IEC 62541-4

Edition 2.0 2015-03

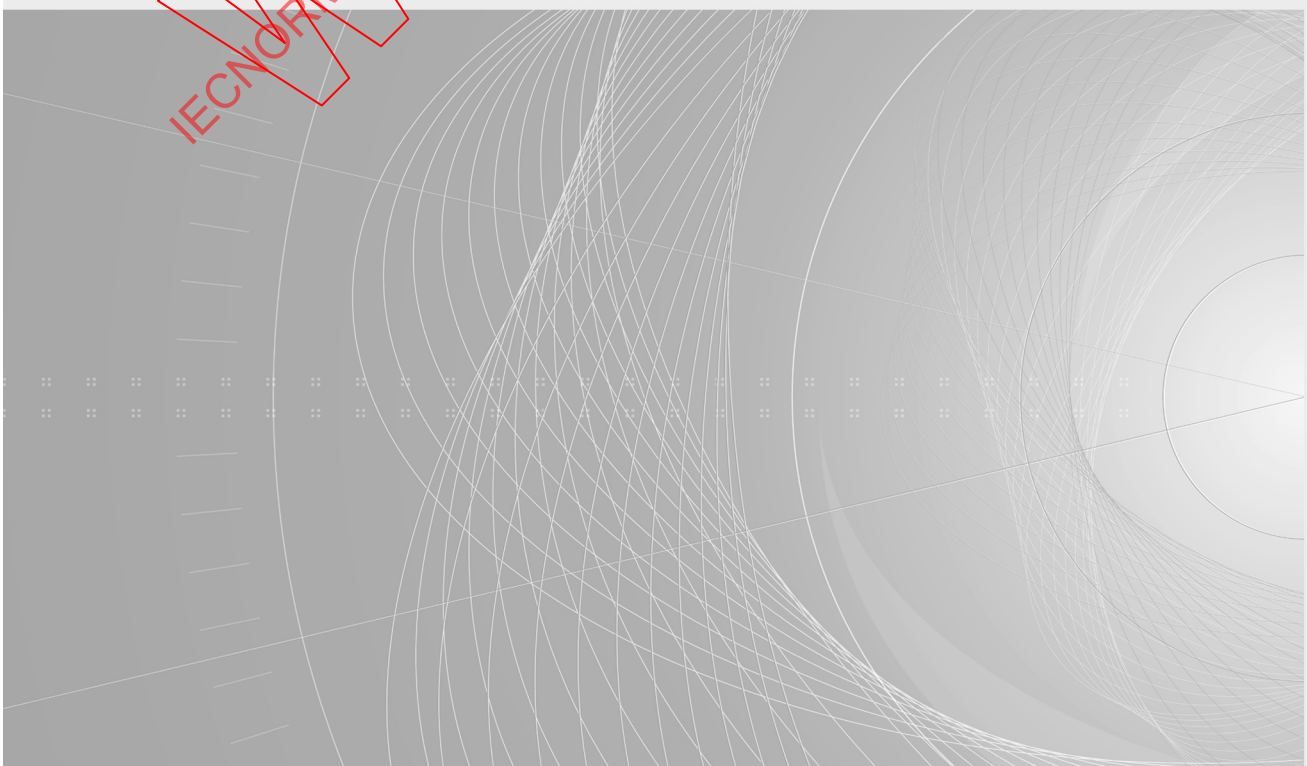
INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC Unified Architecture – Part 4: Services

Architecture Unifiée OPC – Partie 4: Services



IEC 62541-4:2015-03(en-fr)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

IEC 62541-4

Edition 2.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC Unified Architecture – Part 4: Services

Architecture Unifiée OPC – Partie 4: Services

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

ICS 25.040.40; 35.100

ISBN 978-2-8322-2369-7

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC Unified Architecture –

Part 4: Services

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 62541-4 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2011. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

a) Update for 6.4 Redundancy.
Added non-transparent redundancy option HotAndMirrored and reworked most of the redundancy description.
b) Clarifications for Publish and Reconnect scenarios.
Reworked different parts of the specificaiton to make sure no data is lost during short communication interruptions and clients can always detect for how long they lost

information during connection interuption. Added new clause 6.5 Re-establishing connections that describes the exact reconnect sequence for clients loosing connection to a server. Changed the minimum requirement for the retransmission queue of sent NotificationMessages from one keep-alive interval to minimum two times the minimum number of Publish requests per Session. Added clarification in which data value the overflow bit is set depending on the discard oldest setting. Changed discard handling for discardOldest is FALSE. The new value is replacing the last value put into the queue for FALSE. Added exception that the overflow bit is not set if the queue size one.

c) Handling of MonitoredItem changes in short network interruption scenarios.
Added new method GetMonitoredItems in Part 5. This method can be used to get the list of monitored items in a subscription if CreateMonitoredItems failed due to a network interruption and the client does not know if the creation succeeded in the server.

d) Update for 6.1.3 Determining if a Certificate is Trusted
Revised rules for certificate validation.

e) Revised definition of parameters semaphoreFile and isOnline in Service RegisterServer

f) Services ModifySubscription and ModifyMonitoredItems
Clarified that changes are applied directly and will take effect as soon as practical but not later than twice the new time interval.

g) There is a long list of minor changes to eliminate ambiguity.

The text of this standard is based on the following documents:

CDV	Report on voting
65E/375/CDV	65E/403/RVC

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The "colour inside" logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this publication using a colour printer.

OPC Unified Architecture –

Part 4: Services

1 Scope

This part of IEC 62541 defines the OPC Unified Architecture (OPC UA) *Services*. The *Services* described are the collection of abstract Remote Procedure Calls (RPC) that are implemented by OPC UA *Servers* and called by OPC UA *Clients*. All interactions between OPC UA *Clients* and *Servers* occur via these *Services*. The defined *Services* are considered abstract because no particular RPC mechanism for implementation is defined in this part. IEC 62541-6 specifies one or more concrete mappings supported for implementation. For example, one mapping in IEC 62541-6 is to XML Web Services. In that case the *Services* described in this part appear as the Web service methods in the WSDL contract.

Not all OPC UA *Servers* will need to implement all of the defined *Services*. IEC 62541-7 defines the *Profiles* that dictate which Services need to be implemented in order to be compliant with a particular *Profile*.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC TR 62541-2, *OPC Unified Architecture – Part 2: Security Model*

IEC 62541-3, *OPC unified architecture – Part 3: Address Space Model*

IEC 62541-5, *OPC unified architecture – Part 5: Information Model*

IEC 62541-6, *OPC unified architecture – Part 6: Mappings*

IEC 62541-7, *OPC unified architecture – Part 7: Profiles*

IEC 62541-8, *OPC unified architecture – Part 8: Data Access*

IEC 62541-11, *OPC Unified Architecture – Part 11: Historical Access*

IEC 62541-13 *OPC Unified Architecture – Part 13: Aggregates*

3 Terms, definitions and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1, IEC TR 62541-2, and IEC 62541-3, as well as the following apply.

3.1.1
Deadband
permitted range for value changes that will not trigger a data change *Notification*

Note 1 to entry: *Deadband* can be applied as a filter when subscribing to *Variables* and is used to keep noisy signals from updating the *Client* unnecessarily. This standard defines *AbsoluteDeadband* as a common filter. IEC 62541-8 defines an additional *Deadband* filter.

3.1.2
Endpoint
physical address available on a network that allows *Clients* to access one or more *Services* provided by a *Server*

Note 1 to entry: Each *Server* may have multiple *Endpoints*. The address of an *Endpoint* shall include a *HostName*.

3.1.3
Gateway Server
Server that acts as an intermediary for one or more *Servers*

Note 1 to entry: *Gateway Servers* may be deployed to limit external access, provide protocol conversion or to provide features that the underlying *Servers* do not support.

3.1.4
Hostname
unique identifier for a machine on a network

Note 1 to entry: This identifier shall be unique within a local network; however, it may also be globally unique. The identifier can be an IP address.

3.1.5
Security Token
identifier for a cryptographic key set

Note 1 to entry: All *Security Tokens* belong to a security context which is in case of OPC UA the *Secure Channel*.

3.1.6
SoftwareCertificate
digital certificate for a software product that can be installed on several hosts to describe the capabilities of the software product

Note 1 to entry: Different installations of one software product could have the same software certificate. Software certificates are not relevant for security. They are used to identify a software product and its supported features. SoftwareCertificates are described in 6.2.

3.2 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

NAT Network Address Translation

UA Unified Architecture

URI Uniform Resource Identifier

URL Uniform Resource Locator

3.3 Conventions for Service definitions

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

Table 1 – Service Definition Table

Name	Type	Description
Request		Defines the request parameters of the *Service*
Simple Parameter Name		Description of this parameter
Constructed Parameter Name		Description of the constructed parameter
Component Parameter Name		Description of the component parameter
Response		Defines the response parameters of the *Service*

The Name, Type and Description columns contain the name, data type and description of each parameter. All parameters are mandatory, although some may be unused under certain circumstances. The Description column specifies the value to be supplied when a parameter is unused.

Two types of parameters are defined in these tables, simple and constructed. Simple parameters have a simple data type, such as *Boolean* or *String*.

Constructed parameters are composed of two or more component parameters, which can be simple or constructed. Component parameter names are indented below the constructed parameter name.

The data types used in these tables may be base types, common types to multiple *Services* or *Service*-specific types. Base data types are defined in IEC 62541-3. The base types used in *Services* are listed in Table 2. Data types that are common to multiple *Services* are defined in Clause 7. Data types that are *Service*-specific are defined in the parameter table of the *Service*.

Table 2 – Parameter Types defined in IEC 62541-3

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

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

4 Overview

4.1 Service Set model

This clause specifies the OPC UA *Services*. The OPC UA *Service* definitions are abstract descriptions and do not represent a specification for implementation. The mapping between the abstract descriptions and the *Communication Stack* derived from these *Services* are defined in IEC 62541-6. 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 this standard and is not used in the implementation.

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

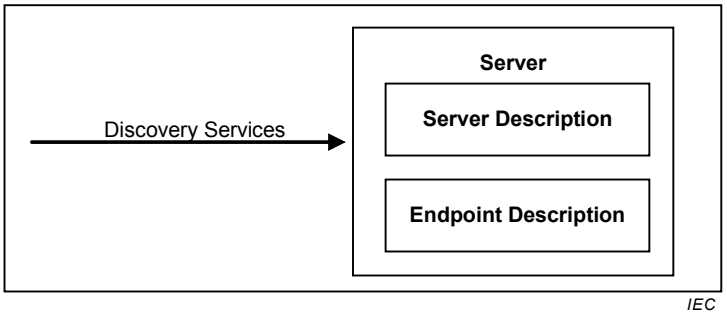


Figure 1 – Discovery Service Set

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

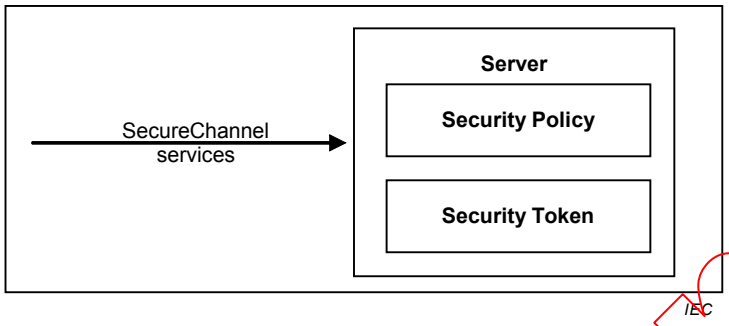


Figure 2 – SecureChannel Service Set

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

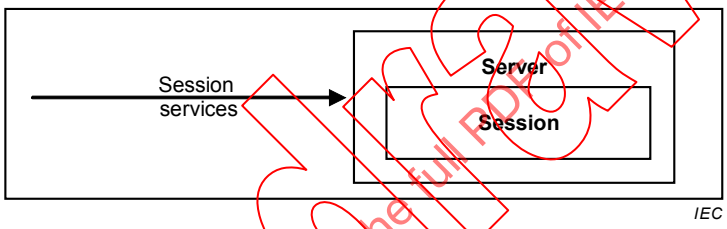


Figure 3 – Session Service Set

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

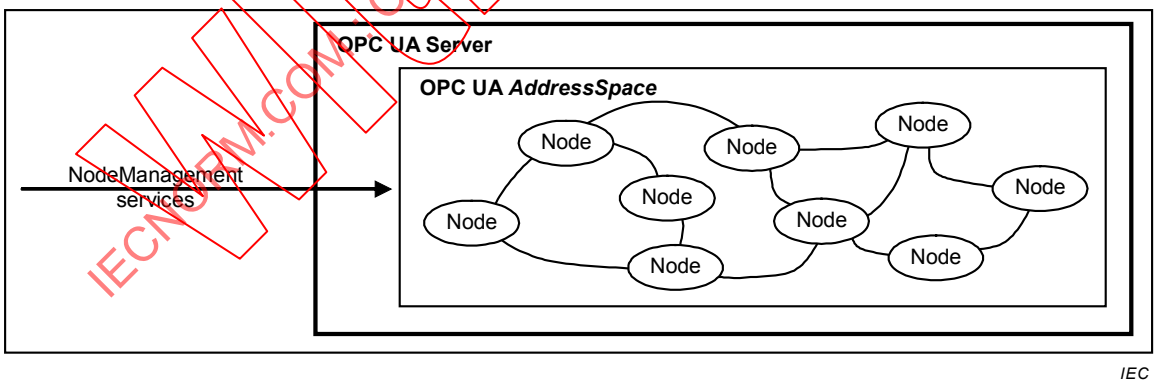


Figure 4 – NodeManagement Service Set

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

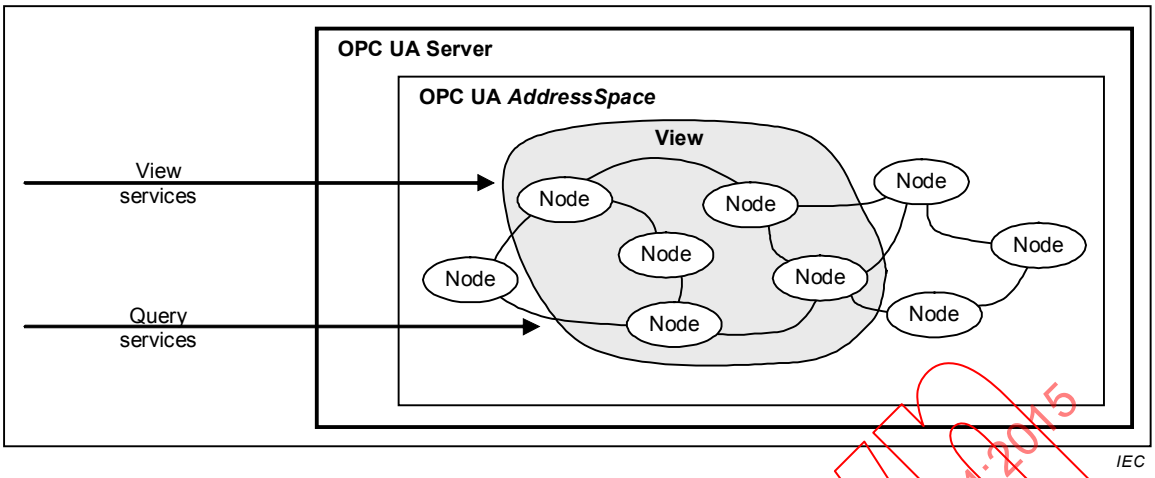


Figure 5 – View Service Set

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

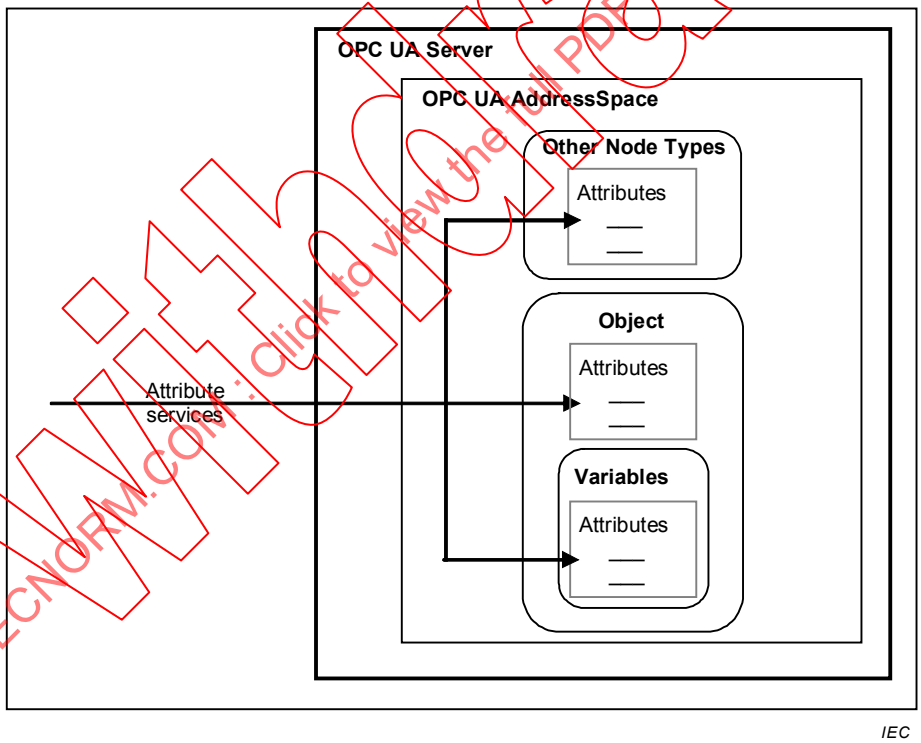


Figure 6 – Attribute Service Set

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

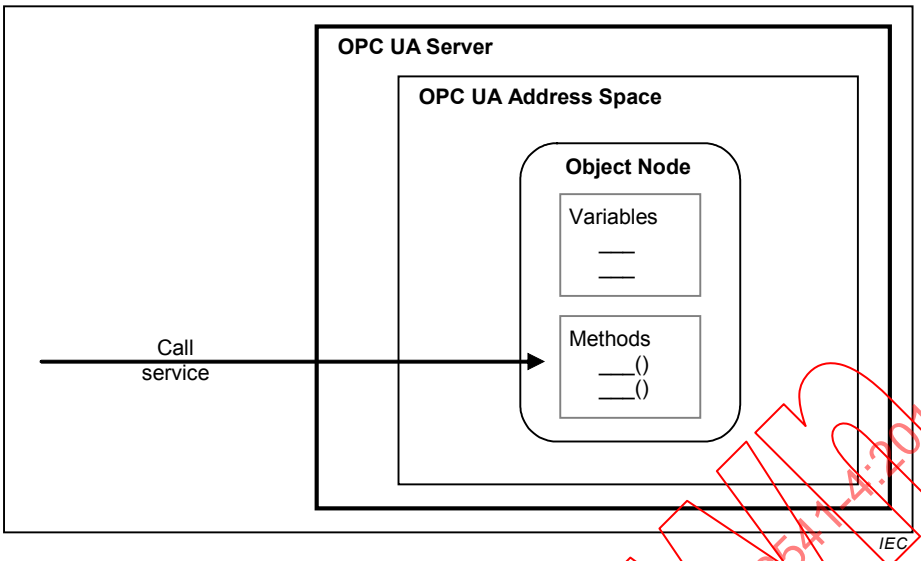


Figure 7 – Method Service Set

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

The *MonitoredItem Service Set* defines *Services* that allow *Clients* to create, modify, and delete *MonitoredItems* used to monitor *Attributes* for value changes and *Objects* for *Events*.

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

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

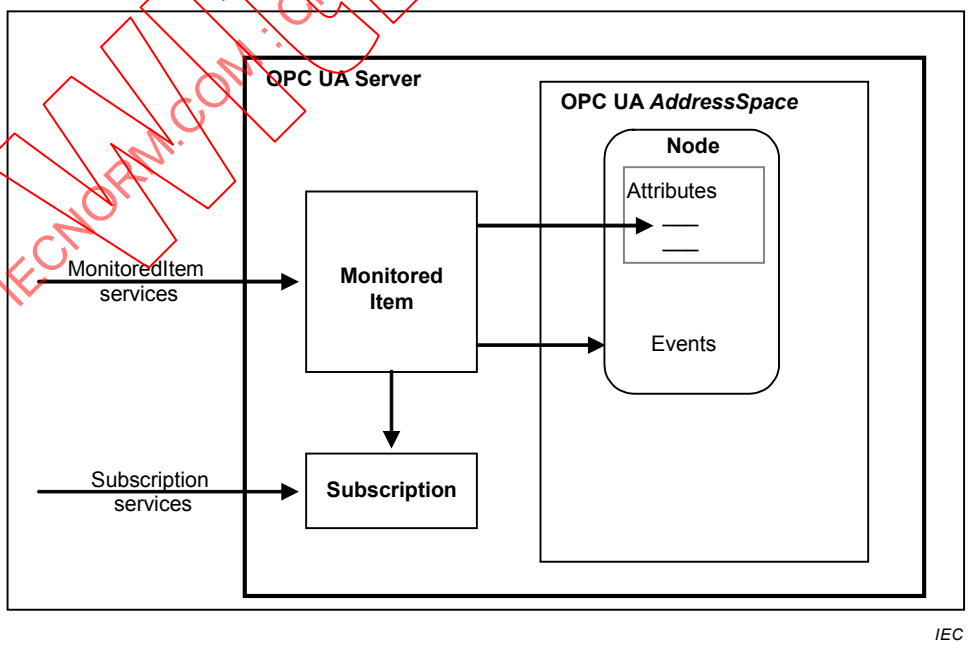


Figure 8 – MonitoredItem and Subscription Service Sets

4.2 Request/response Service procedures

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

Upon receipt of the request, the *Server* processes the *Message* in two steps. In the first step, it attempts to decode and locate the *Service* to execute. The error handling for this step is specific to the communication technology used and is described in IEC 62541-6.

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

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

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

5 Service Sets

5.1 General

This clause defines the OPC UA *Service Sets* and their *Services*. Clause 7 contains the definitions of common parameters used by these *Services*. Clause 6.2 describes auditing requirements for all services.

Whether or not a *Server* supports a *Service Set*, or a *Service* within a *Service Set*, is defined by its *Profile*. *Profiles* are described in IEC 62541-7.

