameter of the *EventFilter*.

**Table 172 –EventNotification**

| Name | Type | Description |
|---|---|---|
| EventNotification | structure | Event *Notification* data |
| events [] | EventFieldList | The list of *Events* being delivered |
| eventFields [] | BaseDataType | List of selected *Event* fields. This will be a one to one match with the fields selected in the *EventFilter*.<br>If an *EventType* does not support a selected field, a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* must be *Bad_NotSupported*. Other *StatusCodes* may indicate other problems such as *Bad_UserAccessDenied*. |

## 8.6 NodeAttributes parameters

### 8.6.1 Overview

The *AddNodes Service* allows specifying the *Attributes* for the *Nodes* to add. The *NodeAttributes* is an extensible parameter whose structure depends on the type of the *Attribute* being added. It identifies the *NodeClass* that defines the structure of the *Attributes* that follow. The *parameterTypeIds* are defined in Table 173.

**Table 173 – NodeAttributes parameterTypeIds**

| Symbolic Id | Description |
|---|---|
| ObjectAttributes | Defines the *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*. |

### 8.6.2 ObjectAttributes parameter

Table 174 defines the *ObjectAttributes* parameter.

**Table 174 – ObjectAttributes**

| Name | Type | Description |
|---|---|---|
| ObjectAttributes | structure | Defines the *Attributes* for the *Object NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| eventNotifier | Byte | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.3 VariableAttributes parameter

Table 175 defines the *VariableAttributes* parameter.

**Table 175 – VariableAttributes**

| Name | Type | Description |
|---|---|---|
| VariableAttributes | structure | Defines the *Attributes* for the *Variable NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| value | Defined by the *DataType Attribute* | See [UA Part 3] for the description of this *Attribute*. |
| dataType | NodeId | See [UA Part 3] for the description of this *Attribute*. |
| arraySize | Int32 | See [UA Part 3] for the description of this *Attribute*. |
| accessLevel | Byte | See [UA Part 3] for the description of this *Attribute*. |
| userAccessLevel | Byte | See [UA Part 3] for the description of this *Attribute*. |
| minimumSamplingInterval | Int32 | See [UA Part 3] for the description of this *Attribute*. |
| historizing | Boolean | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.4 MethodAttributes parameter

Table 176 defines the *MethodAttributes* parameter.

**Table 176 – MethodAttributes**

| Name | Type | Description |
|---|---|---|
| BaseAttributes | structure | Defines the *Attributes* for the *Method NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| executable | Boolean | See [UA Part 3] for the description of this *Attribute*. |
| userExecutable | Boolean | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.5 ObjectTypeAttributes parameter

Table 177 defines the *ObjectTypeAttributes* parameter.

**Table 177 – ObjectTypeAttributes**

| Name | Type | Description |
|---|---|---|
| ObjectTypeAttributes | structure | Defines the *Attributes* for the *ObjectType NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| isAbstract | Boolean | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.6 VariableTypeAttributes parameter

Table 178 defines the *VariableTypeAttributes* parameter.

**Table 178 – VariableTypeAttributes**

| Name | Type | Description |
|---|---|---|
| VariableTypeAttributes | structure | Defines the *Attributes* for the *VariableType NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| value | Defined by the *DataType Attribute* | See [UA Part 3] for the description of this *Attribute*. |
| dataType | NodeId | See [UA Part 3] for the description of this *Attribute*. |
| arraySize | Int32 | See [UA Part 3] for the description of this *Attribute*. |
| isAbstract | Boolean | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.7 ReferenceTypeAttributes parameter

Table 179 defines the *ReferenceTypeAttributes* parameter.

**Table 179 – ReferenceTypeAttributes**

| Name | Type | Description |
|---|---|---|
| ReferenceTypeAttributes | structure | Defines the *Attributes* for the *ReferenceType NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| isAbstract | Boolean | See [UA Part 3] for the description of this *Attribute*. |
| symmetric | Boolean | See [UA Part 3] for the description of this *Attribute*. |
| inverseName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.8 DataTypeAttributes parameter

Table 176 defines the *DataTypeAttributes* parameter.