5.2 Service request and response header

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

The *RequestHeader* structure is defined in 7.26 and contains common request parameters such as *authenticationToken*, *timestamp* and *requestHandle*.

The *ResponseHeader* structure is defined in 7.27 and contains common response parameters such as *serviceResult* and *diagnosticInfo*.

5.3 Service results

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

Service results are defined via the *StatusCode* (see 7.33).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see 7.27). The mechanism for returning this parameter is specific to the communication technology used to convey the *Service* response and is defined in IEC 62541-6.

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

The following cases define the use of these parameters.

a) A bad code is returned in *serviceResult* if the *Service* itself failed. In this case, a *ServiceFault* is returned. The *ServiceFault* is defined in 7.28.

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

All *Services* with arrays of operations in the request shall return a bad code in the *serviceResult* if the array is empty.

The *Services* define various specific *StatusCodes* and a *Server* shall use these specific *StatusCodes* as described in the *Service*. A *Client* should be able to handle these *Service* specific *StatusCodes*. In addition, a *Client* shall expect other common *StatusCodes* defined in Table 165 and Table 166. Additional details for *Client* handling of specific *StatusCodes* may be defined in IEC 62541-7.

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

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

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

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

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

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

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

5.4 Discovery Service Set

5.4.1 Overview

This *Service Set* defines *Services* used to discover the *Endpoints* implemented by a *Server* and to read the security configuration for those *Endpoints*. The *Discovery Services* are implemented by individual *Servers* and by dedicated *Discovery Servers*. IEC 62541-12[1] describes how to use the *Discovery Services* with dedicated *Discovery Servers.*

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

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

The complete discovery process is illustrated in Figure 9.

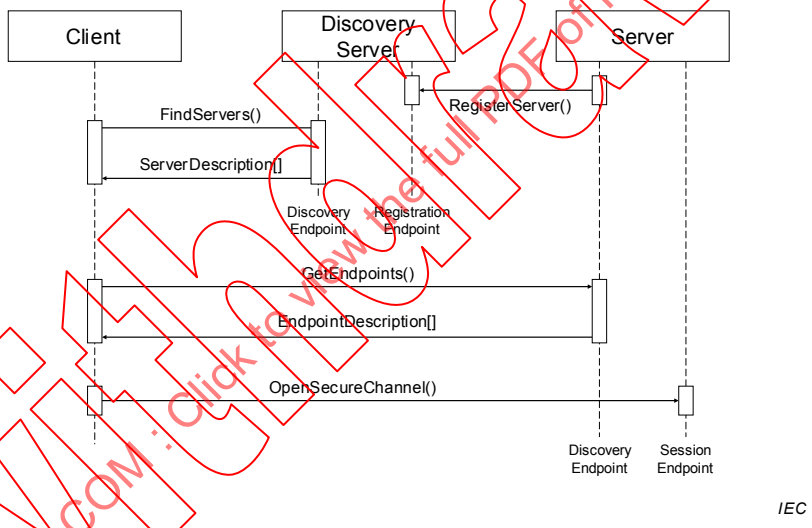


Figure 9 – Discovery process

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

Once a *Client* retrieves the *Endpoints*, the C*lient* can save this information and use it to connect directly to the *Server* again without going through the discovery process. If the *Client* finds that it cannot connect then the *Server* configuration may have changed and the *Client* needs to go through the discovery process again.

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

---

1 Under consideration.

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

a) The *ApplicationUri* specified in the *Server Certificate* is the same as the ApplicationUri provided in the *EndpointDescription.*

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

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

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

A *Client* shall verify the *HostName* specified in the *Server Certificate* is the same as the *HostName* contained in the *endpointUrl* provided in the *EndpointDescription* returned by *CreateSession*. If there is a difference then the *Client* shall report the difference and may close the *SecureChannel*. *Servers* shall add all possible *HostNames* like MyHost and MyHost.local into the *Server Certificate*. This includes IP addresses of the host or the *HostName* exposed by a NAT router used to connect to the *Server*.

5.4.2 FindServers

5.4.2.1 Description

This *Service* returns the *Servers* known to a *Server* or *Discovery Server*. The behaviour of *Discovery Servers* is described in detail in IEC 62541-12[2].

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

Every *Server* shall provide a *Discovery Endpoint* that supports this *Service*. The *Server* shall always return a record that describes itself, however in some cases more than one record may be returned. *Gateway Servers* shall return a record for each *Server* that they provide access to plus (optionally) a record that allows the *Gateway Server* to be accessed as an ordinary OPC UA *Server*. Non-transparent redundant *Servers* shall provide a record for each *Server* in the redundant set.

Every *Server* shall have a globally unique identifier called the *ServerUri*. This identifier should be a fully qualified domain name; however, it may be a GUID or similar construct that ensures global uniqueness. The *ServerUri* returned by this *Service* shall be the same value that appears in index 0 of the *ServerArray* property (see IEC 62541-5). The *ServerUri* is returned as the *applicationUri* field in the *ApplicationDescription* (see 7.1)

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

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

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

2 Under consideration.

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

This *Service* can be used without security and it is therefore vulnerable to Denial Of Service (DOS) attacks. A *Server* should minimize the amount of processing required to send the response for this *Service*. This can be achieved by preparing the result in advance. The *Server* should also add a short delay before starting processing of a request during high traffic conditions.

The *DiscoveryUrl* returned by this *Service* is ambiguous if there are multiple *TransportProfiles* (e.g. UA XML or UA Binary encoding) associated with the URL scheme. *Clients* that support multiple *TransportProfiles* should attempt to use alternate *TransportProfiles* if the first choice does not succeed.

5.4.2.2 Parameters

Table 3 defines the parameters for the *Service*.

Table 3 – FindServers Service Parameters

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

5.4.2.3 Service results

Common *StatusCodes* are defined in Table 165.

5.4.3 GetEndpoints

5.4.3.1 Description

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

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

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

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

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

Server Application Instance Certificate

Message Security Mode

Security Policy

Supported User Identity Tokens

The *ApplicationInstanceCertificate* is used to secure the *OpenSecureChannel* request (see 5.5.2). The *MessageSecurityMode* and the *SecurityPolicy* tell the *Client* how to secure messages sent via the *SecureChannel*. The *UserIdentityTokens* tell the client which type of user credentials shall be passed to the *Server* in the *ActivateSession* request (see 5.6.3).

If the *securityPolicyUri* is NONE and none of the *UserTokenPolicies* requires encryption, the *Client* shall ignore the *ApplicationInstanceCertificate*.

Each *EndpointDescription* also specifies a URI for the *Transport Profile* that the *Endpoint* supports. The *Transport Profiles* specify information such as message encoding format and protocol version and are defined in IEC 62541-7. *Clients* shall fetch the *Server's SoftwareCertificates* if they want to discover the complete list of *Profiles* supported by the *Server* (see 7.30).

Messages are secured by applying standard cryptography algorithms to the messages before they are sent over the network. The exact set of algorithms used depends on the *SecurityPolicy* for the *Endpoint*. IEC 62541-7 defines *Profiles* for common *SecurityPolicies* and assigns a unique URI to them. It is expected that applications have built in knowledge of the *SecurityPolicies* that they support, as a result, only the Profile URI for the *SecurityPolicy* is specified in the *EndpointDescription*. A *Client* cannot connect to an *Endpoint* that does not support a *SecurityPolicy* that it recognizes.

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

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

This *Service* can be used without security and it is therefore vulnerable to Denial Of Service (DOS) attacks. A *Server* should minimize the amount of processing required to send the response for this *Service*. This can be achieved by preparing the result in advance. The *Server* should also add a short delay before starting processing of a request during high traffic conditions.

Some of the *EndpointDescriptions* returned in a response shall specify the *Endpoint* information for a *Gateway Server* that can be used to access another *Server*. In these situations, the *gatewayServerUri* is specified in the *EndpointDescription* and all security

checks used to verify *Certificates* shall use the *gatewayServerUri* (see 6.1.3) instead of the *serverUri*.

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

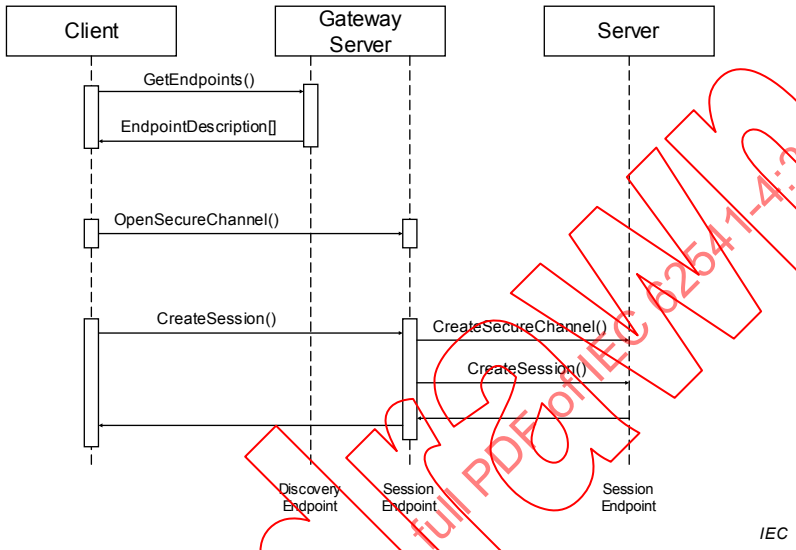


Figure 10 – Using a Gateway Server

5.4.3.2 Parameters

Table 4 defines the parameters for the *Service*.

Table 4 – GetEndpoints Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The *authenticationToken* shall be ignored if it is provided. The type *RequestHeader* is defined in 7.26.
endpointUrl	String	The network address that the *Client* used to access the *Discovery Endpoint*. The *Server* uses this information for diagnostics and to determine what URLs to return in the response. The *Server* should return a suitable default URL if it does not recognize the *HostName* in the URL.
localeIds []	LocaleId	List of locales to use. Specifies the locale to use when returning human readable strings. This parameter is described in 5.4.2.2.
profileUris []	String	List of *Transport Profile* that the returned *Endpoints* shall support. IEC 62541-7 defines URIs for the *Transport Profiles*. All *Endpoints* are returned if the list is empty.
Response		
responseHeader	ResponseHeader	Common response parameters. The *ResponseHeader* type is defined in 7.27.
Endpoints []	EndpointDescription	List of *Endpoints* that meet criteria specified in the request. This list is empty if no *Endpoints* meet the criteria. The *EndpointDescription* type is defined in 7.9.

5.4.3.3 Service Results

Common *StatusCodes* are defined in Table 165.

5.4.4 RegisterServer

5.4.4.1 Description

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

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

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

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

Servers shall be able to register themselves with a *Discovery Server* running on the same machine. The exact mechanisms depend on the *Discovery Server* implementation and are described in IEC 62541-6.

There are two types of *Server* applications: those which are manually launched including a start by the operating system at boot and those that are automatically launched when a *Client* attempts to connect. The registration process that a *Server* shall use depends on which category it falls into.

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

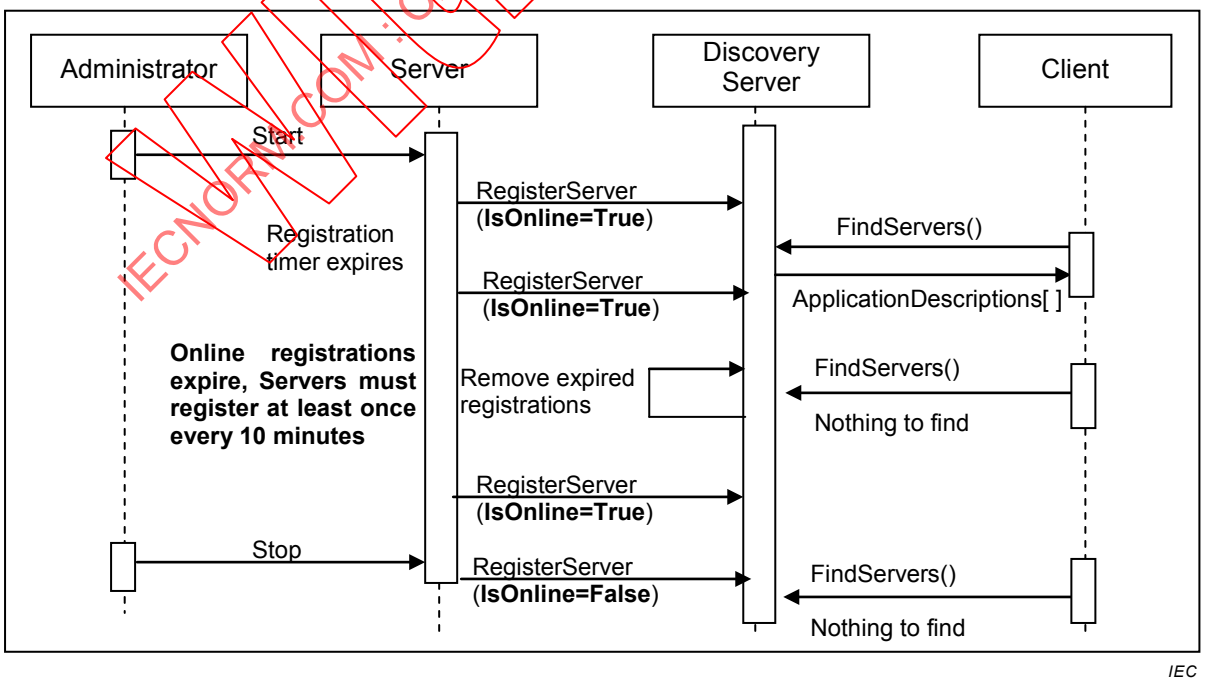


Figure 11 – The Registration Process – Manually Launched Servers

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

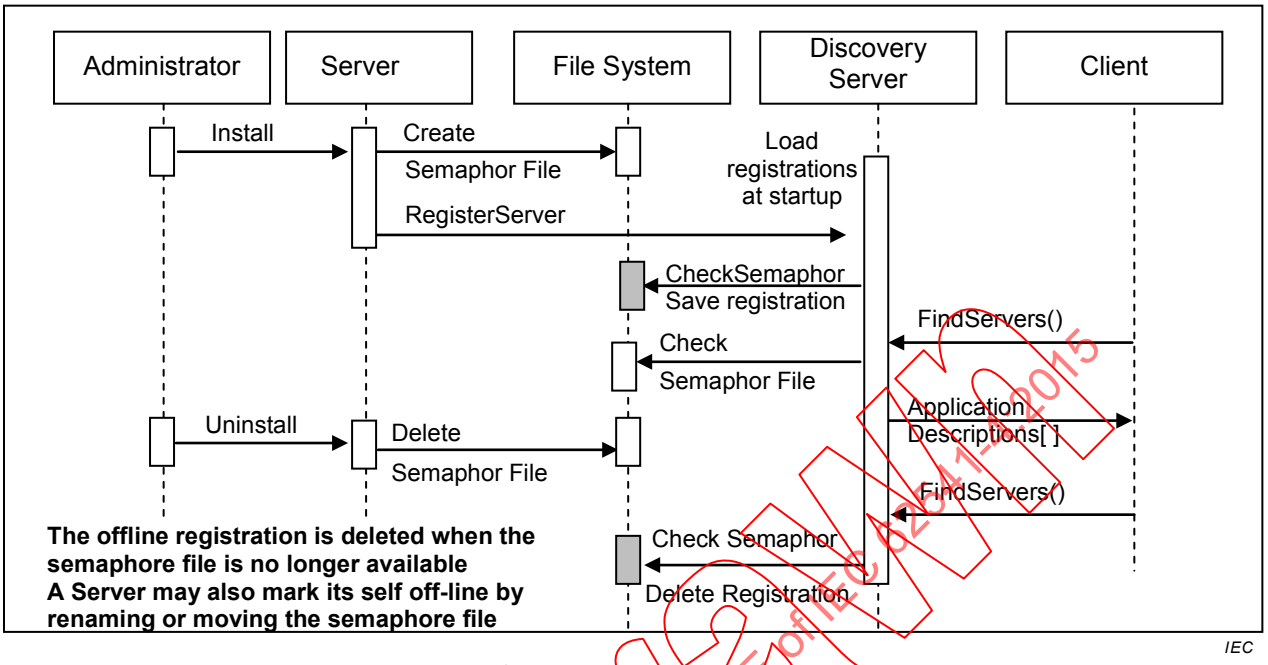


Figure 12 – The Registration Process – Automatically Launched Servers

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

Under normal conditions, manually launched *Servers* shall periodically register with the *Discovery Server* as long as they are able to receive connections from *Clients*. If a *Server* goes offline then it shall register itself once more and indicate that it is going offline. The registration frequency should be configurable; however, the maximum is 10 minutes. If an error occurs during registration (e.g. the *Discovery Server* is not running) then the *Server* shall periodically re-attempt registration. The frequency of these attempts should start at 1 second but gradually increase until the registration frequency is the same as what it would be if no errors occurred. The recommended approach would be to double the period of each attempt until reaching the maximum.

When an automatically launched *Server* (or its install program) registers with the *Discovery Server* it shall provide a path to a semaphore file which the *Discovery Server* can use to determine if the *Server* has been uninstalled from the machine. The *Discovery Server* shall have read access to the file system that contains the file.

5.4.4.2 Parameters

Table 5 defines the parameters for the *Service*.

Table 5 – RegisterServer Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
Server	RegisteredServer	The server to register. This structure is defined in-line with the following indented items.
serverUri	String	The globally unique identifier for the *Server* instance. The *serverUri* matches the *applicationUri* from the *ApplicationDescription* defined in 7.1.
productUri	String	The globally unique identifier for the *Server* product.
serverNames []	LocalizedText	A list of localized descriptive names for the *Server*. The list shall have at least one valid entry.
serverType	Enum ApplicationType	The type of application. The enumeration values are defined in Table 103. The value "CLIENT_1" (The application is a *Client)* is not allowed. The *Service* result shall be Bad_InvalidArgument in this case.
gatewayServerUri	String	The URI of the *Gateway Server* associated with the *discoveryUrls*. This value is only specified by *Gateway Servers* that wish to register the *Servers* that they provide access to. For *Servers* that do not act as a *Gateway Server* this parameter shall be null.
discoveryUrls []	String	A list of *Discovery Endpoints* for the *Server*. The list shall have at least one valid entry.
semaphoreFilePath	String	The path to the semaphore file used to identify an automatically-launched server instance; Manually-launched servers will not use this parameter. If a Semaphore file is provided, the isOnline flag is ignored. If a Semaphore file is provided and exists, the *LocalDiscoveryServer* shall save the registration information in a persistent data store that it reads whenever the *LocalDiscoveryServer* starts. If a Semaphore file is specified but does not exist the *Discovery Server* shall remove the registration from any persistent data store. If the Server has registered with a semaphoreFilePath, the *DiscoveryServer* shall check that this file exists before returning the *ApplicationDescription* to the client. If the Server did not register with a semaphoreFilePath (it is null or empty) then the *DiscoveryServer* does not attempt to verify the existence of the file before returning the ApplicationDescription to the client.
isOnline	Boolean	True if the *Server* is currently able to accept connections from *Clients*. The *DiscoveryServer* shall return *ApplicationDescriptions* to the *Client*. The Server is expected to periodically re-register with the *DiscoveryServer*. False if the *Server* is currently unable to accept connections from *Clients*. The *DiscoveryServer* shall NOT return *ApplicationDescriptions* to the *Client*. This parameter is ignored if a semaphoreFilePath is provided.
Response		
ResponseHeader	ResponseHeader	Common response parameters. The type *ResponseHeader* is defined in 7.27.

5.4.4.3 Service Results

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

Table 6 – RegisterServer Service Result Codes

Symbolic Id	Description
Bad_InvalidArgument	See Table 165 for the description of this result code.
Bad_ServerUriInvalid	See Table 165 for the description of this result code.
Bad_ServerNameMissing	No *ServerName* was specified.
Bad_DiscoveryUrlMissing	No discovery URL was specified.
Bad_SempahoreFileMissing	The semaphore file specified is not valid.

5.5 SecureChannel Service Set

5.5.1 Overview

This *Service Set* defines *Services* used to open a communication channel that ensures the confidentiality and *Integrity* of all *Messages* exchanged with the *Server*. The base concepts for OPC UA security are defined in IEC TR 62541-2.

The *SecureChannel Services* are unlike other *Services* because they are not implemented directly by the *OPC UA Application*. Instead, they are provided by the *Communication Stack* on which the *OPC UA Application* is built. For example, an OPC UA *Server* may be built on a SOAP stack that allows applications to establish a *SecureChannel* using the WS Secure Conversation specification. In these cases, the *OPC UA Application* shall verify that the *Message* it received was in the context of a WS Secure Conversation. IEC 62541-6 describes how the *SecureChannel Services* are implemented.

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

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

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

It should be noted that some *SecurityPolicies* defined in IEC 62541-7 will turn off authentication and encryption resulting in a *SecureChannel* that provides no security.

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

The relationship between the *SecureChannel* and the *OPC UA Application* depends on the implementation technology. IEC 62541-6 defines any requirements that depend on the technology used.

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

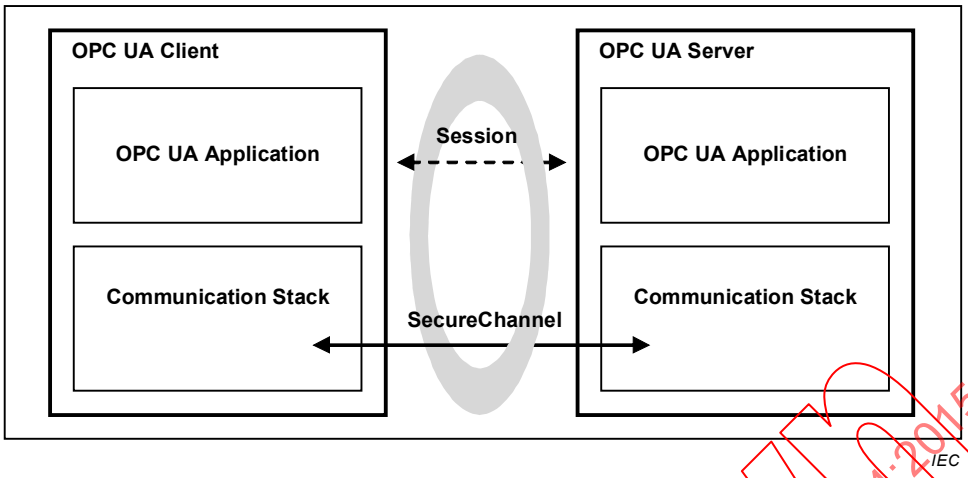


Figure 13 – SecureChannel and Session Services

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

If a *Server* acts as a *Client* to other *Servers*, which is commonly referred to as *Server* chaining, then the Server shall be able to maintain user level security. By this we mean that the user identity should be passed to the underlying server or it should be mapped to an appropriate user identity in the underlying server. It is unacceptable to ignore user level security. This is required to ensure that security is maintained and that a user does not obtain information that they should not have access to. Whenever possible a *Server* should impersonate the original *Client* by passing the original *Client's* user identity to the underlying *Server* when it calls the *ActivateSession Service*. If impersonation is not an option then the *Server* shall map the original *Client's* user identity onto a new user identity which the underlying *Server* does recognize.

5.5.2 OpenSecureChannel

5.5.2.1 Description

This *Service* is used to open or renew a *SecureChannel* that can be used to ensure *Confidentiality* and *Integrity* for *Message* exchange during a *Session*. This *Service* requires the *Communication Stack* to apply the various security algorithms to the *Messages* as they are sent and received. Specific implementations of this *Service* for different *Communication Stacks* are described in IEC 62541-6.

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:

a) *Clients* should request a new *SecurityToken* after 75 % of its lifetime has elapsed. This should ensure that *Clients* will receive the new *SecurityToken* before the old one actually expires.

b) *Servers* shall use the existing *SecurityToken* to secure outgoing *Messages* until the *SecurityToken* expires or the *Server* receives a *Message* secured with a new *SecurityToken*. This should ensure that *Clients* do not reject *Messages* secured with the new *SecurityToken* that arrive before the *Client* receives the new *SecurityToken*.

c) *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. A *Server* application should limit the number of *SecureChannels*. To protect against misbehaving Clients and denial of service attacks, the Server shall close the oldest *SecureChannel* that has no *Session* assigned before reaching the maximum number of supported *SecureChannels*.

The *OpenSecureChannel* request and response *Messages* shall be signed with the sender's *Certificate*. These *Messages* shall always be encrypted. If the transport layer does not provide encryption, then these *Messages* shall be encrypted with the receiver's *Certificate*. These requirements for *OpenSecureChannel* only apply if the *securityPolicyUri* is not None.

The *Certificates* used in the *OpenSecureChannel* service shall be the *Application Instance Certificates*. *Clients* and *Servers* shall verify that the same *Certificates* were used in the *CreateSession* and *ActivateSession* services. Certificates are not provided and shall not be verified if the *securityPolicyUri* is None.

If the *securityPolicyUri* is not None, a *Client* shall verify the *HostName* specified in the *Server Certificate* is the same as the *HostName* contained in the *endpointUrl*. If there is a difference then the *Client* shall report the difference and may choose to not open the *SecureChannel*. *Servers* shall add all possible *HostNames* like MyHost and MyHost.local into the *Server Certificate*. This includes IP addresses of the host or the *HostName* exposed by a NAT router used to connect to the *Server*.

Clients should be prepared to replace the *HostName* returned in the *EndPointDescription* with the *HostName* or the IP addresses they used to call *GetEndpoints*.

5.5.2.2 Parameters

Table 7 defines the parameters for the *Service*.

Unlike other *Services*, the parameters for this *Service* provide only an abstract defintion. The concrete representation on the network depends on the mappings defined in IEC 62541-6.

Table 7 – OpenSecureChannel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
clientCertificate	ApplicationInstance Certificate	A *Certificate* that identifies the *Client*. The *OpenSecureChannel* request shall be signed with this *Certificate*. The *ApplicationInstanceCertificate* type is defined in 7.2. If the *securityPolicyUri* is None, the *Server* shall ignore the *ApplicationInstanceCertificate*.
requestType	Enum SecurityToken RequestType	The type of *SecurityToken* request: An enumeration that shall be one of the following: ISSUE_0 creates a new *SecurityToken* for a new *SecureChannel*. RENEW_1 creates a new *SecurityToken* for an existing *SecureChannel*.
secureChannelId	ByteString	The identifier for the *SecureChannel* that the new token should belong to. This parameter shall be null when creating a new *SecureChannel*.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type *MessageSecurityMode* type is defined in 7.14. A *SecureChannel* may have to be created even if the *securityMode* is NONE. The exact behaviour depends on the mapping used and is described in the IEC 62541-6.
securityPolicyUri	String	The URI for *SecurityPolicy* to use when securing messages sent over the *SecureChannel*. The set of known URIs and the *SecurityPolicies* associated with them are defined in IEC 62541-7.
clientNonce	ByteString	A random number that shall not be used in any other request. A new *clientNonce* shall be generated for each time a *SecureChannel* is renewed. This parameter shall have a length equal to key size used for the symmetric encryption algorithm that is identified by the *securityPolicyUri*.
requestedLifetime	Duration	The requested lifetime, in milliseconds, for the new *SecurityToken*. It specifies when the *Client* expects to renew the *SecureChannel* by calling the *OpenSecureChannel Service* again. If a *SecureChannel* is not renewed, then all *Messages* sent using the current *SecurityTokens* shall be rejected by the receiver.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type definition).
securityToken	ChannelSecurityToken	Describes the new *SecurityToken* issued by the *Server*. This structure is defined in-line with the following indented items.
channelId	ByteString	A unique identifier for the *SecureChannel*. This is the identifier that shall be supplied whenever the *SecureChannel* is renewed.
tokenId	ByteString	A unique identifier for a single *SecurityToken* within the channel. This is the identifier that shall be passed with each *Message* secured with the *SecurityToken*.
createdAt	UtcTime	The time when the *SecurityToken* was created.
revisedLifetime	Duration	The lifetime of the *SecurityToken* in milliseconds. The UTC expiration time for the token may be calculated by adding the lifetime to the *createdAt* time.
serverNonce	ByteString	A random number that shall not be used in any other request. A new *serverNonce* shall be generated for each time a *SecureChannel* is renewed. This parameter shall have a length equal to key size used for the symmetric encryption algorithm that is identified by the *securityPolicyUri*.

5.5.2.3 Service results

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

Table 8 – OpenSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecurityChecksFailed	See Table 165 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 165 for the description of this result code.
Bad_CertificateUriInvalid	See Table 165 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateUntrusted	See Table 165 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateRevoked	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 165 for the description of this result code.
Bad_RequestTypeInvalid	The security token request type is not valid.
Bad_SecurityModeRejected	The security mode does not meet the requirements set by the *Server*.
Bad_SecurityPolicyRejected	The security policy does not meet the requirements set by the *Server*.
Bad_SecureChannelIdInvalid	See Table 165 for the description of this result code.
Bad_NonceInvalid	See Table 165 for the description of this result code. A server shall check the minimum length of the client nonce and return this status if the length is below 32 bytes. A check for duplicated nonces can only be done in OpenSecureChannel calls with the request type RENEW_1.

5.5.3 CloseSecureChannel

5.5.3.1 Description

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

The request *Messages* shall be signed with the appropriate key associated with the current token for the *SecureChannel*.

5.5.3.2 Parameters

Table 9 defines the parameters for the *Service*.

Table 9 – CloseSecureChannel Service Parameters

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

5.5.3.3 Service results

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

Table 10 – CloseSecureChannel Service Result Codes

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

5.6 Session Service Set

5.6.1 Overview

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

5.6.2 CreateSession

5.6.2.1 Description

This *Service* is used by an OPC UA *Client* to create a *Session* and the *Server* returns two values which uniquely identify the *Session*. The first value is the *sessionId* which is used to identify the *Session* in the audit logs and in the *Server's* address space. The second is the *authenticationToken* which is used to associate an incoming request with a *Session*.

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

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

Depending upon on the *SecurityPolicy* and the *SecurityMode* of the *SecureChannel,* the exchange of *ApplicationInstanceCertificates* and *Nonces* may be optional and the signatures may be empty. See IEC 62541-7 for the definition of *SecurityPolicies* and the handling of these parameters.

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

The *Session* created with this *Service* shall not be used until the *Client* calls the *ActivateSession Service* and provides its *SoftwareCertificates* and proves possession of its *Application Instance Certificate* and any user identity token that it provided.

The *SoftwareCertificates* parameter in the *Server* response is deprecated to reduce the message size for OPC UA Applications with limited resources. The *SoftwareCertificates* are provided in the *Server's AddressSpace* as defined in IEC 62541-5. A *SoftwareCertificate* identifies the capabilities of the *Server* and also contains the list of OPC UA *Profiles* supported by the *Server*. OPC UA *Profiles* are defined in IEC 62541-7.

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 *SessionDiagnosticsArray Variable*. See IEC 62541-5 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 as described in IEC 62541-6.

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

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

If a *Client* invokes the *CloseSession Service* then all *Subscriptions* associated with the *Session* are also deleted if the *deleteSubscriptions* flag is set to TRUE. If a *Server* terminates a *Session* for any other reason, *Subscriptions* associated with the *Session*, are not deleted. Each *Subscription* has its own lifetime to protect against data loss in the case of a *Session* termination. In these cases, the *Subscription* can be reassigned to another *Client* before its lifetime expires.

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

First, each OPC UA *Service* request contains a string parameter that is used to carry an audit record id. A *Client*, or any *Server* operating as a *Client*, such as an aggregating *Server*, can create a local audit log entry for a request that it submits. This parameter allows the *Client* to pass the identifier for this entry with the request. If the *Server* also maintains an audit log, then 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 IEC TR 62541-2 for additional information on auditing. A *Server* that maintains an audit log shall provide the information in the audit log entries via event *Messages* defined in this standard. The *Server* may choose to only provide the *Audit* information via event *Messages*. The *Audit EventType* is defined in IEC 62541-3.

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

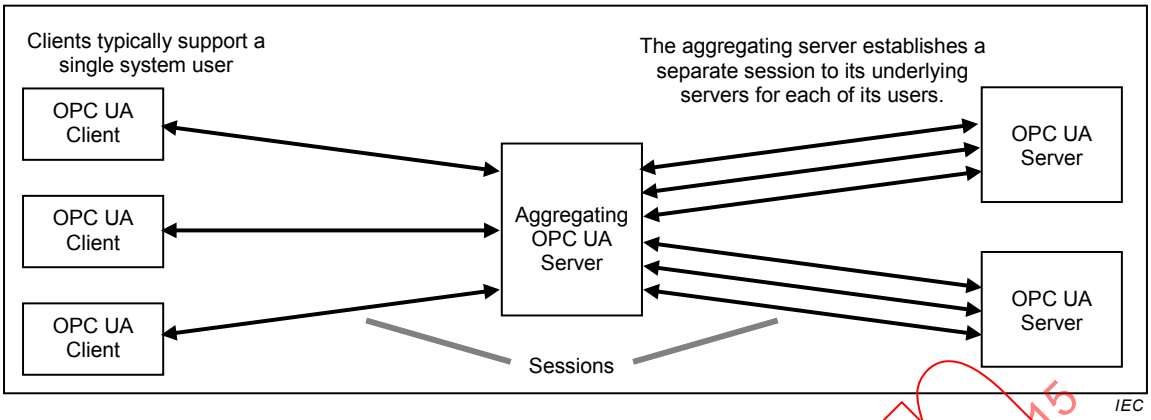


Figure 14 – Multiplexing Users on a Session

5.6.2.2 Parameters

Table 11 defines the parameters for the *Service*.

Table 11 – CreateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The *authenticationToken* is always omitted. The type *RequestHeader* is defined in 7.26.
clientDescription	Application Description	Information that describes the *Client* application. The type *ApplicationDescription* is defined in 7.1.
serverUri	String	This value is only specified if the *EndpointDescription* has a *gatewayServerUri*. This value is the *applicationUri* from the *EndpointDescription* which is the *applicationUri* for the underlying *Server*. The type *EndpointDescription* is defined in 7.9.
endpointUrl	String	The network address that the *Client* used to access the *Session Endpoint*. The *HostName* portion of the URL should be one of the *HostNames* for the application that are specified in the *Server's ApplicationInstanceCertificate* (see 7.2). The *Server* shall raise an *AuditUrlMismatchEventType* event if the URL does not match the Server's *HostNames*. *AuditUrlMismatchEventType* event type is defined in IEC 62541-5. The Server uses this information for diagnostics and to determine the set of *EndpointDescriptions* to return in the response.
sessionName	String	Human readable string that identifies the *Session*. The *Server* makes this name and the *sessionId* visible in its *AddressSpace* for diagnostic purposes. The *Client* should provide a name that is unique for the instance of the *Client*. If this parameter is not specified the *Server* shall assign a value.
clientNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. Profiles may increase the required length. The *Server* shall use this value to prove possession of its *Application Instance Certificate* in the response.
clientCertificate	ApplicationInstance Certificate	The *Application Instance Certificate* issued to the *Client*. The *ApplicationInstanceCertificate* type is defined in 7.2. If the *securityPolicyUri* is None, the *Server* shall ignore the *ApplicationInstanceCertificate*.
Requested SessionTimeout	Duration	Requested maximum number of milliseconds that a *Session* should remain open without activity. If the *Client* fails to issue a *Service* request within this interval, then the *Server* shall automatically terminate the *Client Session*.
maxResponse MessageSize	UInt32	The maximum size, in bytes, for the body of any response message. The *Server* should return a *Bad_ResponseTooLarge* service fault if a response message exceeds this limit. The value zero indicates that this parameter is not used. More information on the use of this parameter is provided in 5.3.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type).
sessionId	NodeId	A unique *NodeId* assigned by the *Server* to the *Session*. This identifier is used to access the diagnostics information for the *Session* in the *Server* address space. It is also used in the audit logs and any events that report information related to the *Session*. The *Session* diagnostic information is described in IEC 62541-5. Audit logs

Name	Type	Description
		and their related events are described in 6.2
authentication Token	Session AuthenticationToken	A unique identifier assigned by the *Server* to the *Session*. This identifier shall be passed in the *RequestHeader* of each request and is used with the *SecureChannelId* to determine whether a *Client* has access to the *Session*. This identifier shall not be reused in a way that the *Client* or the *Server* has a chance of confusing them with a previous or existing *Session*. The *SessionAuthenticationToken* type is described in 7.29.
revisedSession Timeout	Duration	Actual maximum number of milliseconds that a *Session* shall remain open without activity. The *Server* should attempt to honour the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The *Client* shall use this value to prove possession of its *Application Instance Certificate* in the *ActivateSession* request. This value may also be used to prove possession of the *userIdentityToken* it specified in the *ActivateSession* request.
serverCertificate	ApplicationInstance Certificate	The *Application Instance Certificate* issued to the *Server*. A *Server* shall prove possession by using the private key to sign the *Nonce* provided by the *Client* in the request. The *Client* shall verify that this *Certificate* is the same as the one it used to create the *SecureChannel*. The *ApplicationInstanceCertificate* type is defined in 7.2. If the *securityPolicyUri* is NONE and none of the *UserTokenPolicies* requires encryption, the *Client* shall ignore the *ApplicationInstanceCertificate*.
serverEndpoints []	Endpoint Description	List of *Endpoints* that the server supports. The *Server* shall return a set of *EndpointDescriptions* available for the *serverUri* specified in the request. The *EndpointDescription* type is defined in 7.9. The *Client* shall verify this list with the list from a *Discovery Endpoint* if it used a *Discovery Endpoint* to fetch the *EndpointDescriptions*. It is recommended that *Servers* only include the *endpointUrl*, *securityMode*, *securityPolicyUri*, *userIdentityTokens*, *transportProfileUri* and *securityLevel* with all other parameters set to null. Only the recommended parameters shall be verified by the client.
serverSoftware Certificates []	SignedSoftware Certificate	This parameter is deprecated and the array shall be empty. The *SoftwareCertificates* are provided in the *Server AddressSpace* as defined in IEC 62541-5.
serverSignature	SignatureData	This is a signature generated with the private key associated with the *serverCertificate*. This parameter is calculated by appending the *clientNonce* to the *clientCertificate* and signing the resulting sequence of bytes. The *SignatureAlgorithm* shall be the *AsymmetricSignatureAlgorithm* specified in the *SecurityPolicy* for the *Endpoint*. The *SignatureData* type is defined in 7.30.
maxRequest MessageSize	UInt32	The maximum size, in bytes, for the body of any request message. The *Client Communication Stack* should return a *Bad_RequestTooLarge* error to the application if a request message exceeds this limit. The value zero indicates that this parameter is not used. 5.3 provides more information on the use of this parameter.

5.6.2.3 Service results

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

Table 12 – CreateSession Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 165 for the description of this result code.
Bad_NonceInvalid	See Table 165 for the description of this result code. A server shall check the minimum length of the client nonce and return this status if the length is below 32 bytes. A check for duplicated nonces is optional and requires access to the nonce used to create the secure channel.
Bad_SecurityChecksFailed	See Table 165 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 165 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 165 for the description of this result code.
Bad_CertificateUriInvalid	See Table 165 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 165 for the description of this result code.
Bad_CertificateUntrusted	See Table 165 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 165 for the description of this result code.
Bad_CertificateRevoked	See Table 165 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 165 for the description of this result code.
Bad_TooManySessions	The server has reached its maximum number of sessions.
Bad_ServerUriInvalid	See Table 165 for the description of this result code.

5.6.3 ActivateSession

5.6.3.1 Description

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

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

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

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

The *ActivateSession Service* is used to associate a user identity with a *Session*. When a *Client* provides a user identity then it shall provide proof that it 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 is included in the token. If the token is an X509IdentityToken then the proof is a signature generated with private key associated with the *Certificate*. The data to sign is created by appending the last *serverNonce* to the *serverCertificate* specified in the *CreateSession* response. If a token includes a secret then it should be encrypted using the public key from the *serverCertificate.*

Clients can change the identity of a user associated with a *Session* by calling the *ActivateSession Service*. The *Server* validates the signatures provided with the request and then validates the new user identity. If no errors occur the *Server* replaces the user identity for the *Session*. Changing the user identity for a *Session* may cause discontinuities in active *Subscriptions* because the *Server* may have to tear down connections to an underlying system and re-establish 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 a *LocalizedText* in the context of the *Session*, 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, then 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, then 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, then the *Server* returns the string that it has along with the locale id.

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

5.6.3.2 Parameters

Table 13 defines the parameters for the *Service*.

Table 13 – ActivateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The type *RequestHeader* is defined in 7.26.
clientSignature	SignatureData	This is a signature generated with the private key associated with the *clientCertificate*. The *SignatureAlgorithm* shall be the *AsymmetricSignatureAlgorithm* specified in the *SecurityPolicy* for the *Endpoint*. The *SignatureData* type is defined in 7.30.
clientSoftwareCertificates []	SignedSoftware Certificate	These are the *SoftwareCertificates* which have been issued to the *Client* application. The *productUri* contained in the *SoftwareCertificates* shall match the *productUri* in the Application*Description* passed by the *Client* in the *CreateSession* requests. *Certificates* without matching *productUri* should be ignored. *Servers* may reject connections from *Clients* if they are not satisfied with the *SoftwareCertificates* provided by the *Client*. This parameter only needs to be specified in the first *ActivateSession* request after *CreateSession*. It shall always be omitted if the *maxRequestMessageSize* returned from the *Server* in the *CreateSession* response is less than one megabyte. The *SignedSoftwareCertificate* type is defined in 7.31.
localeIds []	LocaleId	List of locale ids in priority order for localized strings. The first *LocaleId* in the list has the highest priority. If the *Server* returns a localized string to the *Client*, the *Server* shall return the translation with the highest priority that it can. If it does not have a translation for any of the locales identified in this list, then it shall return the string value that it has and include the locale id with the string. See IEC 62541-3 for more detail on locale ids. If the *Client* fails to specify at least one locale id, the *Server* shall use any that it has. This parameter only needs to be specified during the first call to *ActivateSession* during a single application *Session*. If it is not specified the *Server* shall keep using the current *localeIds* for the *Session*.
userIdentityToken	Extensible Parameter UserIdentityToken	The credentials of the user associated with the *Client* application. The *Server* uses these credentials to determine whether the *Client* should be allowed to activate a *Session* and what resources the *Client* has access to during this *Session*. The *UserIdentityToken* is an extensible parameter type defined in 7.35. The EndpointDescription specifies what *UserIdentityTokens* the Server shall accept. Null or empty user token shall always be interpreted as anonymous.
userTokenSignature	SignatureData	If the *Client* specified a user identity token that supports digital signatures, then it shall create a signature and pass it as this parameter. Otherwise the parameter is omitted. The *SignatureAlgorithm* depends on the identity token type. The *SignatureData* type is defined in 7.30.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The *Client* shall use this value to prove possession of its *Application Instance Certificate* in the next call to *ActivateSession* request.
results []	StatusCode	List of validation results for the *SoftwareCertificates* (see 7.33 for *StatusCode* definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information associated with *SoftwareCertificate* validation errors (see 7.8 for *DiagnosticInfo* definition). This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.6.3.3 Service results

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

Table 14 – ActivateSession Service Result Codes

Symbolic Id	Description
Bad_IdentityTokenInvalid	See Table 165 for the description of this result code.
Bad_IdentityTokenRejected	See Table 165 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_ApplicationSignatureInvalid	The signature provided by the client application is missing or invalid.
Bad_UserSignatureInvalid	The user token signature is missing or invalid.
Bad_NoValidCertificates	The *Client* did not provide at least one *Software Certificate* that is valid and meets the profile requirements for the *Server*.
Bad_IdentityChangeNotSupported	The *Server* does not support changing the user identity assigned to the session.

5.6.4 CloseSession

5.6.4.1 Description

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

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

b) It returns negative responses with the *StatusCode* Bad_SessionClosed to all requests that are currently outstanding to provide for the timely return of the *CloseSession* response. *Clients* are urged to wait for all outstanding requests to complete before submitting the *CloseSession* request.

c) It removes the entry for the *Client* in its *SessionDiagnosticsArray Variable*.

5.6.4.2 Parameters

Table 15 defines the parameters for the *Service*.

Table 15 – CloseSession Service Parameters

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

5.6.4.3 Service results

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

Table 16 – CloseSession Service Result Codes

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

5.6.5 Cancel

5.6.5.1 Description

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

5.6.5.2 Parameters

Table 17 defines the parameters for the *Service*.