**Table 180 – DataTypeAttributes**

| Name | Type | Description |
|---|---|---|
| DataTypeAttributes | structure | Defines the *Attributes* for the *DataType NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| isAbstract | Boolean | See [UA Part 3] for the description of this *Attribute*. |

### 8.6.9 ViewAttributes parameter

Table 181 defines the *ViewAttributes* parameter.

**Table 181 – ViewAttributes**

| Name | Type | Description |
|---|---|---|
| ViewAttributes | structure | Defines the *Attributes* for the *View NodeClass* |
| displayName | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| description | LocalizedText | See [UA Part 3] for the description of this *Attribute*. |
| containsNoLoops | Boolean | See [UA Part 3] for the description of this *Attribute*. |
| eventNotifier | Byte | See [UA Part 3] for the description of this *Attribute*. |

## 8.7 UserIdentityToken parameters

### 8.7.1 Overview

The *UserIdentityToken* structure used in the *Server Service Set* allows *Clients* to specify the identity of the user they are acting on behalf of. The exact mechanism used to identify users depends on the system configuration. The different types of identity tokens are based on the most common mechanisms that are used in systems today. Table 182 defines the current set of user identity tokens.

**Table 182 – UserIdentityToken parameterTypeIds**

| Symbolic Id | Description |
|---|---|
| UserName | A user identified by user name and password. |
| X509v3 | A user identified by an X509v3 *Certificate*. |
| WSS | A user identified by a WS-*SecurityToken*. |

### 8.7.2 UserName identity tokens

The *UserName* identity token is used to pass simple username/password credentials to the *Server*. The password may be passed in its literal form (in which case the *Message* must be encrypted) or it may be hashed.

The hash algorithm depends on the application; however, UA *Profiles* may define standard algorithms.

Table 183 defines the User Name Identity Token parameter.

**Table 183 – UserName Identity Token**

| Name | Type | Description |
|---|---|---|
| UserName | structure | UserName value. |
| userName | String | A string that identifies the user. |
| password | String | The password for the user (may be hashed) |
| hashAlgorithm | String | The hash algorithm used. If not specified, the password is passed as plain text. |

### 8.7.3 X509v3 identity tokens

The X.509 Identiy Token is used to pass an X509v3 *Certificate* which is issued by the user.

Table 184 defines the X509IdentityToken parameter.

**Table 184 – X509 Identity Token**

| Name | Type | Description |
|---|---|---|
| X509v3 | structure | X509v3 value. |
| CertificateData | ByteString | The X509 *Certificate* in DER format. |

### 8.7.4 WSS identity tokens

The W*SS IdentityToken* 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 must be exchanged in UA as XML Security Tokens.

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

Table 185 defines the WSS Identity Token parameter.

**Table 185 – Issued Identity Token**

| Name | Type | Description |
|---|---|---|
| WSS | structure | WSS value. |
| tokenData | XmlElement | The XML representation of the token. |

# Appendix A: BNF definitions

## A.1 Overview over BNF

The BNF (Backus-Naur form) used in this Appendix uses `<´ and `>´ to mark symbols, `[´ and `]´ to identify optional pathes and `|´ to identify alternatives. The '(' and ')' symbols are used it indicate sets.

## A.2 BNF of RelativePath

The following BNF describes the syntax of the *RelativePath* parameter used in the TranslateBrowsePathToNodeIds and the QueryFirst *Services*.

```
<relative-path>   ::= <reference-type> <browse-name> [relative-path]

<reference-type>  ::= '/' | '.' | '<' ['!'] <browse-name> '>'

<browse-name>     ::= [<namespace-index> ':'] <name>

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

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

<name>            ::= (<name-char> | '&' <reserved-char>) [<name>]

<reserved-char>   ::= '/' | '.' | '<' | '>' | ':' | '!' | '&'

<name-char>       ::= All valid characters for a String (see [UA Part 3]) excluding reserved-chars.
```

## A.3 BNF of NumericRange

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

```
<numeric-range>   ::= <index> [':' <index>]

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

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