Table 17 – Cancel Service Parameters

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

5.6.5.3 Service results

Common *StatusCodes* are defined in Table 165.

5.7 NodeManagement Service Set

5.7.1 Overview

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

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 IEC 62541-3).

5.7.2.2 Parameters

Table 18 defines the parameters for the *Service*.

Table 18 – AddNodes Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToAdd []	AddNodesItem	List of *Nodes* to add. All *Nodes* are added as a *Reference* to an existing *Node* using a hierarchical *ReferenceType*. This structure is defined in-line with the following indented items.
parentNodeId	Expanded NodeId	*ExpandedNodeId* of the parent *Node* for the *Reference*. The *ExpandedNodeId* type is defined in 7.10.
referenceTypeId	NodeId	*NodeId* of the hierarchical *ReferenceType* to use for the *Reference* from the parent *Node* to the new *Node*.
requestedNewNodeId	Expanded NodeId	*Client* requested expanded *NodeId* of the *Node* to add. The *serverIndex* in the expanded NodeId shall be 0. If the *Server* cannot use this *NodeId*, it rejects this *Node* and returns the appropriate error code. If the *Client* does not want to request a *NodeId*, then it sets the value of this parameter to the null expanded *NodeId*. If the *Node* to add is a *ReferenceType Node*, its *NodeId* should be a numeric id. See IEC 62541-3 for a description of *ReferenceType NodeIds*.
browseName	QualifiedName	The browse name of the *Node* to add.
nodeClass	NodeClass	*NodeClass* of the *Node* to add.
nodeAttributes	Extensible Parameter NodeAttributes	The *Attributes* that are specific to the *NodeClass*. The *NodeAttributes* parameter type is an extensible parameter type specified in 7.18. A *Client* is allowed to omit values for some or all *Attributes*. If an *Attribute* value is omitted, the *Server* shall use the default values from the *TypeDefinitionNode*. If a *TypeDefinitionNode* was not provided the *Server* shall choose a suitable default value. The *Server* may still add an optional Attribute to the *Node* with an appropriate default value even if the *Client* does not specify a value.
typeDefinition	Expanded NodeId	*NodeId* of the *TypeDefinitionNode* for the *Node* to add. This parameter shall be null for all *NodeClasses* other than *Object* and *Variable* in which case it shall be provided.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	AddNodesResult	List of results for the *Nodes* to add. The size and order of the list matches the size and order of the *nodesToAdd* request parameter. This structure is defined in-line with the following indented items.
statusCode	StatusCode	*StatusCode* for the *Node* to add (see 7.33 for *StatusCode* definition).
addedNodeId	NodeId	*Server* assigned *NodeId* of the added *Node*. Null *NodeId* if the operation failed.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Nodes* to add (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *nodesToAdd* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.2.3 Service results

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

Table 19 – AddNodes Service Result Codes

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

5.7.2.4 StatusCodes

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

Table 20 – AddNodes Operation Level Result Codes

Symbolic Id	Description
Bad_ParentNodeIdInvalid	The parent node id does not to refer to a valid node.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_ReferenceNotAllowed	The reference could not be created because it violates constraints imposed by the data model.
Bad_NodeIdRejected	The requested node id was rejected either because it was invalid or because the server does not allow node ids to be specified by the client.
Bad_NodeIdExists	The requested node id is already used by another node.
Bad_NodeClassInvalid	See Table 166 for the description of this result code.
Bad_BrowseNameInvalid	See Table 166 for the description of this result code.
Bad_BrowseNameDuplicated	The browse name is not unique among nodes that share the same relationship with the parent.
Bad_NodeAttributesInvalid	The node *Attributes* are not valid for the node class.
Bad_TypeDefinitionInvalid	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.

5.7.3 AddReferences

5.7.3.1 Description

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

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

5.7.3.2 Parameters

Table 21 defines the parameters for the *Service*.

Table 21 – AddReferences Service Parameters

Name	Type	Description
Request		
requestHeader	Request Header	Common request parameters (see 7.26 for *RequestHeader* definition).
referencesToAdd []	AddReferences Item	List of *Reference* instances to add to the *SourceNode*. The *targetNodeClass* of each *Reference* in the list shall match the *NodeClass* of the *TargetNode*. This structure is defined in-line with the following indented items.
sourceNodeId	NodeId	*NodeId* of the *Node* to which the *Reference* is to be added. The source *Node* shall always exist in the *Server* to add the *Reference*. The isForward parameter can be set to FALSE if the target *Node* is on the local *Server* and the source *Node* on the remote *Server*.
referenceTypeId	NodeId	*NodeId* of the *ReferenceType* that defines the *Reference*.
isForward	Boolean	If the value is TRUE, the Server creates a forward Reference. If the value is FALSE, the Server creates an inverse Reference.
targetServerUri	String	URI of the remote *Server*. If this parameter is not null, it overrides the *serverIndex* in the *targetNodeId*.
targetNodeId	Expanded NodeId	Expanded *NodeId* of the *TargetNode*. The *ExpandedNodeId* type is defined in 7.10.
targetNodeClass	NodeClass	*NodeClass* of the *TargetNode*. The *Client* shall specify this since the *TargetNode* might not be accessible directly by the *Server*.
Response		
responseHeader	Response Header	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of *StatusCodes* for the *References* to add (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *referencesToAdd* request parameter.
diagnosticInfos []	Diagnostic Info	List of diagnostic information for the *References* to add (see 7.8 for

Name	Type	Description
Request		
		DiagnosticInfo definition). The size and order of the list matches the size and order of the *referencesToAdd* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.3.3 Service results

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

Table 22 – AddReferences Service Result Codes

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

5.7.3.4 StatusCodes

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

Table 23 – AddReferences Operation Level Result Codes

Symbolic Id	Description
Bad_SourceNodeIdInvalid	See Table 166 for the description of this result code.
Bad_ReferenceTypeIdInvalid	See Table 166 for the description of this result code.
Bad_ServerUriInvalid	See Table 166 for the description of this result code.
Bad_TargetNodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeClassInvalid	See Table 166 for the description of this result code.
Bad_ReferenceNotAllowed	The reference could not be created because it violates constraints imposed by the data model on this server.
Bad_ReferenceLocalOnly	The reference type is not valid for a reference to a remote *Server*.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_DuplicateReferenceNotAllowed	The reference type between the nodes is already defined.
Bad_InvalidSelfReference	The server does not allow this type of self reference on this node.

5.7.4 DeleteNodes

5.7.4.1 Description

This *Service* is used to delete one or more *Nodes* from the *AddressSpace*.

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

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

5.7.4.2 Parameters

Table 24 defines the parameters for the *Service*.

Table 24 – DeleteNodes Service Parameters

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

5.7.4.3 Service results

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

Table 25 – DeleteNodes Service Result Codes

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

5.7.4.4 StatusCodes

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

Table 26 – DeleteNodes Operation Level Result Codes

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

5.7.5 DeleteReferences

5.7.5.1 Description

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

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

Table 27 defines the parameters for the *Service*.

Table 27 – DeleteReferences Service Parameters

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

5.7.5.2 Service results

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

Table 28 – DeleteReferences Service Result Codes

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

5.7.5.3 StatusCodes

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

Table 29 – DeleteReferences Operation Level Result Codes

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

5.8 View Service Set

5.8.1 Overview

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

A *View* is a subset of the *AddressSpace* created by the *Server*. Future versions of this standard may also define services to create *Client*-defined *Views*. See IEC 62541-5 for a description of the organisation of views in the *AddressSpace*.

5.8.2 Browse

5.8.2.1 Description

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

5.8.2.2 Parameters

Table 30 defines the parameters for the *Service*.

Table 30 – Browse Service Parameters

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

5.8.2.3 Service results

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

Table 31 – Browse Service Result Codes

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

5.8.2.4 StatusCodes

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

Table 32 – Browse Operation Level Result Codes

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

5.8.3 BrowseNext

5.8.3.1 Description

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

5.8.3.2 Parameters

Table 33 defines the parameters for the *Service*.

Table 33 – BrowseNext Service Parameters

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

5.8.3.3 Service results

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

Table 34 – BrowseNext Service Result Codes

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

5.8.3.4 StatusCodes

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

Table 35 – BrowseNext Operation Level Result Codes

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

5.8.4 TranslateBrowsePathsToNodeIds

5.8.4.1 Description

This *Service* is used to request that the *Server* translates one or more browse paths to *NodeIds*. Each browse path is constructed of a starting *Node* and a *RelativePath*. The specified starting *Node* identifies the *Node* from which the *RelativePath* is based. The *RelativePath* contains a sequence of *ReferenceTypes* and *BrowseNames*.

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

If a *Node* has multiple targets with the same *BrowseName*, the *Server* shall return a list of *NodeIds*. However, since one of the main purposes of this *Service* is to support programming against type definitions, the *NodeId* of the *Node* based on the type definition of the starting *Node* is returned as the first *NodeId* in the list.

5.8.4.2 Parameters

Table 36 defines the parameters for the *Service*.

Table 36 – TranslateBrowsePathsToNodeIds Service Parameters

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

5.8.4.3 Service results

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

Table 37 – TranslateBrowsePathsToNodeIds Service Result Codes

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

5.8.4.4 StatusCodes

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

Table 38 – TranslateBrowsePathsToNodeIds Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_NothingToDo	See Table 165 for the description of this result code. This code indicates that the relativePath contained an empty list.
Bad_BrowseNameInvalid	See Table 166 for the description of this result code. This code indicates that a TargetName was missing in a RelativePath.
Uncertain_ReferenceOutOfServer	The path element has targets which are in another server.
Bad_TooManyMatches	The requested operation has too many matches to return. Users should use queries for large result sets. *Servers* should allow at least 10 matches before returning this error code.
Bad_QueryTooComplex	The requested operation requires too many resources in the server.
Bad_NoMatch	The requested relativePath cannot be resolved to a target to return.

5.8.5 RegisterNodes

5.8.5.1 Description

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

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

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

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

5.8.5.2 Parameters

Table 39 defines the parameters for the *Service*.

Table 39 – RegisterNodes Service Parameters

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

5.8.5.3 Service results

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

Table 40 – RegisterNodes Service Result Codes

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

5.8.6 UnregisterNodes

5.8.6.1 Description

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

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

5.8.6.2 Parameters

Table 46 defines the parameters for the *Service*.

Table 41 – UnregisterNodes Service Parameters

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

5.8.6.3 Service results

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

Table 42 – UnregisterNodes Service Result Codes

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

5.9 Query Service Set

5.9.1 Overview

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

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

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

See Annex B for example queries.

5.9.2 Querying Views

A *View* is a subset of the *AddressSpace* available in the *Server*. See IEC 62541-5 for a description of the organisation of *Views* in the *AddressSpace*.

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

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

Servers that support *Query* are expected 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 the remote *Server*, but it is not required. If a *Server* does access a remote *Server* the access shall be accomplished using the user identity of the *Client* as described in 5.5.1.

5.9.3 QueryFirst

5.9.3.1 Description

This *Service* is used to issue a *Query* request to the *Server*. The complexity of the *Query* can range from very simple to highly sophisticated. The *Query* can simply request data from instances of a *TypeDefinitionNode* or *TypeDefinitionNode* subject to restrictions specified by the filter. On the other hand, the *Query* can request data from instances of related *Node* types by specifying *a RelativePath* from an originating *TypeDefinitionNode*. In the filter, a separate set of paths can be constructed for limiting the instances that supply data. A filtering path can include multiple *RelatedTo* operators to define a multi-hop path between source instances and target instances. For example, one could filter on students that attend a particular school, but return information about students and their families. In this case, the student school relationship is traversed for filtering, but the student family relationship is traversed to select data. For a complete description of *ContentFilter* see 7.4, also see Clause B.1 for simple examples and Clause B.2 for more complex examples of content filter and queries.

The *Client* provides an array of *NodeTypeDescription* which specify the *NodeId* of a *TypeDefinitionNode* and selects what *Attributes* are to be returned in the response. A client

can also provide a set of *RelativePaths* through the type system starting from an originating *TypeDefinitionNode*. Using these paths, the client selects a set of *Attributes* from *Nodes* that are related to instances of the originating *TypeDefinitionNode*. Additionally, the *Client* can request the *Server* return instances of subtypes of *TypeDefinitionNodes*. If a selected *Attribute* does not exist in a *TypeDefinitionNode* but does exist in a subtype, it is assumed to have a null value in the *TypeDefinitionNode* in question. Therefore, this does not constitute an error condition and a null value is returned for the *Attribute*.

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

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

Query Services allow a special handling of the targetName field in the RelativePath. In several Query use cases a type NodeId is necessary in the path instead of a QualifiedName. Therefore the Client is allowed to specify a NodeId in the QualifiedName. This is done by setting the namespaceIndex of the targetName to zero and the name part of the targetName to the XML representation of the NodeId. The XML representation is defined in IEC 62541-6. When matching instances are returned as the target node, the target node shall be an instance of the specified type or subtype of the specified type.

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

Table 43 – QueryFirst Request Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
view	ViewDescription	Specifies a *View* and temporal context to a *Server* (see 7.37 for *ViewDescription* definition).
nodeTypes []	NodeTypeDescription	This is the *Node* type description. This structure is defined in-line with the following indented items.
typeDefinitionNode	ExpandedNodeId	*NodeId* of the originating *TypeDefinitionNode* of the instances for which data is to be returned.
includeSubtypes	Boolean	A flag that indicates whether the *Server* should include instances of subtypes of the TypeDefinitionNode in the list of instances of the *Node* type.
dataToReturn []	QueryDataDescription	Specifies an *Attribute* or *Reference* from the originating typeDefinitionNode along a given relativePath for which to return data. This structure is defined in-line with the following indented items.
relativePath	RelativePath	Browse path relative to the originating Node that identifies the Node which contains the data that is being requested, where the originating *Node* is an instance *Node* of the type defined by the type definition *Node*. The instance *Nodes* are further limited by the filter provided as part of this call. For a definition of relativePath see 7.25. This relative path could end on a *Reference*, in which case the *ReferenceDescription* of the *Reference* would be returned as its value. The targetName field of the relativePath may contain a type NodeId. This is done by setting the *namespaceIndex* of the *targetName* to zero and the *name* part of the *targetName* to the XML representation of the *NodeId*. The XML representation is defined in IEC 62541-6. When matching instances are returned as the target node, the target node shall be an instance of the specified type or subtype of the specified type.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* Id. The *IntegerId* is defined in 7.13. The IntegerId for Attributes are defined in IEC 62541-6. If the *RelativePath* ended in a *Reference* then this parameter is 0 and ignored by the server.
indexRange	NumericRange	This parameter is used to identify a single element of a structure or an array, or a single range of indexes for arrays. If a range of elements are specified, the values are returned as a composite. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is null if the specified *Attribute* is not an array or a structure. However, if the specified *Attribute* is an array or a structure, and this parameter is null, then all elements are to be included in the range.
filter	ContentFilter	Resulting *Nodes* shall be limited to the *Nodes* matching the criteria defined by the filter. ContentFilter is discussed in 7.4. If an empty filter is provided then the entire address space shall be examined and all *Nodes* that contain a matching requested *Attribute* or *Reference* are returned.
maxDataSetsToReturn	Counter	The number of *QueryDataSets* that the *Client* wants the *Server* to return in the response and on each subsequent continuation call response. The Server is allowed to further limit the response, but shall not exceed this limit. A value of 0 indicates that the *Client* is imposing no limitation.
maxReferencesToReturn	Counter	The number of *References* that the *Client* wants the *Server* to return in the response for each *QueryDataSet* and on each subsequent continuation call response. The Server is allowed to further limit the response, but shall not exceed this limit. A value of 0 indicates that the *Client* is imposing no limitation. For example a result where 4 *Nodes* are being returned, but each has 100 *References*, if this limit were set to 50 then only the first 50 *References* for each *Node* would be returned on the initial call and a continuation point would be set indicating additional data.

Table 44 – QueryFirst Response Parameters

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

5.9.3.2 Service results

If the *Query* is invalid or cannot be processed, then *QueryDataSets* are not returned and only a *Service* result, filterResult, parsingResults and optional *DiagnosticInfo* is returned. Table 45 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 165.

Table 45 – QueryFirst Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_ContentFilterInvalid	See Table 166 for the description of this result code.
Bad_ViewIdUnknown	See Table 165 for the description of this result code.
Bad_ViewTimestampInvalid	See Table 165 for the description of this result code.
Bad_ViewParameterMismatchInvalid	See Table 165 for the description of this result code.
Bad_ViewVersionInvalid	See Table 165 for the description of this result code.
Bad_InvalidFilter	The provided filter is invalid, see the filterResult for specific errors
Bad_NodelistError	The NodeTypes provided contain an error, see the parsingResults for specific errors
Bad_InvalidView	The provided ViewDescription is not a valid ViewDescription.
Good_ResultsMayBeIncomplete	The server should have followed a reference to a node in a remote server but did not. The result set may be incomplete.

5.9.3.3 StatusCodes

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

Table 46 – QueryFirst Operation Level Result Codes

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

5.9.4 QueryNext

5.9.4.1 Descriptions

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

5.9.4.2 Parameters

Table 47 defines the parameters for the *Service*.

Table 47 – QueryNext Service Parameters

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

5.9.4.3 Service results

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

Table 48 – QueryNext Service Result Codes

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

5.10 Attribute Service Set

5.10.1 Overview

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

5.10.2 Read

5.10.2.1 Description

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

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

5.10.2.2 Parameters

Table 49 defines the parameters for the *Service*.

Table 49 – Read Service Parameters

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

5.10.2.3 Service results

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

Table 50 – Read Service Result Codes

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

5.10.2.4 StatusCodes

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

Table 51 – Read Operation Level Result Codes

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

5.10.3 HistoryRead

5.10.3.1 Description

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

The *AccessLevel Attribute* defined in IEC 62541-3 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 IEC 62541-3 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 IEC 62541-11.

5.10.3.2 Parameters

Table 52 defines the parameters for the *Service*.

Table 52 – HistoryRead Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
historyReadDetails	Extensible Parameter HistoryReadDetails	The details define the types of history reads that can be performed. The *HistoryReadDetails* parameter type is an extensible parameter type formally defined in IEC 62541-11. The *ExtensibleParameter* type is defined in 7.11.
timestampsToReturn	Enum TimestampsTo Return	An enumeration that specifies the timestamps to be returned for each requested *Variable Value Attribute*. The *TimestampsToReturn* enumeration is defined in 7.34. Specifying a *TimestampsToReturn* of NEITHER is not valid. A *Server* shall return a *Bad_InvalidTimestampArgument StatusCode* in this case. IEC 62541-11 defines exceptions where this parameter shall be ignored.
releaseContinuation Points	Boolean	A *Boolean* parameter with the following values: TRUE passed *continuationPoints* shall be reset to free resources in the *Server*. FALSE passed *continuationPoints* shall be used to get the next set of historical information. A *Client* shall always use the continuation point returned by a *HistoryRead* response to free the resources for the continuation point in the *Server*. If the *Client* does not want to get the next set of historical information, *HistoryRead* shall be called with this parameter set to TRUE.
nodesToRead []	HistoryReadValueId	This parameter contains the list of items upon which the historical retrieval is to be performed. This structure is defined in-line with the following indented items.
nodeId	NodeId	If the *HistoryReadDetails* is RAW, PROCESSED, MODIFIED or ATTIME: The *nodeId* of the *Nodes* whose historical values are to be read. The value returned shall always include a timestamp. If the *HistoryReadDetails* is EVENTS: The *NodeId* of the *Node* whose *Event* history is to be read. If the *Node* does not support the requested access for historical values or historical *Events* the appropriate error response for the given *Node* shall be generated.
indexRange	NumericRange	This parameter is used to identify a single element of an array, or a single range of indexes for arrays. If a range of elements is specified, the values are returned as a composite. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is null if the value is not an array. However, if the value is an array, and this parameter is null, then all elements are to be included in the range.
dataEncoding	QualifiedName	A *QualifiedName* that specifies the data encoding to be returned for the *Value* to be read (see 7.23 for definition how to specify the data encoding). The parameter is ignored when reading history of *Events*.
continuationPoint	ByteString	For each *NodesToRead* item this parameter specifies a continuation point returned from a previous *HistoryRead* call, allowing the *Client* to continue that read from the last value received. The *HistoryRead* is used to select an ordered sequence of historical values or events. A continuation point marks a point in that ordered sequence, such that the *Server* returns the subset of the sequence that follows that point. A null value indicates that this parameter is not used. This continuation point is described in more detail in IEC 62541-11.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* type).
results []	HistoryReadResult	List of read results. The size and order of the list matches the size and order of the *nodesToRead* request parameter. This structure is defined in-line with the following indented items.
statusCode	StatusCode	*StatusCode* for the *NodesToRead* item (see 7.33 for *StatusCode* definition).
continuationPoint	ByteString	This parameter is used only if the number of values to be returned is too large to be returned in a single response. When this parameter is not used, its value is null. *Servers* shall support at least one continuation point per *Session*. *Servers* specify a max history continuation points per *Session* in the *Server* capabilities *Object* defined in IEC 62541-5. A continuation point shall remain active until the *Client* passes the continuation point to *HistoryRead* or the *Session* is closed. If the max continuation points have been reached the oldest continuation point shall be reset.
historyData	Extensible Parameter HistoryData	The history data returned for the *Node*. The *HistoryData* parameter type is an extensible parameter type formally defined in IEC 62541-11. It specifies the types of history data that can be returned. The *ExtensibleParameter* base type is defined in 7.11.
diagnosticInfos []	Diagnostic Info	List of diagnostic information. The size and order of the list matches the size and order of the *nodesToRead* request parameter. There is one entry in this list

Name	Type	Description
		for each *Node* contained in the *nodesToRead* parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.3.3 Service results

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

Table 53 – HistoryRead Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_TimestampsToReturnInvalid	See Table 165 for the description of this result code.
Bad_HistoryOperationInvalid	See Table 166 for the description of this result code.
Bad_HistoryOperationUnsupported	See Table 166 for the description of this result code. The requested history operation is not supported by the server.

5.10.3.4 StatusCodes

Table 54 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 166. History access specific *StatusCodes* are defined in IEC 62541-11.

Table 54 – HistoryRead Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_DataEncodingInvalid	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_ContinuationPointInvalid	See Table 165 for the description of this result code.
Bad_InvalidTimestampArgument	The defined timestamp to return was invalid.
Bad_HistoryOperationUnsupported	See Table 166 for the description of this result code. The requested history operation is not supported for the requested node.

5.10.4 Write

5.10.4.1 Description

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

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

The order the operations are processed in the *Server* is not defined and depends on the different data sources and the internal *Server* logic. If an *Attribute* and *Node* combination is contained in more than one operation, the order of the processing is undefined. If a *Client* requires sequential processing the *Client* needs separate *Service* calls.

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

If a *Server* allows writing of *Attributes* with the *DataType* LocalizedText, the Client can add or overwrite the text for a locale by writing the text with the associated *LocaleId*. Writing a null *String* for the text for a locale shall delete the *String* for that locale. Writing a null *String* for the *locale* and a non-null *String* for the *text* is setting the *text* for an invariant locale. Writing a null *String* for the *text* and a null *String* for the *locale* shall delete the entries for all locales. If a *Client* attempts to write a *locale* that is either syntactically invalid or not supported, the *Server* returns Bad_LocaleNotSupported.

5.10.4.2 Parameters

Table 55 defines the parameters for the *Service*.

Table 55 – Write Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
nodesToWrite []	WriteValue	List of *Nodes* and their *Attributes* to write. This structure is defined in-line with the following indented items.
nodeId	NodeId	*NodeId* of the *Node* that contains the *Attributes*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* Id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in IEC 62541-6.
indexRange	NumericRange	This parameter is used to identify a single element of an array, or a single range of indexes for arrays. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is not used if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used. A *Server* shall return a Bad_WriteNotSupported error if an *indexRange* is provided and writing of *indexRange* is not possible for the *Node*.
value	DataValue	The *Node*'s *Attribute* value (see 7.7 for *DataValue* definition). If the *indexRange* parameter is specified then the *Value* shall be an array even if only one element is being written. If the *SourceTimestamp* or the *ServerTimestamp* is specified, the *Server* shall use these values. The *Server* returns a Bad_WriteNotSupported error if it does not support writing of timestamps. A *Server* shall return a Bad_TypeMismatch error if the data type of the written value is not the same type or subtype of the *Attribute's DataType*. Based on the *DataType* hierarchy, subtypes of the *Attribute DataType* shall be accepted by the *Server*. For the *Value Attribute* the *DataType* is defined through the *DataType Attribute*. A *ByteString* is structurally the same as a one dimensional array of Byte. A *Server* shall accept a *ByteString* if an array of *Byte* is expected. The *Server* returns a Bad_DataEncodingUnsupported error if it does not support the provided data encoding. *Simple DataTypes* (see IEC 62541-3) use the same representation on the wire as their supertypes and therefore writing a value of a simple *DataType* cannot be distinguished from writing a value of its supertype. The *Server* shall assume that by receiving the correct wire representation for a simple *DataType* the correct type was chosen. *Servers* are allowed to impose additional data validations on the value independent of the encoding (e.g. having an image in GIF format in a *ByteString*). In this case the *Server* shall return a Bad_TypeMismatch error if the validation fails.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	StatusCode	List of results for the *Nodes* to write (see 7.33 for *StatusCode* definition). The size and order of the list matches the size and order of the *nodesToWrite* request parameter. There is one entry in this list for each *Node* contained in the *nodesToWrite* parameter.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *Nodes* to write (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of the *nodesToWrite* request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.4.3 Service results

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

Table 56 – Write Service Result Codes

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

5.10.4.4 StatusCodes

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

Table 57 – Write Operation Level Result Codes

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

5.10.5 HistoryUpdate

5.10.5.1 Description

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

5.10.5.2 Parameters

Table 58 defines the parameters for the *Service*.

Table 58 – HistoryUpdate Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
historyUpdateDetails []	Extensible Parameter HistoryUpdate Details	The details defined for this update. The *HistoryUpdateDetails* parameter type is an extensible parameter type formally defined in IEC 62541-11. It specifies the types of history updates that can be performed. The *ExtensibleParameter* type is defined in 7.11.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	HistoryUpdate Result	List of update results for the history update details. The size and order of the list matches the size and order of the details element of the *historyUpdateDetails* parameter specified in the request. This structure is defined in-line with the following indented items.
statusCode	StatusCode	*StatusCode* for the update of the *Node* (see 7.33 for *StatusCode* definition).
operationResults []	StatusCode	List of *StatusCodes* for the operations to be performed on a *Node*. The size and order of the list matches the size and order of any list defined by the details element being reported by this result entry.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the operations to be performed on a *Node* (see 7.8 for *DiagnosticInfo* definition). The size and order of the list matches the size and order of any list defined by the details element being reported by this *updateResults* entry. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the history update details. The size and order of the list matches the size and order of the details element of the *historyUpdateDetails* parameter specified in the request. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.10.5.3 Service results

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

Table 59 – HistoryUpdate Service Result Codes

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

5.10.5.4 StatusCodes

Table 60 defines values for the *statusCode* and *operationResults* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166. History access specific *StatusCodes* are defined in IEC 62541-11.

Table 60 – HistoryUpdate Operation Level Result Codes

Symbolic Id	Description
Bad_NotWritable	See Table 166 for the description of this result code.
Bad_HistoryOperationInvalid	See Table 166 for the description of this result code.
Bad_HistoryOperationUnsupported	See Table 166 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code. The current user does not have permission to update the history.

5.11 Method Service Set

5.11.1 Overview

Methods represent the function calls of *Objects*. They are defined in IEC 62541-3. *Methods* are invoked and return only after completion (successful or unsuccessful). Execution times for methods may vary, depending on the function that they perform.

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

Because *Methods* may control some aspect of plant operations, method invocation may depend on environmental or other conditions. This may be especially true when attempting to re-invoke a method immediately after it has completed execution. Conditions that are required to invoke the method might not yet have returned to the state that permits the method to start again.

5.11.2 Call

5.11.2.1 Description

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

The order the operations are processed in the *Server* is not defined and depends on the different tasks and the internal *Server* logic. If a *Method* is contained in more than one operation, the order of the processing is undefined. If a *Client* requires sequential processing the *Client* needs separate *Service* calls.

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

5.11.2.2 Parameters

Table 61 defines the parameters for the *Service*.

Table 61 – Call Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
methodsToCall []	CallMethodRequest	List of *Methods* to call. This structure is defined in-line with the following indented items.
objectId	NodeId	The *NodeId* shall be that of the *Object* or *ObjectType* that is the source of a *HasComponent Reference* (or subtype of *HasComponent Reference*) to the *Method* specified in *methodId*. See IEC 62541-3 for a description of *Objects* and their *Methods*.
methodId	NodeId	*NodeId* of the *Method* to invoke. If the *objectId* is the *NodeId* of an *Object*, it is allowed to use the *NodeId* of a *Method* that is the target of a *HasComponent Reference* from the *ObjectType* of the *Object*.
inputArguments []	BaseDataType	List of input argument values. An empty list indicates that there are no input arguments. The size and order of this list matches the size and order of the input arguments defined by the input *InputArguments Property* of the *Method*. The name, a description and the data type of each argument are defined by the *Argument* structure in each element of the method's *InputArguments Property*.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
results []	CallMethodResult	Result for the *Method* calls. This structure is defined in-line with the following indented items.
statusCode	StatusCode	*StatusCode* of the *Method* executed in the server. This *StatusCode* is set to the Bad_InvalidArgument *StatusCode* if at least one input argument broke a constraint (e.g. wrong data type, value out of range). This *StatusCode* is set to a bad *StatusCode* if the *Method* execution failed in the server, e.g. based on an exception.
inputArgumentResults []	StatusCode	List of *StatusCodes* corresponding to the *inputArguments*. This list is empty unless the operation level result is Bad_InvalidArgument. If this list is populated, it has the same length as the *inputArgument* list.
inputArgumentDiagnosticInfos []	DiagnosticInfo	List of diagnostic information corresponding to the *inputArguments*. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.
outputArguments []	BaseDataType	List of output argument values. An empty list indicates that there are no output arguments. The size and order of this list matches the size and order of the output arguments defined by the *OutputArguments Property* of the *Method*. The name, a description and the data type of each argument are defined by the *Argument* structure in each element of the methods *OutputArguments Property*.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the *statusCode* of the *results*. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.11.2.3 Service results

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

Table 62 – Call Service Result Codes

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

5.11.2.4 StatusCodes

Table 63 defines values for the *statusCode* and *inputArgumentResults* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 166.

Table 63 – Call Operation Level Result Codes

Symbolic Id	Description
Bad_NodeIdInvalid	See Table 166 for the description of this result code. Used to indicate that the specified object is not valid.
Bad_NodeIdUnknown	See Table 166 for the description of this result code. Used to indicate that the specified object is not valid.
Bad_ArgumentsMissing	The client did not specify all of the input arguments for the method.
Bad_TooManyArguments	The client specified more input arguments than defined for the method.
Bad_InvalidArgument	See Table 165 for the description of this result code. Used to indicate in the operation level results that one ore more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.
Bad_UserAccessDenied	See Table 165 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object.
Bad_OutOfRange	See Table 166 for the description of this result code. Used to indicate that an input argument is outside the acceptable range.
Bad_TypeMismatch	See Table 166 for the description of this result code. Used to indicate that an input argument does not have the correct data type. A ByteString is structurally the same as a one dimensional array of Byte. A server shall accept a ByteString if an array of Byte is expected.
Bad_NoCommunication	See Table 166 for the description of this result code.

5.12 MonitoredItem Service Set

5.12.1 MonitoredItem model

5.12.1.1 Overview

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

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

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

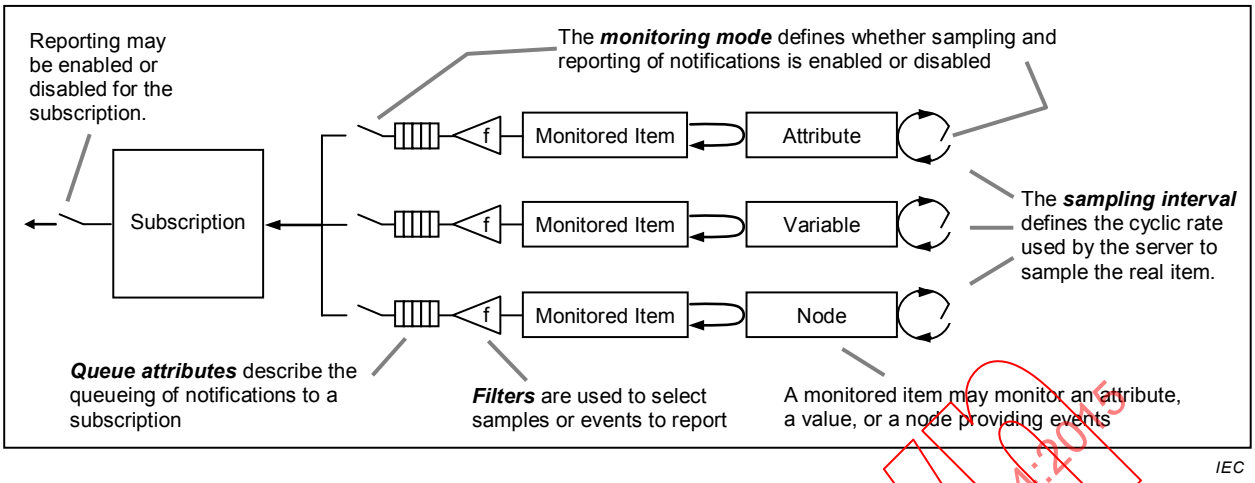


Figure 15 – MonitoredItem Model

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

The *Value Attribute* is used when monitoring *Variables*. *Variable* values are monitored for a change in value or a change in their status. The filters defined in this standard (see 7.16.2) and in IEC 62541-8 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 standard (see 7.16.3) is used to determine if an *Event* received from the *Node* is sent to the *Client*. The filter also allows selecting fields of the *EventType* that will be contained in the *Event* such as *EventId*, *EventType*, *SourceNode*, *Time* and *Description*.

IEC 62541-3 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 IEC 62541-5.

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. A negative number indicates that the default sampling interval defined by the publishing interval of the *Subscription* is requested. The sampling interval indicates the fastest rate at which the *Server* should sample its underlying source for data changes.

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

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

The *Client* may also specify 0 for the sampling interval, which indicates that the *Server* should use the fastest practical rate. It is expected that *Servers* will support only a limited set of

sampling intervals to optimize their operation. If the exact interval requested by the *Client* is not supported by the *Server*, then the *Server* assigns to the *MonitoredItem* the most appropriate interval as determined by the *Server*. It returns this assigned interval to the *Client*. The *Server* Capabilities *Object* defined in IEC 62541-5 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. An exception-based model means that the underlying system does not require sampling and reports data changes.

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

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

If the behaviour by which the underlying system updates the item is known, it will be available via the *MinimumSamplingInterval Attribute* defined in IEC 62541-3. If the *Server* specifies a value for the *MinimumSamplingInterval Attribute* it shall always return a *revisedSamplingInterval* that is equal or higher than the *MinimumSamplingInterval* if the *Client* subscribes to the *Value Attribute*.

Clients should also be aware that the sampling by the OPC UA *Server* and the update cycle of the underlying system are usually not synchronized. This can cause additional delays in change detection, as illustrated in Figure 16.

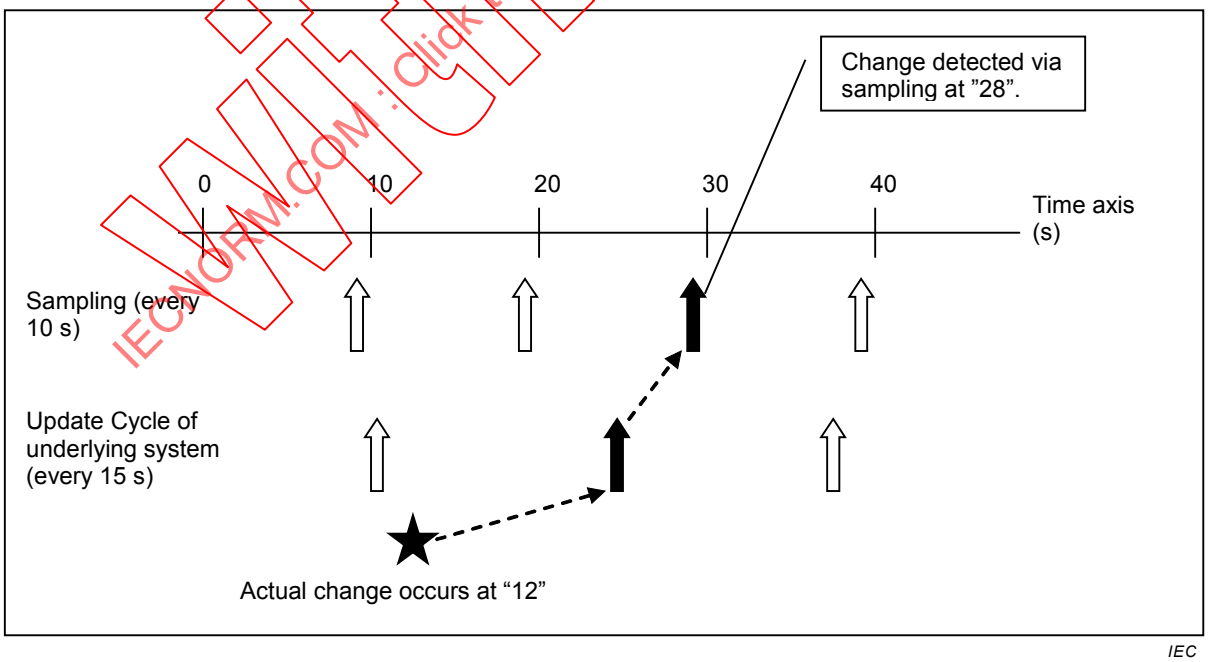


Figure 16 – Typical delay in change detection

5.12.1.3 Monitoring mode

The monitoring mode parameter is used to enable and disable the sampling of a *MonitoredItem*, and also to provide for independently enabling and disabling the reporting of *Notifications*. This capability allows a *MonitoredItem* to be configured to sample, sample and report, or neither. Disabling sampling does not change the values of any of the other *MonitoredItem* parameter, such as its sampling interval.

When a *MonitoredItem* is enabled (i.e. when the *MonitoringMode* is changed from *DISABLED* to *SAMPLING* or *REPORTING*) or it is created in the enabled state, the *Server* shall report the first sample as soon as possible and the time of this sample becomes the starting point for the next sampling interval.

5.12.1.4 Filter

Each time a *MonitoredItem* is sampled, the *Server* evaluates the sample using the filter defined for the *MonitoredItem*. The filter parameter defines the criteria that the *Server* uses to determine if a *Notification* should be generated for the sample. The type of filter is dependent on the type of the item that is being monitored. For example, the *DataChangeFilter* and the *AggregateFilter* are used when monitoring *Variable Values* and the *EventFilter* is used when monitoring *Events*. Sampling and evaluation, including the use of filters, are described in this standard. Additional filters may be defined in other parts of this series of standards.

5.12.1.5 Queue parameters

If the sample passes the filter criteria, a *Notification* is generated and queued for transfer by the *Subscription*. The size of the queue is defined when the *MonitoredItem* is created. When the queue is full and a new *Notification* is received, the *Server* either discards the oldest *Notification* and queues the new one, or it replaces the last value added to the queue with 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* and the size of the queue is larger than one, then the *Overflow* bit (flag) in the *InfoBits* portion of the *DataValue statusCode* is set. If *discardOldest* is TRUE, the oldest value gets deleted from the queue and the next value in the queue gets the flag set. If *discardOldest* is FALSE, the last value added to the queue gets replaced with the new value. The new value gets the flag set to indicate the lost values in the next *NotificationMessage*. Figure 17 illustrates the queue overflow handling.

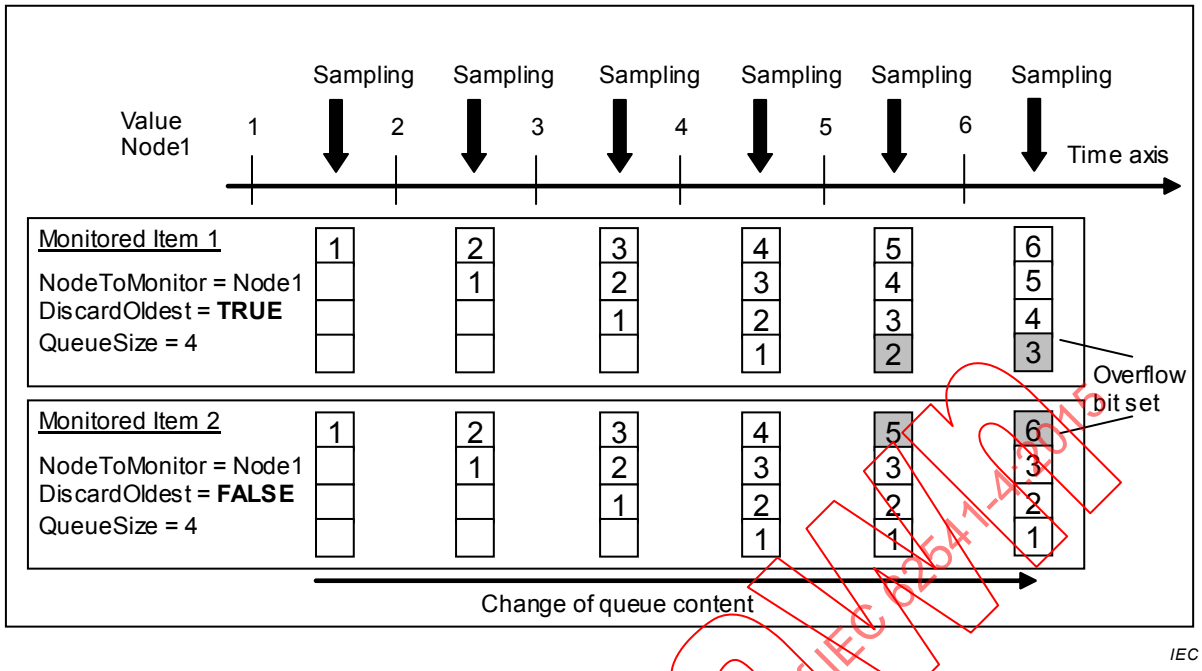


Figure 17 – Queue overflow handling

If the queue size is one, 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. The discard policy is ignored if the queue size is one.

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

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

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

- 100
- 111
- 100
- 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. The new first value in the queue may not exceed the *Deadband* limit of the previous value sent to the *Client*.

The queue size is the maximum value supported by the *Server* when monitoring *Events*. In this case, the *Server* is responsible for the *Event* buffer. If *Events* are lost, an *Event* of the type *EventQueueOverflowEventType* is placed in the queue. This Event is generated when the first Event has to be discarded on a MonitoredItem subscribing for Events. It is put into the Queue of the MonitoredItem in addition to the size of the Queue defined for this MonitoredItem without discarding any other Event. If *discardOldest* is set to TRUE it is put at the beginning of the queue and is never discarded, otherwise at the end. An aggregating *Server* shall not pass on such an *Event*. It shall be handled like other connection error scenarios.

5.12.1.6 Triggering model

The *MonitoredItems Service* allows the addition of 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 18 illustrates this concept.

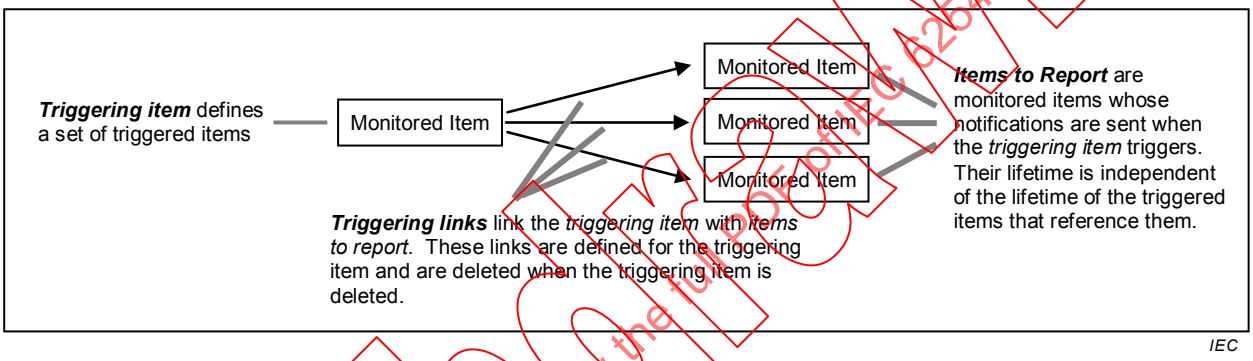


Figure 18 – Triggering Model

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) If the monitoring mode of the triggering item is *SAMPLING, then* it is not reported when the triggering item triggers the items to report.

b) If the monitoring mode of the triggering item is *REPORTING, then it* is reported when the triggering item triggers the items to report.

c) If the monitoring mode of the triggering item is *DISABLED*, then the triggering item does not trigger the items to report.

d) If the monitoring mode of the item to report is *SAMPLING, then it* is reported when the triggering item triggers the items to report.

e) If the monitoring mode of the item to report is *REPORTING*, this effectively causes the triggering item to be ignored. All notifications of the items to report are sent after the publishing interval expires.

f) If the monitoring mode of the item to report is *DISABLED*, then there will be no sampling of the item to report and therefore no notifications to report.

g) The first trigger shall occur when the first notification is queued for the triggering item after the creation of the link.

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_NodeIdUnknown* that indicates the deletion, but the *MonitoredItem* is not deleted.

5.12.2 CreateMonitoredItems

5.12.2.1 Description

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

Calling the *CreateMonitoredItems Service* repetitively to add a small number of *MonitoredItems* each time may adversely affect the performance of the *Server*. Instead, *Clients* should add a complete set of *MonitoredItems* to a *Subscription* whenever possible.

When a *MonitoredItem* is added, the *Server* performs initialization processing for it. The initialization processing is defined by the *Notification* type of the item being monitored. *Notification* types are specified in this standard and in the Access Type Specification parts of this series of standards, such as IEC 62541-8. See IEC TR 62541-1 for a description of the Access Type Parts.

When a user adds a monitored item that the user is denied read access to, the add operation for the item shall succeed and the bad status *Bad_NotReadable* or *Bad_UserAccessDenied* shall be returned in the Publish response. This is the same behaviour for the case where the access rights are changed after the call to *CreateMonitoredItems*. If the access rights change to read rights, the *Server* shall start sending data for the *MonitoredItem*. The same procedure shall be applied for an *IndexRange* that does not deliver data for the current value but could deliver data in the future.

Monitored *Nodes* can be removed from the *AddressSpace* after the creation of a *MonitoredItem*. This does not affect the validity of the *MonitoredItem* but a *Bad_NodeIdUnknown* shall be returned in the Publish response.

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

Illegal request values for parameters that can be revised do not generate errors. Instead the server will choose default values and indicate them in the corresponding revised parameter.

It is strongly recommended by OPC UA that a *Client* reuses a *Subscription* after a short network interruption by activating the existing *Session* on a new *SecureChannel* as described in 6.5. If a *Client* called *CreateMonitoredItems* during the network interruption and the call succeeded in the *Server* but did not return to the *Client*, then the *Client* does not know if the call succeeded. The *Client* may receive data changes for these monitored items but is not able to remove them since it does not know the *Server* handle for each monitored item. There is also no way for the *Client* to detect if the create succeeded. To delete and recreate the *Subscription* is also not an option since there may be several monitored items operating normally that should not be interrupted. To resolve this situation, the *Server Object* provides a *Method GetMonitoredItems* that returns the list of server and client handles for the monitored items in a *Subscription*. This *Method* is defined in IEC 62541-5.

5.12.2.2 Parameters

Table 64 defines the parameters for the *Service*.

Table 64 – CreateMonitoredItems Service Parameters

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

5.12.2.3 Service results

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

Table 65 – CreateMonitoredItems Service Result Codes

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

5.12.2.4 StatusCodes

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

Table 66 – CreateMonitoredItems Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoringModeInvalid	See Table 166 for the description of this result code.
Bad_NodeIdInvalid	See Table 166 for the description of this result code.
Bad_NodeIdUnknown	See Table 166 for the description of this result code.
Bad_AttributeIdInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeInvalid	See Table 166 for the description of this result code.
Bad_IndexRangeNoData	See Table 166 for the description of this result code. If the *ArrayDimensions* have a fixed length that can not change and no data exists within the range of indexes specified, Bad_IndexRangeNoData is returned in *CreateMonitoredItems*. Otherwise if the length of the array is dynamic, the *Server* shall return this status in a *Publish* response for the *MonitoredItem* if no data exists within the range.
Bad_DataEncodingInvalid	See Table 166 for the description of this result code.
Bad_DataEncodingUnsupported	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterUnsupported	See Table 166 for the description of this result code.
Bad_FilterNotAllowed	See Table 165 for the description of this result code.
Bad_TooManyMonitoredItems	The *Server* has reached its maximum number of monitored items.

5.12.3 ModifyMonitoredItems

5.12.3.1 Description

This *Service* is used to modify *MonitoredItems* of a *Subscription*. Changes to the *MonitoredItem* settings shall be applied immediately by the *Server*. They take effect as soon as practical but not later than twice the new *revisedSamplingInterval*.

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

Illegal request values for parameters that can be revised do not generate errors. Instead the server will choose default values and indicate them in the corresponding revised parameter.

5.12.3.2 Parameters

Table 67 defines the parameters for the *Service*.

Table 67 – ModifyMonitoredItems Service Parameters

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

5.12.3.3 Service results

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

Table 68 – ModifyMonitoredItems Service Result Codes

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

5.12.3.4 StatusCodes

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

Table 69 – ModifyMonitoredItems Operation Level Result Codes

Symbolic Id	Description
Bad_MonitoredItemIdInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterInvalid	See Table 166 for the description of this result code.
Bad_MonitoredItemFilterUnsupported	See Table 166 for the description of this result code.
Bad_FilterNotAllowed	See Table 165 for the description of this result code.

5.12.4 SetMonitoringMode

5.12.4.1 Description

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

5.12.4.2 Parameters

Table 70 defines the parameters for the *Service*.

Table 70 – SetMonitoringMode Service Parameters

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

5.12.4.3 Service results

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

Table 71 – SetMonitoringMode Service Result Codes

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

5.12.4.4 StatusCodes

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

Table 72 – SetMonitoringMode Operation Level Result Codes

Value	Description
Bad_MonitoredItemIdInvalid	See Table 166 for the description of this result code.

5.12.5 SetTriggering

5.12.5.1 Description

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

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

See 5.12.1.6 for a description of the triggering model.

5.12.5.2 Parameters

Table 73 defines the parameters for the *Service*.

Table 73 – SetTriggering Service Parameters

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

5.12.5.3 Service results

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

Table 74 – SetTriggering Service Result Codes

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

5.12.5.4 StatusCodes

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

Table 75 – SetTriggering Operation Level Result Codes

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

5.12.6 DeleteMonitoredItems

5.12.6.1 Description

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

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

5.12.6.2 Parameters

Table 76 defines the parameters for the *Service*.

Table 76 – DeleteMonitoredItems Service Parameters

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

5.12.6.3 Service results

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

Table 77 – DeleteMonitoredItems Service Result Codes

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

5.12.6.4 StatusCodes

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

Table 78 – DeleteMonitoredItems Operation Level Result Codes

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

5.13 Subscription Service Set

5.13.1 Subscription model

5.13.1.1 Description

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

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 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 de-queued and processed by a subscription related to this *Session* for each publishing cycle, if there are *Notifications* to report. When there are not, the *Publish* request is not de-queued 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 de-queued 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 clauses 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 available to send a *Publish* response for the *Subscription*. Any *Service* call that uses the *SubscriptionId* or the processing of a *Publish* response resets the lifetime counter of this *Subscription*. When this counter reaches the value calculated for the lifetime of a *Subscription* based on the MaxKeepAliveCount parameter in the *CreateSubscription Service* (5.13.2), the *Subscription* is closed. Closing the *Subscription* causes its *MonitoredItems* to be deleted. In addition the *Server* shall issue a *StatusChangeNotification notificationMessage* with the status code Bad_Timeout. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4.

i) *Sessions* maintain a retransmission queue of sent *NotificationMessages*. *NotificationMessages* are retained in this queue until they are acknowledged. The *Session* shall maintain a retransmission queue size of at least two times the number of *Publish* requests per *Session* the *Server* supports. The minimum size of the retransmission queue may be changed by a *Profile* in IEC 62541-7. The minimum number of *Publish* requests per *Session* the *Server* shall support is defined in IEC 62541-7. *Clients* are required to acknowledge *NotificationMessages* as they are received. In the case of a retransmission queue overflow, the oldest sent *NotificationMessage* gets deleted. If a *Subscription* is transferred to another *Session*, the queued *NotificationMessages* for this *Subscription* are moved from the old to the new *Session*.

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

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

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

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

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

5.13.1.2 State table

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

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

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

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 de-queues and processes a *Publish* request. When a *Subscription* processes a *Publish* request, it accesses the queues of its *MonitoredItems* and de-queues 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 *Notifications* have been generated, a *Publish* request is de-queued and a *NotificationMessage* is returned in the response. However, if the publishing timer expires without a *Notification* becoming available, a *Publish* request is de-queued and a keep-alive *Message* is returned in the response. The *Subscription* then returns to the normal state of waiting for the publishing timer to expire again. If, in either of these cases, there are no *Publish* requests queued, the *Subscription* waits for the next *Publish* request to be received, as described in case (b).

The *Subscription* states are defined in Table 79.

Table 79 – Subscription States

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

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

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 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 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 80 – Subscription State Table

#	Current State	Event/Conditions	Action	Next State
1	CLOSED	Receive CreateSubscription Request	CreateSubscription()	CREATING
2	CREATING	CreateSubscription fails	ReturnNegativeResponse()	CLOSED
3	CREATING	CreateSubscription succeeds	InitializeSubscription() MessageSent = FALSE ReturnResponse()	NORMAL
4	NORMAL	Receive *Publish* Request && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && MoreNotifications == FALSE))	DeleteAckedNotificationMsgs() EnqueuePublishingReq()	NORMAL
5	NORMAL	Receive *Publish* Request && PublishingEnabled == TRUE && MoreNotifications == TRUE	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
6	NORMAL	PublishingTimer Expires && PublishingReqQueued == TRUE && PublishingEnabled == TRUE && NotificationsAvailable == TRUE	ResetLifetimeCounter() StartPublishingTimer() DequeuePublishReq() ReturnNotifications() MessageSent == TRUE	NORMAL
7	NORMAL	PublishingTimer Expires && PublishingReqQueued == TRUE && MessageSent == FALSE && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE))	ResetLifetimeCounter() StartPublishingTimer() DequeuePublishReq() ReturnKeepAlive() MessageSent == TRUE	NORMAL
8	NORMAL	PublishingTimer Expires && PublishingReqQueued == FALSE && (MessageSent == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == TRUE))	StartPublishingTimer()	LATE
9	NORMAL	PublishingTimer Expires && MessageSent == TRUE && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE)	StartPublishingTimer() ResetKeepAliveCounter()	KEEPALIVE

#	Current State	Event/Conditions	Action	Next State
		)		
10	LATE	Receive *Publish* Request && PublishingEnabled == TRUE && (NotificationsAvailable == TRUE || MoreNotifications == TRUE)	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
11	LATE	Receive *Publish* Request && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE && MoreNotifications == FALSE))	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnKeepAlive() MessageSent = TRUE	KEEPALIVE
12	LATE	PublishingTimer Expires	StartPublishingTimer()	LATE
13	KEEPALIVE	Receive *Publish* Request	DeleteAckedNotificationMsgs() EnqueuePublishingReq()	KEEPALIVE
14	KEEPALIVE	PublishingTimer Expires && PublishingEnabled == TRUE && NotificationsAvailable == TRUE && PublishingReqQueued == TRUE	ResetLifetimeCounter() 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
18	NORMAL || LATE || KEEPALIVE	Receive ModifySubscription Request	ResetLifetimeCounter() UpdateSubscriptionParams() ReturnResponse()	SAME
19	NORMAL || LATE || KEEPALIVE	Receive SetPublishingMode Request	ResetLifetimeCounter() SetPublishingEnabled() MoreNotifications = FALSE ReturnResponse()	SAME
20	NORMAL || LATE || KEEPALIVE	Receive Republish Request && RequestedMessageFound == TRUE	ResetLifetimeCounter() ReturnResponse()	SAME
21	NORMAL || LATE || KEEPALIVE	Receive Republish Request && RequestedMessageFound == FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
22	NORMAL || LATE || KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == FALSE	ResetLifetimeCounter() ReturnNegativeResponse ()	SAME

#	Current State	Event/Conditions	Action	Next State
23	NORMAL || LATE || KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == TRUE && ClientValidated() ==TRUE	SetSession() ResetLifetimeCounter() ReturnResponse() IssueStatusChangeNotification()	SAME
24	NORMAL || LATE || KEEPALIVE	Receive TransferSubscriptions Request && SessionChanged() == TRUE && ClientValidated() == FALSE	ReturnNegativeResponse()	SAME
25	NORMAL || LATE || KEEPALIVE	Receive DeleteSubscriptions Request && SubscriptionAssignedToClient ==TRUE	DeleteMonitoredItems() DeleteClientPublReqQueue()	CLOSED
26	NORMAL || LATE || KEEPALIVE	Receive DeleteSubscriptions Request && SubscriptionAssignedToClient ==FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
27	NORMAL || LATE || KEEPALIVE	LifetimeCounter == 1 The LifetimeCounter is decremented if PublishingTimer expires and PublishingReqQueued == FALSE The LifetimeCounter is reset if PublishingReqQueued == TRUE.	DeleteMonitoredItems() IssueStatusChangeNotification()	CLOSED

5.13.1.3 State Variables and parameters

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

Table 81 – State variables and parameters

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

5.13.1.4 Functions

The action functions are defined alphabetically in Table 82.

Table 82 – Functions

Function	Description
ClientValidated()	A boolean function that returns TRUE only when the *Client* that is submitting a TransferSubscriptions request is operating on behalf of the same user and supports the same *Profiles* as the *Client* of the previous *Session*.
CreateNotificationMsg()	Increment the SeqNum and create a *NotificationMessage* from the *MonitoredItems* assigned to the *Subscription*. Save the newly-created *NotificationMessage* in the retransmission queue. If all available *Notifications* can be sent in the *Publish* response, the MoreNotifications state *Variable* is set to FALSE. Otherwise, it is set to TRUE.
CreateSubscription()	Attempt to create the *Subscription*.
DeleteAckedNotificationMsgs()	Delete the *NotificationMessages* from the retransmission queue that were acknowledged by the request.
DeleteClientPublReqQueue()	Clear the *Publish* request queue for the *Client* that is sending the DeleteSubscriptions request, if there are no more *Subscriptions* assigned to that *Client*.
DeleteMonitoredItems()	Delete all *MonitoredItems* assigned to the *Subscription*.
DequeuePublishReq()	De-queue a publishing request in first-in first-out order. Validate if the publish request is still valid by checking the timeoutHint in the RequestHeader. If the request timed out, send a *Bad_Timeout* service result for the request and de-queue another publish request. ResetLifetimeCounter()
EnqueuePublishingReq()	Enqueue the publishing request.
InitializeSubscription()	ResetLifetimeCounter() MoreNotifications = FALSE PublishRateChange = FALSE PublishingEnabled = value of publishingEnabled parameter in the CreateSubscription request PublishingReqQueued = FALSE SeqNum = 0 SetSession() StartPublishingTimer()
IssueStatusChangeNotification()	Issue a *StatusChangeNotification notificationMessage* with a status code for the status change of the Subscription. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4. Bad_Timeout status code is used if the lifetime expires and Good_SubscriptionTransferred is used if the Subscriptions was transferred to another Session.
ResetKeepAliveCounter()	Reset the keep-alive counter to the maximum keep-alive count of the *Subscription*. The maximum keep-alive count is set by the *Client* when the *Subscription* is created and may be modified using the ModifySubscription *Service*.
ResetLifetimeCounter()	Reset the LifetimeCounter *Variable* to the value specified for the lifetime of a *Subscription* in the CreateSubscription *Service* (5.13.2).
ReturnKeepAlive()	CreateKeepAliveMsg() ReturnResponse()
ReturnNegativeResponse ()	Return a *Service* response indicating the appropriate *Service* level error. No parameters are returned other than the responseHeader that contains the *Service* level *StatusCode*.

ReturnNotifications()	CreateNotificationMsg()
	ReturnResponse()
	If (MoreNotifications == TRUE) && (PublishingReqQueued == TRUE)
	{
	DequeuePublishReq()
	Loop through this function again
	}

ReturnResponse()	Return the appropriate response, setting the appropriate parameter values and *StatusCodes* defined for the *Service*.
SessionChanged()	A boolean function that returns TRUE only when the *Session* used to send a TransferSubscriptions request is different from 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.

Illegal request values for parameters that can be revised do not generate errors. Instead the server will choose default values and indicate them in the corresponding revised parameter.

5.13.2.2 Parameters

Table 83 defines the parameters for the *Service*.

Table 83 – CreateSubscription Service Parameters

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

Name	Type	Description
Request		
revisedMaxKeepAliveCount	Counter	The actual maximum keep-alive count (see 7.5 for *Counter* 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.

5.13.2.3 Service results

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

Table 84 – CreateSubscription Service Result Codes

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

5.13.3 ModifySubscription

5.13.3.1 Description

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

Illegal request values for parameters that can be revised do not generate errors. Instead the server will choose default values and indicate them in the corresponding revised parameter.

Changes to the *Subscription* settings shall be applied immediately by the *Server*. They take effect as soon as practical but not later than twice the new *revisedPublishingInterval*.

5.13.3.2 Parameters

Table 85 defines the parameters for the *Service*.

Table 85 – ModifySubscription Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.26 for *RequestHeader* definition).
subscriptionId	IntegerId	The *Server*-assigned identifier for the *Subscription* (see 7.13 for *IntegerId* definition).
requestedPublishingInterval	Duration	This interval defines the cyclic rate that the *Subscription* is being requested to return *Notifications* to the *Client*. This interval is expressed in milliseconds. This interval is represented by the publishing timer in the *Subscription* state table (see 5.13.1.2). The negotiated value for this parameter returned in the response is used as the default sampling interval for *MonitoredItems* assigned to this *Subscription*. If the requested value is 0 or negative, the server shall revise with the fastest supported publishing interval.
requestedLifetimeCount	Counter	Requested lifetime count (see 7.5 for *Counter* definition). The lifetime count shall be a minimum of three times the keep keep-alive count. When the publishing timer has expired this number of times without a *Publish* request being available to send a *NotificationMessage*, then the *Subscription* shall be deleted by the *Server*.
requestedMaxKeepAliveCount	Counter	Requested maximum keep-alive count (see 7.5 for *Counter* definition). When the publishing timer has expired this number of times without requiring any *NotificationMessage* to be sent, the *Subscription* sends a keep-alive *Message* to the *Client*. The negotiated value for this parameter is returned in the response. If the requested value is 0, the server shall revise with the smallest supported keep-alive count.
maxNotificationsPerPublish	Counter	The maximum number of notifications that the *Client* wishes to receive in a single *Publish* response. A value of zero indicates that there is no limit.
priority	Byte	Indicates the relative priority of the *Subscription*. When more than one *Subscription* needs to send *Notifications*, the *Server* should de-queue a Publish request to the *Subscription* with the highest *priority* number. For

Name	Type	Description
Request		
		Subscriptions with equal *priority* the *Server* should de-queue Publish requests in a round-robin fashion. Any *Subscription* that needs to send a keep-alive *Message* shall take precedence regardless of its *priority*, in order to prevent the *Subscription* from expiring. A *Client* that does not require special priority settings should set this value to zero.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.27 for *ResponseHeader* definition).
revisedPublishingInterval	Duration	The actual publishing interval that the *Server* will use, expressed in milliseconds. The *Server* should attempt to honour the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.
revisedLifetimeCount	Counter	The lifetime of the *Subscription* shall be a minimum of three times the keep-alive interval negotiated by the *Server*.
revisedMaxKeepAliveCount	Counter	The actual maximum keep-alive count (see 7.5 for *Counter* definition). The *Server* should attempt to honour the *Client* request for this parameter, but may negotiate this value up or down to meet its own constraints.

5.13.3.3 Service results

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

Table 86 – ModifySubscription Service Result Codes

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

5.13.4 SetPublishingMode

5.13.4.1 Description

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

5.13.4.2 Parameters

Table 87 defines the parameters for the *Service*.

Table 87 – SetPublishingMode Service Parameters

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

5.13.4.3 Service results

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

Table 88 – SetPublishingMode Service Result Codes

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

5.13.4.4 StatusCodes

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

Table 89 – SetPublishingMode Operation Level Result Codes

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

5.13.5 Publish

5.13.5.1 Description

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

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

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

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

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

occurs then the *Client* will have to adjust its message processing limits or the parameters for the *Subscription and/or MonitoredItems*.

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

5.13.5.2 Parameters

Table 90 defines the parameters for the *Service*.

Table 90 – Publish Service Parameters

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

5.13.5.3 Service results

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

Table 91 – Publish Service Result Codes

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

5.13.5.4 StatusCodes

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

Table 92 – Publish Operation Level Result Codes

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

5.13.6 Republish

5.13.6.1 Description

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

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

See 6.5 for a description of the issues and strategies regarding reconnect handling and *Republish*.

5.13.6.2 Parameters

Table 93 defines the parameters for the *Service*.

Table 93 – Republish Service Parameters

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

5.13.6.3 Service results

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

Table 94 – Republish Service Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.
Bad_MessageNotAvailable	The requested message is no longer available.

5.13.7 TransferSubscriptions

5.13.7.1 Description

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

The *authenticationToken* contained in the request header identifies the *Session* to which the *Subscription* and *MonitoredItems* shall be transferred. The *Server* shall validate that the *Client* of that *Session* is operating on behalf of the same user and that the potentially new *Client* supports the *Profiles* that are necessary for the *Subscription*. If the *Server* transfers the

Subscription, it returns the sequence numbers of the *NotificationMessages* that are available for retransmission. The *Client* should acknowledge all *Messages* in this list for which it will not request retransmission.

If the *Server* transfers the *Subscription* to the new *Session*, the *Server* shall issue a *StatusChangeNotification notificationMessage* with the status code Good_SubscriptionTransferred to the old *Session*. The *StatusChangeNotification notificationMessage* type is defined in 7.19.4.

5.13.7.2 Parameters

Table 95 defines the parameters for the *Service*.

Table 95 – TransferSubscriptions Service Parameters

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

5.13.7.3 Service results

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

Table 96 – TransferSubscriptions Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 165 for the description of this result code.
Bad_TooManyOperations	See Table 165 for the description of this result code.
Bad_InsufficientClientProfile	The *Client* of the current *Session* does not support one or more *Profiles* that are necessary for the *Subscription*.

5.13.7.4 StatusCodes

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

Table 97 – TransferSubscriptions Operation Level Result Codes

Symbolic Id	Description
Bad_SubscriptionIdInvalid	See Table 165 for the description of this result code.
Bad_UserAccessDenied	See Table 165 for the description of this result code. The *Client* of the current *Session* is not operating on behalf of the same user as the *Session* that owns the *Subscription*.

5.13.8 DeleteSubscriptions

5.13.8.1 Description

This *Service* is invoked to delete one or more *Subscriptions* that belong to the *Client's Session*.

Successful completion of this *Service* causes all *MonitoredItems* that use the *Subscription* to be deleted. If this is the last *Subscription* for the *Session*, then all *Publish* requests still queued for that *Session* are de-queued and shall be returned with Bad_NoSubscription.

Subscriptions that were transferred to another *Session* shall be deleted by the *Client* that owns the *Session*.

5.13.8.2 Parameters

Table 98 defines the parameters for the *Service*.

Table 98 – DeleteSubscriptions Service Parameters

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

5.13.8.3 Service results

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

Table 99 – DeleteSubscriptions Service Result Codes

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

5.13.8.4 StatusCodes

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

Table 100 – DeleteSubscriptions Operation Level Result Codes

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

6 Service behaviours

6.1 Security

6.1.1 Overview

The OPC UA *Services* define a number of mechanisms to meet the security requirements outlined in IEC TR 62541-2. This clause describes a number of important security-related procedures that *OPC UA Applications* shall follow.

6.1.2 Obtaining and Installing an Application Instance Certificate

All *OPC UA Applications* require an *Application Instance Certificate* which shall contain the following information:

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

In addition, each *Application Instance Certificate* has a private key which should be stored in a location that can only be accessed by the application. If this private key is compromised, the administrator shall assign a new *Application Instance Certificate* and private key to the application.

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

If the administrator responsible for the application decides that a self-signed *Certificate* does not meet the security requirements of the organisation, then the administrator should install a *Certificate* issued by a *Certification Authority*. The steps involved in requesting an *Application Instance Certificate* from a *Certificate Authority* are shown in Figure 19.

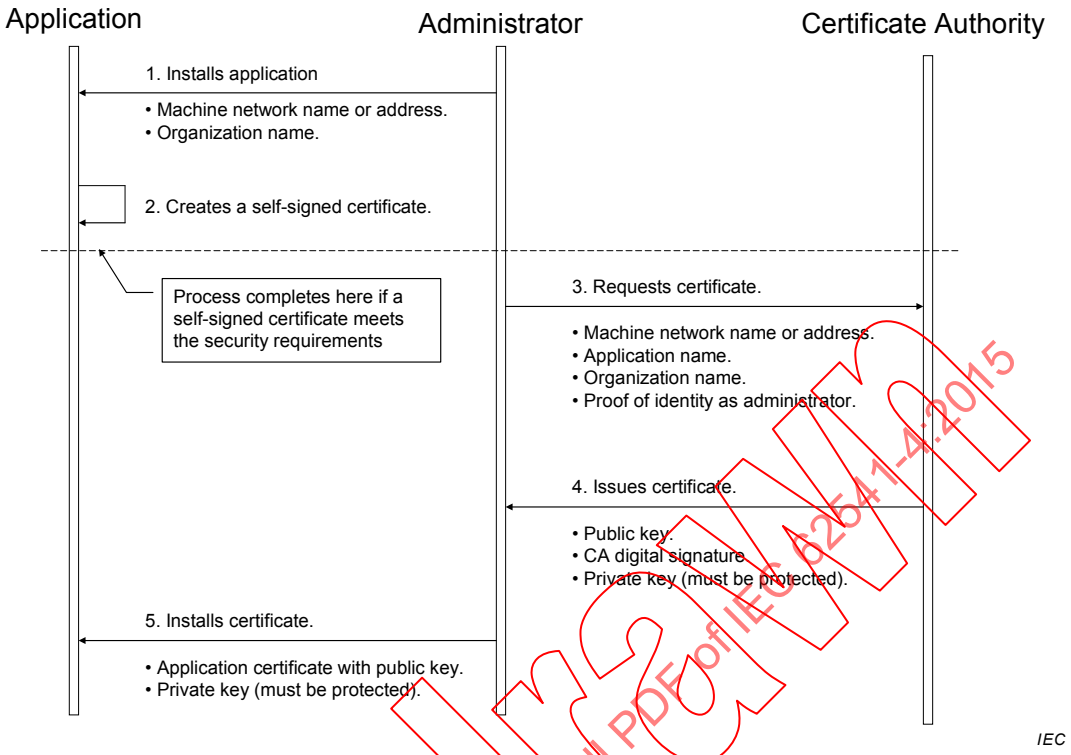


Figure 19 – Obtaining and Installing an Application Instance Certificate

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

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

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

Vendors may choose to automate the process of acquiring *Certificates* from an authority. If this is the case, the *Administrator* would still go through the steps illustrated in Figure 19, however, the installation program for the application would do them automatically and only prompt the *Administrator* to provide information about the application instance being installed.

6.1.3 Determining if a Certificate is Trusted

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

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

Building a trust chain requires access to all *Certificates* in the chain. These *Certificates* may be stored locally or they may be provided with the application *Certificate*. Processing fails with Bad_SecurityChecksFailed if a CA *Certificate* cannot be found.

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

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

Table 101 – Certificate Validation Steps

Step	Error/AuditEvent	Description
Certificate Structure	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	The *Certificate* structure is verified. This error may not be suppressed.
Signature	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	A *Certificate* with an invalid signature shall always be rejected. A *Certificate* signature is invalid if the *Issuer Certificate* is unknown. A self-signed *Certificate* is its own issuer.
Trust List Check	Bad_SecurityChecksFailed AuditCertificateUntrustedEventType	If the *Application Instance Certificate* is not trusted and none of the CA *Certificates* in the chain is trusted, the result of the *Certificate* validation shall be Bad_SecurityChecksFailed.
Validity Period	Bad_CertificateTimeInvalid Bad_CertificateIssuerTimeInvalid AuditCertificateExpiredEventType	The current time shall be after the start of the validity period and before the end. This error may be suppressed.
Host Name	Bad_CertificateHostNameInvalid AuditCertificateDataMismatchEventType	The HostName in the URL used to connect to the Server shall be the same as one of the HostNames specified in the *Certificate*. This check is skipped for CA *Certificates*. This error may be suppressed.
URI	Bad_CertificateUriInvalid AuditCertificateDataMismatchEventType	*Application and Software Certificates* contain an application or product URI that shall match the URI specified in the *ApplicationDescription* provided with the *Certificate*. This check is skipped for CA *Certificates*. This error may not be suppressed. The *gatewayServerUri* is used to validate an *Application Certificate* when connecting to a *Gateway Server* (see 7.1).
Certificate Usage	Bad_CertificateUseNotAllowed Bad_CertificateIssuerUseNotAllowed AuditCertificateMismatchEventType	Each *Certificate* has a set of uses for the *Certificate* (see IEC 62541-6). These uses shall match use requested for the *Certificate* (i.e. Application, Software or CA). This error may be suppressed unless the *Certificate* indicates that the usage is mandatory.
Find Revocation List	Bad_CertificateRevocationUnknown Bad_CertificateIssuerRevocationUnknown AuditCertificateRevokedEventType	Each CA *Certificate* may have a revocation list. This check fails if this list is not available (i.e. a network interruption prevents the application from accessing the list). No error is reported if the *Administrator* disables revocation checks for a CA *Certificate*. This error may be suppressed.
Revocation Check	Bad_CertificateRevoked Bad_CertificateIssuerRevoked AuditCertificateRevokedEventType	The *Certificate* has been revoked and may not be used. This error may not be suppressed.

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

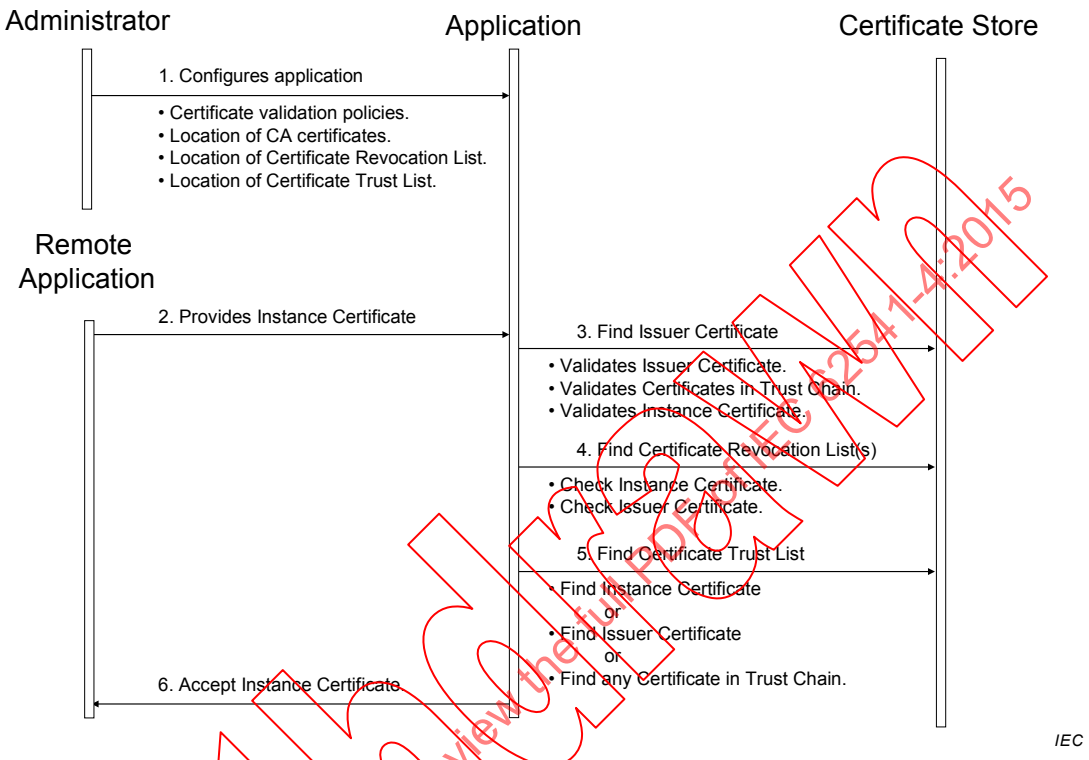


Figure 20 – Determining if a Application Instance Certificate is Trusted

6.1.4 Creating a SecureChannel

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

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

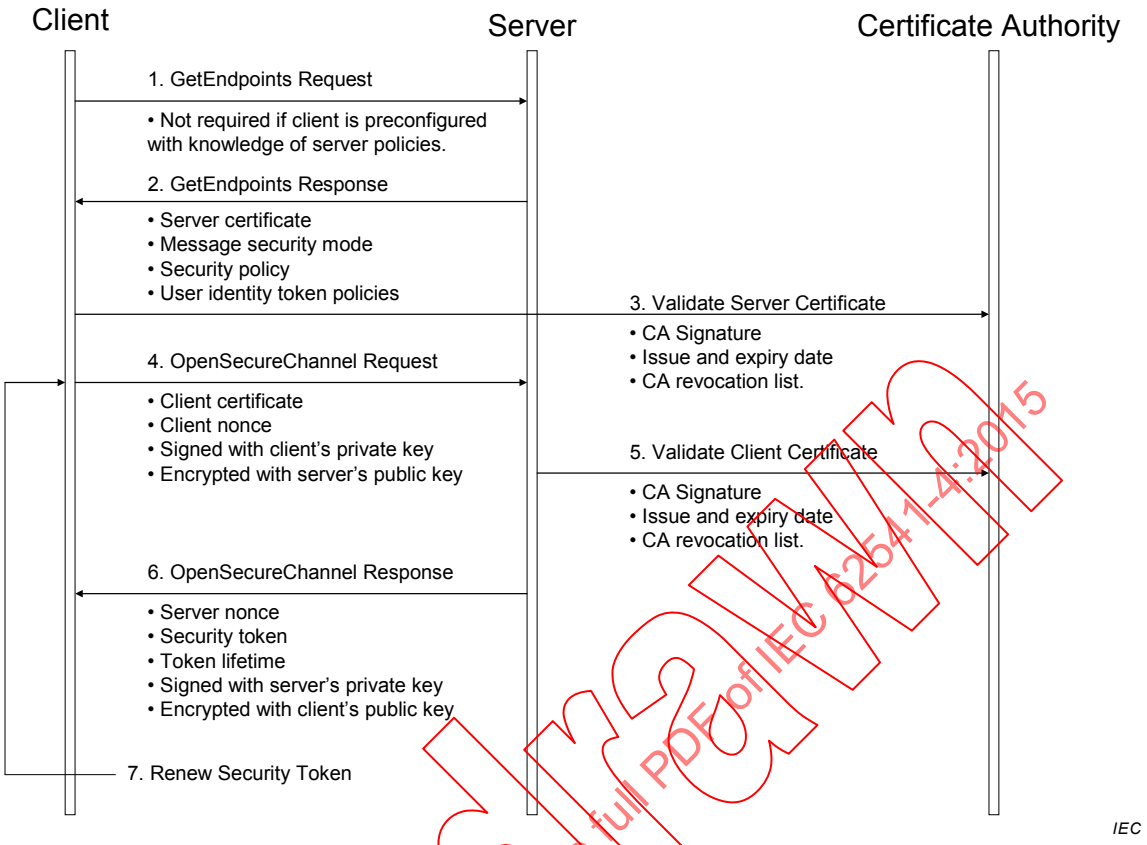


Figure 21 – Establishing a SecureChannel

Figure 21 above assumes *Client* and *Server* have online access to a *Certificate*Authority (CA). If online access is not available and if the administrator has installed the CA public key on the local machine, then the *Client* and *Server* shall still validate the application *Certificates* using that key. The figure shows only one CA, however, there is no requirement that the *Client* and *Server Certificates* be issued by the same authority. A self-signed *Application Instance Certificate* does not need to be verified with a CA. Any *Certificate* shall be rejected if it is not in a trust list provided by the administrator.

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

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

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

There are two security risks which a *Client* shall be aware of when using the *GetEndpoints Service*. The first could come from a rogue *Discovery Server* that tries to direct the *Client* to a rogue *Server*. For this reason the *Client* shall verify that the *ServerCertificate* in the *EndpointDescription* is a trusted *Certificate* before it calls *CreateSession*.

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

The exact mechanisms for using the security token to sign and encrypt *Messages* exchanged over the *SecureChannel* are described in IEC 62541-6. The process for renewing tokens is also described in detail in IEC 62541-6.

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

6.1.5 Creating a Session

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

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

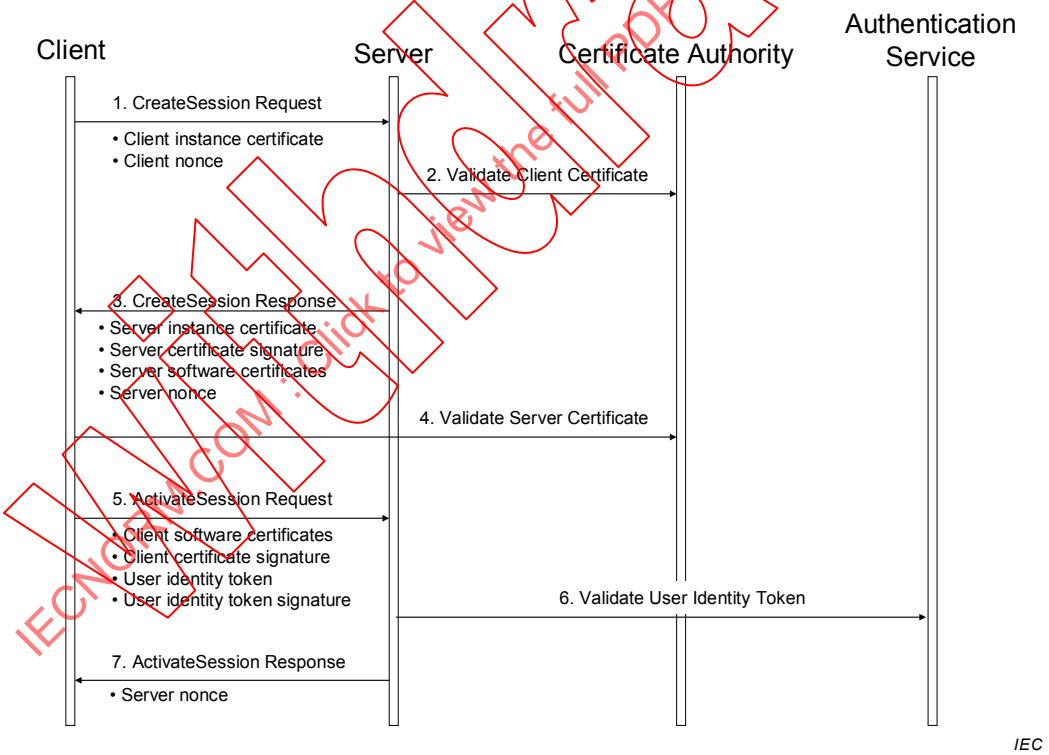


Figure 22 – Establishing a Session

Figure 22 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 Instance Certificates*. If the *Client* or *Server* does not have online access to the CA, then they shall validate the *Application Instance Certificates* using the CA public key that the administrator shall install on the local machine.

The authentication service is a central database that can verify that user token provided by the *Client*. This authentication service may also tell the *Server* which access rights the user has. The authentication service depends on the user identity token. It could be a *Certificate*

Authority, a Kerberos ticket granting service, a WS-Trust *Server* or a proprietary database of some sort.

The *Client* and *Server* shall prove possession of their *Application Instance Certificates* by signing the *Certificates* with a nonce appended. The exact mechanism used to create the proof of possession signatures is described in 5.6.2. Similarly, the *Client* shall prove possession of some types of user identity tokens by creating signatures with the secret associated with the token.

6.1.6 Impersonating a User

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

The steps involved in impersonating a user are shown in Figure 23.

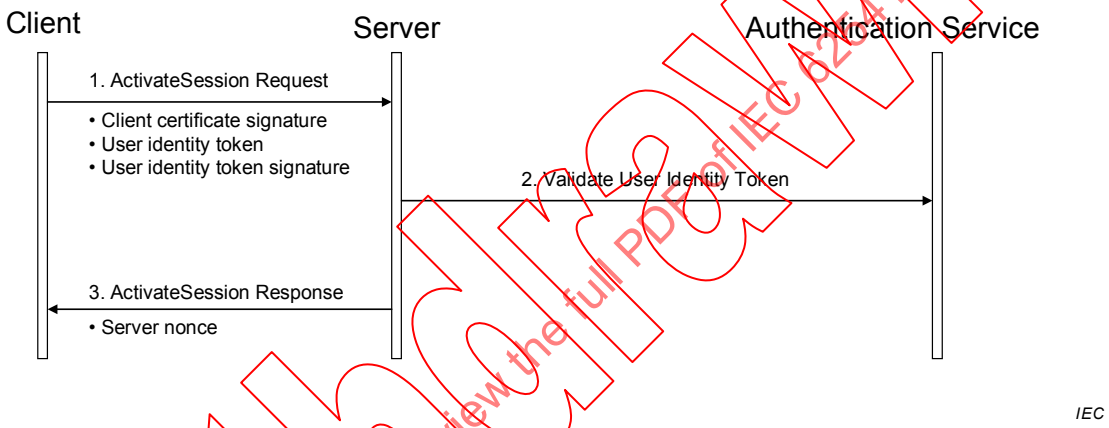


Figure 23 – Impersonating a User

6.2 Software Certificates

6.2.1 Overview

All *OPC UA Applications* may have one or more software *Certificates* that are issued by certification authorities and specify the profiles that the application supports. This clause describes how *SoftwareCertificates* are obtained, installed and validated.

SoftwareCertificates are used to identify a software product that is an *OPC UA Application* and to describe the capabilities of the product. The capabilities are described through *Profiles* defined in IEC 62541-7. *SoftwareCertificates* are not relevant for security in OPC UA.

6.2.2 Obtaining and Installing a Software Certificate

The software *Certificates* contain the following information:

- The name of the vendor responsible for the product;
- The name of the product;
- The URI for the product;
- The software version and build number;
- The vendor product *Certificate*;
- A list of profiles supported by the application;
- The certification testing status for each supported profile;

- The name of the testing authority that issued the *Certificate*;
- The issue and expiry date for the *Certificate*;
- The public key issued to the application by the certifying authority;
- A digital signature created by the certifying authority.

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

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

The steps involved in acquiring and installing a software *Certificate* from a *Certificate Authority* are shown in Figure 24.

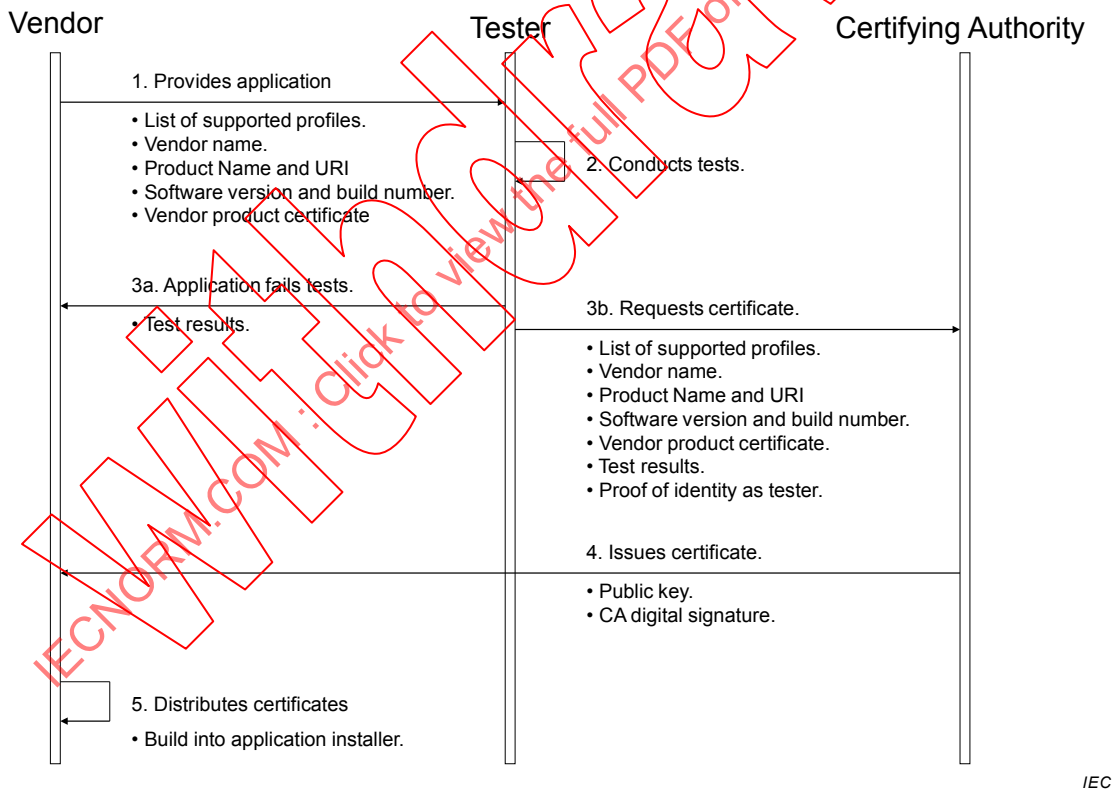


Figure 24 – Obtaining and Installing a Software Certificate

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

The certifying authority will issue *Certificates* only to vendors it trusts. For that reason, the tester shall provide proof of identity before the certifying authority will issue a *Certificate*.

6.2.3 Validating a Software Certificate

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

A *SoftwareCertificate* is valid if:

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

The steps used to validate *SoftwareCertificates* are almost the same as the steps described for *Application Instance Certificates* in 6.1.3. The only difference is *SoftwareCertificates* are not checked against the *TrustList* for the *Application.*

SoftwareCertificates always contain an *Application Instance Certificates* which is owned by the vendor that distributes the application. The application *Certificate* may also be in the *Certificate* chain used to issue *Application Instance Certificates*. For this reason, *OPC UA Applications* may reject *SoftwareCertificates* provided by applications if the application *Certificate* is not part of the *Certificate* chain for the *Application Instance Certificate*. *OPC UA Applications* should allow *Administrators* to require this behaviour. Profiles defined in IEC 62541-7 may further specify expected *Certificate* handling behaviours.

6.3 Auditing

6.3.1 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 IEC TR 62541-2. This subclause describes what is expected of an OPC UA *Server* and *Client* with respect to auditing and it details the audit requirements for each service set. Auditing can be accomplished using one or both of the following methods:

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

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

6.3.2 General audit logs

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

6.3.3 General audit Events

A *Server* that maintains an audit log shall provide the audit log entries via *Event Messages*. The *AuditEventType* and its sub-types are defined in IEC 62541-3. An audit *Event Message* also includes the audit record Id. The details of the *AuditEventType* and its subtypes are defined in IEC 62541-5. A *Server* that is an aggregating *Server* that supports auditing shall also subscribe for audit events for all of the *Servers* that it is aggregating (assuming they provide auditing). The combined stream should be available from the aggregating *Server*.

6.3.4 Auditing for Discovery Service Set

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

6.3.5 Auditing for SecureChannel Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for failed service invocations and for successful invocation of the *OpenSecureChannel* and *CloseSecureChannel* Services. The *Client* generated audit entries should be setup prior to the actual call, allowing the correct audit record Id to be provided. The *OpenSecureChannel* Service shall generate an audit *Event* of type *AuditOpenSecureChannelEventType* or a subtype of it for the *requestType* ISSUE_0. Audit *Events* for the *requestType* RENEW_1 are only created if the renew fails. The *CloseSecureChannel* service shall generate an audit *Event* of type *AuditChannelEventType* or a subtype of it. Both of these *Event* types are subtypes of the *AuditChannelEventType*. See IEC 62541-5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure cases the *Message* for *Events* of this type should include a description of why the service failed. This description should be more detailed than what was returned to the client. From a security point of view a client only needs to know that it failed, but from an *Auditing* point of view the exact details of the failure need to be known. In the case of *Certificate* validation errors the description should include the audit *EventId* of the specific *AuditCertificateEventType* that was generated to report the *Certificate* error. The *AuditCertifcateEvent* shall also contain the detailed *Certificate* validation error. The additional parameters should include the details of the request. It is understood that these events may be generated by the underlying *Communication Stacks* in many cases, but they shall be made available to the *Server* and the *Server* shall report them.

6.3.6 Auditing for Session Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditSessionEventType* or a subtype of it. In particular, they shall generate the base *EventType* or the appropriate subtype, depending on the service that was invoked. The *CreateSession* service shall generate *AuditCreateSessionEventType* events or sub-types of it. The *ActivateSession* service shall generate *AuditActivateSessionEventType* events or subtypes of it. When the *ActivateSession Service* is called to change the user identity then the server shall generate *AuditActivateSessionEventType* events or subtypes of it. The CloseSession service shall generate the base *EventType* of *AuditSessionEventType* or subtypes of it. See IEC 62541-5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the *Service* failed. The additional parameters should include the details of the request.

This *Service Set* shall also generate additional audit events in the cases when *Certificate* validation errors occur. These audit events are generated in addition to the *AuditSessionEventTypes*. See IEC 62541-3 for the definition of *AuditCertificateEventType* and its subtypes.

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

6.3.7 Auditing for NodeManagement Service Set

All Services in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditNodeManagementEventType* or subtypes of it. See IEC 62541-5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for *Events* of this type should include a description of why the service failed. The additional parameters should include the details of the request.

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

6.3.8 Auditing for Attribute Service Set

The *Write* or *HistoryUpdate* Services in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed *Service* invocations. These *Services* shall generate an audit *Event* of type *AuditUpdateEventType* or subtypes of it. In particular, the *Write Service* shall generate an audit event of type *AuditWriteUpdateEventType* or a subtype of it. The *HistoryUpdate Service* shall generate an audit *Event* of type *AuditHistoryUpdateEvent* or a subtype of it. Three subtypes of *AuditHistoryUpdateEvent* are defined as *AuditHistoryEventUpdateEventType*, *AuditHistoryValueUpdateEventType* and *AuditHistoryDeleteEventType*. The subtype depends on the type of operation being performed, historical event update, historical data value update or a historical delete. See IEC 62541-5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case the *Message* for *Events* of this type should include a description of why the *Service* failed. The additional parameters should include the details of the request.

The *Read* and *HistoryRead Services* may generate audit entries and audit *Events* for failed *Service* invocations. These *Services* should generate an audit *Event* of type *AuditEventType* or a subtype of it. See IEC 62541-5 for the detailed assignment of the *SourceNode*, *SourceName* and additional parameters. The *Message* for *Events* of this type should include a description of why the *Service* failed.

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

6.3.9 Auditing for Method Service Set

All *Services* in this *Service Set* for *Servers* that support auditing may generate audit entries and shall generate audit *Events* for both successful and failed service invocations if the invocation modifies the address space, writes a value or modifies the state of the system (alarm acknowledge, batch sequencing or other system changes). These method calls shall generate an audit *Event* of type *AuditUpdateMethodEventType* or subtypes of it. Methods that do not modify the address space, write values or modify the state of the system may generate events. See IEC 62541-5 for the detailed assignment of the *SourceNode*, *SourceName* and additional parameters.

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

6.3.10 Auditing for View, Query, MonitoredItem and Subscription Service Set

All of the *Services* in these four *Service Sets* only provide the *Client* with information, with the exception of the *TransferSubscriptions Service* in the *Subscription Service Set*. In general, these services will not generate audit entries or audit Event *Messages*. The *TransferSubscriptions Service* shall generate an audit *Event* of type *AuditSessionEventType* or subtypes of it for both successful and failed *Service* invocations. See IEC 62541-5 for the detailed assignment of the *SourceNode*, the *SourceName* and additional parameters. For the failure case, the *Message* for Events of this type should include a description of why the service failed.

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

6.4 Redundancy

6.4.1 Redundancy overview

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

6.4.2 Server redundancy overview

6.4.2.1 General

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

The *ServerRedundancy Object* defined in IEC 62541-5 indicates the mode supported by the *Server*. The *ServerRedundancyType* ObjectType and its subtypes *TransparentRedundancyType* and *NonTransparentRedundancyType* defined in IEC 62541-5 specify information for the supported redundancy mode.

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.

Independent of the used redundancy mode it is expected that all *Servers* in the redundant set have an identical address space including identical *NodeIds* and the identical logic for setting the service level.

6.4.2.2 Transparent redundancy

For transparent redundancy, all that OPC UA provides is the data structures to allow the *Client* to identify which *Servers* are in the redundant set, what the service level of each *Server* is and which *Server* is currently responsible for the *Client Session*. This information is specified in *TransparentRedundancyType ObjectType* defined in IEC 62541-5.

All OPC UA interactions within a given session shall be supported by one *Server* and the *Client* is able to identify which *Server* that is, allowing a complete audit trail for the data. It is the responsibility of the *Servers* to ensure that information is synchronised between the *Servers*. A functional *Server* will take over the *Session* and *Subscriptions* from the failed

Server. Failover may require a transport layer reconnect of the *Client* but the *Endpoint* URL of the *Server* shall not change. See 6.5 for more details on re-establishing connections.

Figure 25 shows a typical transparent redundancy setup.

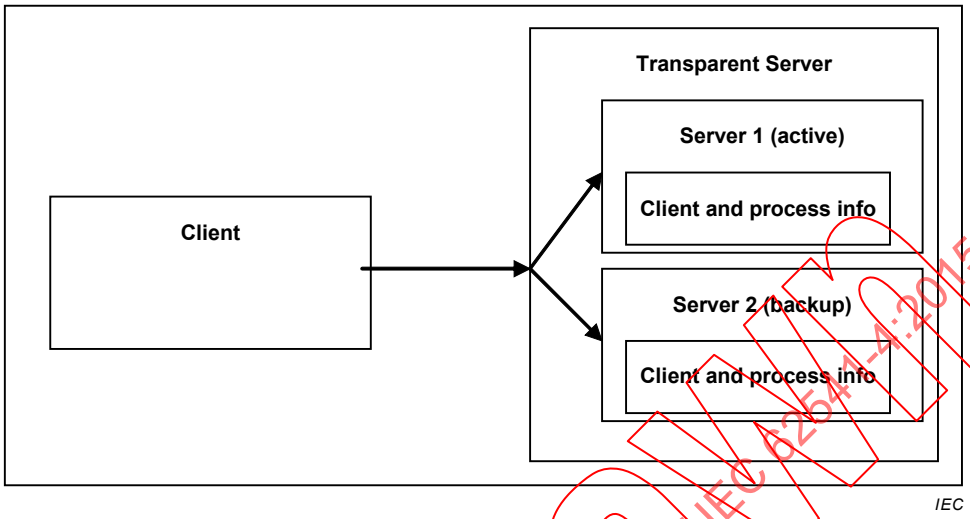


Figure 25 – Transparent Redundancy setup

6.4.2.3 Non-transparent redundancy

For non-transparent redundancy, OPC UA provides the data structures to allow the *Client* to identify what *Servers* are available in the redundant set 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. This information is specified in *NonTransparentRedundancyType ObjectType* defined in IEC 62541-5.

Figure 26 shows a typical non-transparent redundancy setup.

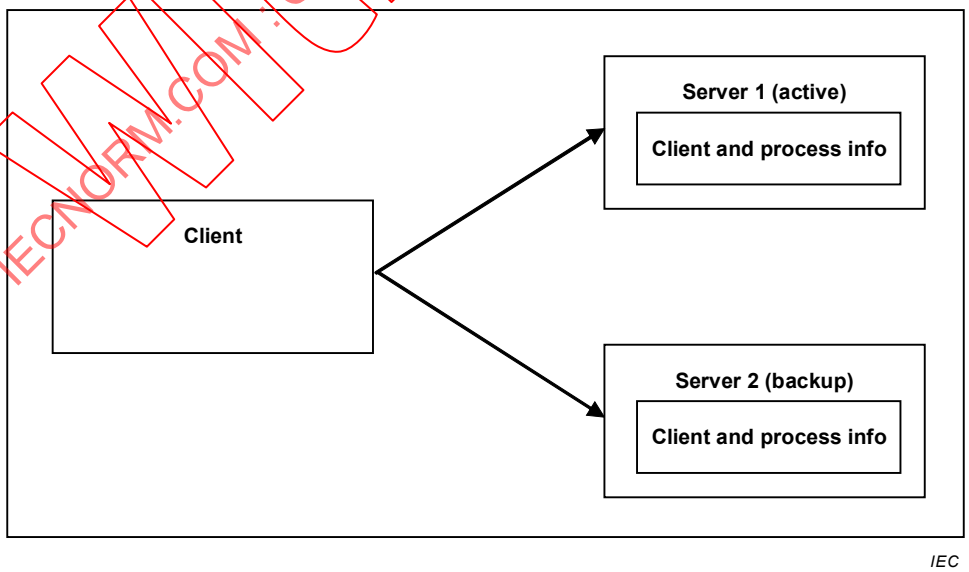


Figure 26 – Non-Transparent Redundancy setup

For non-transparent redundancy the *Server* has additional concepts of *Cold*, *Warm*, *Hot* and *HotAndMirrored* failover modes. The failover mode provides information about the failover

capabilities the *Server* supports and allows a *Client* to determine the available failover actions described in Table 102.

Cold failovers are for *Servers* where only one *Server* can be active at a time. The Client can only connect to one of the Servers in the redundant set.

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). The *ServiceLevel Variable* defined in IEC 62541-5 indicates the ability of the *Server* to provide its data to the *Client*. The *Client* shall use the *Server* with the highest service level. The designation of the active *Server* is vendor specific.

Hot failovers are for *Servers* where more than one *Server* can be active and fully operational. The *ServiceLevel Variable* defined in IEC 62541-5 should be used by the *Client* to find the *Servers* with the highest service level to achieve load balancing.

HotAndMirrored failovers are for *Servers* that are mirroring their internal states to all *Servers* in the redundant set and more than one *Server* can be active and fully operational. Mirroring state minimally includes *Sessions*, *Subscriptions*, registered *Nodes*, sequence numbers and sent *Notifications*. This allows *Clients* to fail over without creating a new context for communication. The *ServiceLevel Variable* defined in IEC 62541-5 should be used by the *Client* to find the *Servers* with the highest service level to achieve load balancing. This failover mode is similar to the transparent redundancy. The advantage is that the *Client* has full control over selecting the *Server*. The disadvantage is that the *Client* needs to be able to handle failovers.

Each *Server* maintains a list of *ServerUris* for all redundant Servers in the redundant set. The list is provided together with the failover mode in the *ServerRedundancy Object* defined in IEC 62541-5. To enable clients to connect to all *Servers* in the list, each *Server* in the list shall provide the *ApplicationDescription* for all *Servers* in the redundant set through the *FindServers Service*. This information is needed by the *Client* to translate the *ServerUri* into information needed to connect to the other *Servers* in the redundant set. Therefore a *Client* needs to know only one of the redundant *Servers* to find the other *Servers* based on the provided information.

Table 102 defines the list of failover actions.

Table 102 – Redundancy failover actions

Failover mode and *Client* options	Cold	Warm	Hot (1)	Hot (2)	HotAndMirrored
On initial connection in addition to actions on active server:					
Connect to more than one OPC UA *Server*.		X	X	X	Optional for status check
Creating *Subscriptions* and adding monitored items.		X	X	X	
Activating sampling on the *Subscriptions*.			X	X	
Activate publishing.				X	
At Failover:					
CreateSecureChannel to backup OPC UA *Server*	X				X
CreateSession on backup OPC UA *Server*	X				
ActivateSession on backup OPC UA *Server*	X				X
Creating *Subscriptions* and adding monitored items.	X				
Activating sampling on the *Subscriptions*.	X	X			
Activate publishing.	X	X	X		

A *Client* connected to a *Cold* failover redundant set can only connect to one *Server*. For failover the *Client* connects to a backup *Server*, creates *Subscriptions* and *MonitoredItems* and activates publishing. There is a loss of data from the time the connection to the active *Server* is interrupted until the time the *Client* gets *Publish Responses* from the backup *Server*.

A *Client* connected to a *Warm* failover redundant set can connect and create *Subscriptions* and *MonitoredItems* on more than one *Server*. Sampling and publishing can only be activated

on one *Server*. This one *Server* can be found by reading the *ServiceLevel Variable* from all *Servers*. The *Server* with the highest *ServiceLevel* is used. For failover the *Client* activates sampling and publishing on the Server with the highest *ServiceLevel*. There is a loss of data from the time the connection to the active *Server* is interrupted until the time the *Client* gets *Publish Responses* from the backup *Server*.

A *Client* connected to a *Hot* failover redundant set has two options to ensure that there is no loss of data in a failover scenario.

- *Hot* (1)
  The *Client* creates *MonitoredItems* and is sampling on more than one *Server* but activates publishing only on the Server with the highest *ServiceLevel*. The Client should setup the queue size for the *MonitoredItems* such that it can buffer all changes during the failover time. The failover time is the time between the connection interruption and the time the *Client* gets *Publish Responses* from the backup *Server*.
- *Hot* (2)
  The *Client* receives the same information from more than one Server by sampling and publishing on more than one *Server*. This mode ensures that there is no interruption in the data stream and no data loss but the *Client* should ensure that it detects duplicate data and it shall process two data streams.

A *Client* connected to a *HotAndMirrored* failover redundant set can assume that the *Servers* are mirroring their state, including connection information and queues. Therefore the *Client* just connects to one Server and in case of a failover re-establishes its connection using a different *Server*. In order to validate the capability to connect to other redundant *Servers* it is allowed to create *Sessions* with other *Servers* and maintain the open connections by periodically reading the *ServiceLevel*. A *Client* shall not create *Subscriptions* on the backup *Servers* for status monitoring. Failover works as described in 6.5. The only difference is that the new *SecureChannel* is created on another *Server* with the highest *ServiceLevel*. *Sessions* and *Subscriptions* can be reused since they are mirrored to all *Servers*.

A *Client* can always use a lesser mode than the server supports. For example, the *Server* supports *Hot* failover mode and the *Client* can use the *Warm* actions. In the case of failover mode *HotAndMirrored*, the *Client* shall not use a lesser mode as this would generate unnecessary load on the *Servers*.

6.4.2.4 Hiding Failover with a Server Proxy (Informative)

A vendor can use the non-transparent redundancy features to create a *Server* proxy running on the *Client* machine that makes redundancy transparent to *Clients*. This reduces the amount of functionality that shall be designed into the *Client* and to enable simpler *Clients* to take advantage of non-transparent redundancy. The *Server* proxy simply duplicates *Subscriptions* and modifications to *Subscriptions*, by passing the calls on to both *Servers*, but only enabling publishing and 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 proxy shall run on the *Client* machine to avoid another single point of failure.

Figure 27 shows the *Server* proxy used to provide redundancy to non redundancy aware *Clients*.

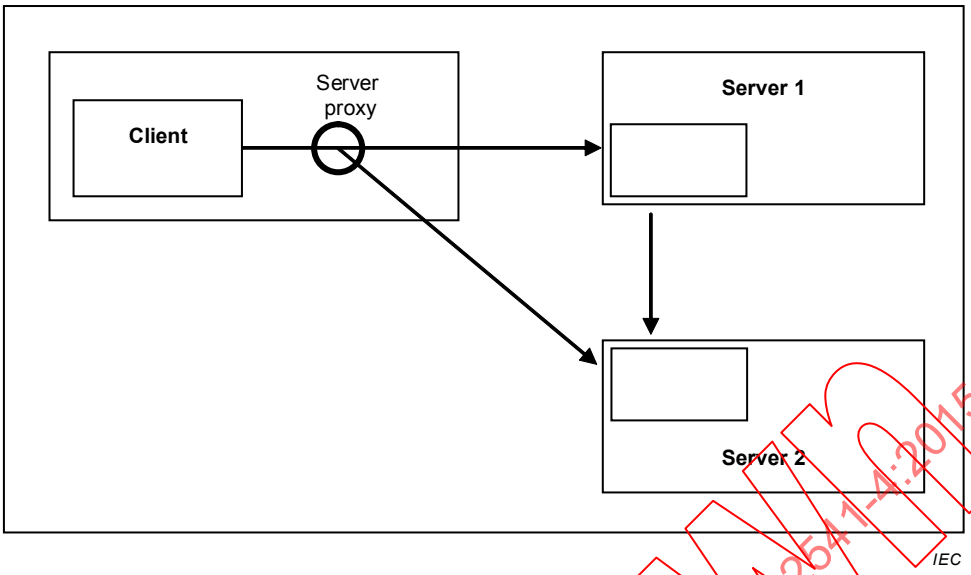


Figure 27 – Server proxy for redundancy

6.4.3 Client redundancy

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

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

6.4.4 Network redundancy

Redundant networks can be used with OPC UA either transparent or non-transparent.

In the transparent case a single *Server Endpoint* can be reached through different network paths. This case is completely handled by the network infrastructure. The selected network path and failover are transparent to the *Client*.

In the non-transparent case the *Server* provides different *Endpoints* for the different network paths. This requires both the Server and the Client to support multiple network connections. In this case the *Client* is responsible for selecting the *Endpoint* and for failover. For failover the normal reconnect scenario described in 6.5 can be used. Only the *SecureChannel* is created with another *Endpoint*. *Sessions* and *Subscriptions* can be reused.

The information about the different network paths is specified in *NonTransparentRedundancyType ObjectType* defined in IEC 62541-5.

Network redundancy can be combined with server redundancy.

6.5 Re-establishing connections

After a *Client* establishes a connection to a *Server* and creates a *Subscription*, the *Client* monitors the connection status. Figure 28 shows the steps to connect a *Client* to a *Server* and the general logic for reconnect handling. Not all possible error scenarios are covered.

The preferred mechanism for a *Client* to monitor the connection status is through the keep-alive of the *Subscription*. A *Client* should subscribe for the *State Variable* in the *ServerStatus* to detect shutdown or other failure states. If no *Subscription* is created or the *Server* does not support *Subscriptions*, the connection can be monitored by periodically reading the *State Variable*.

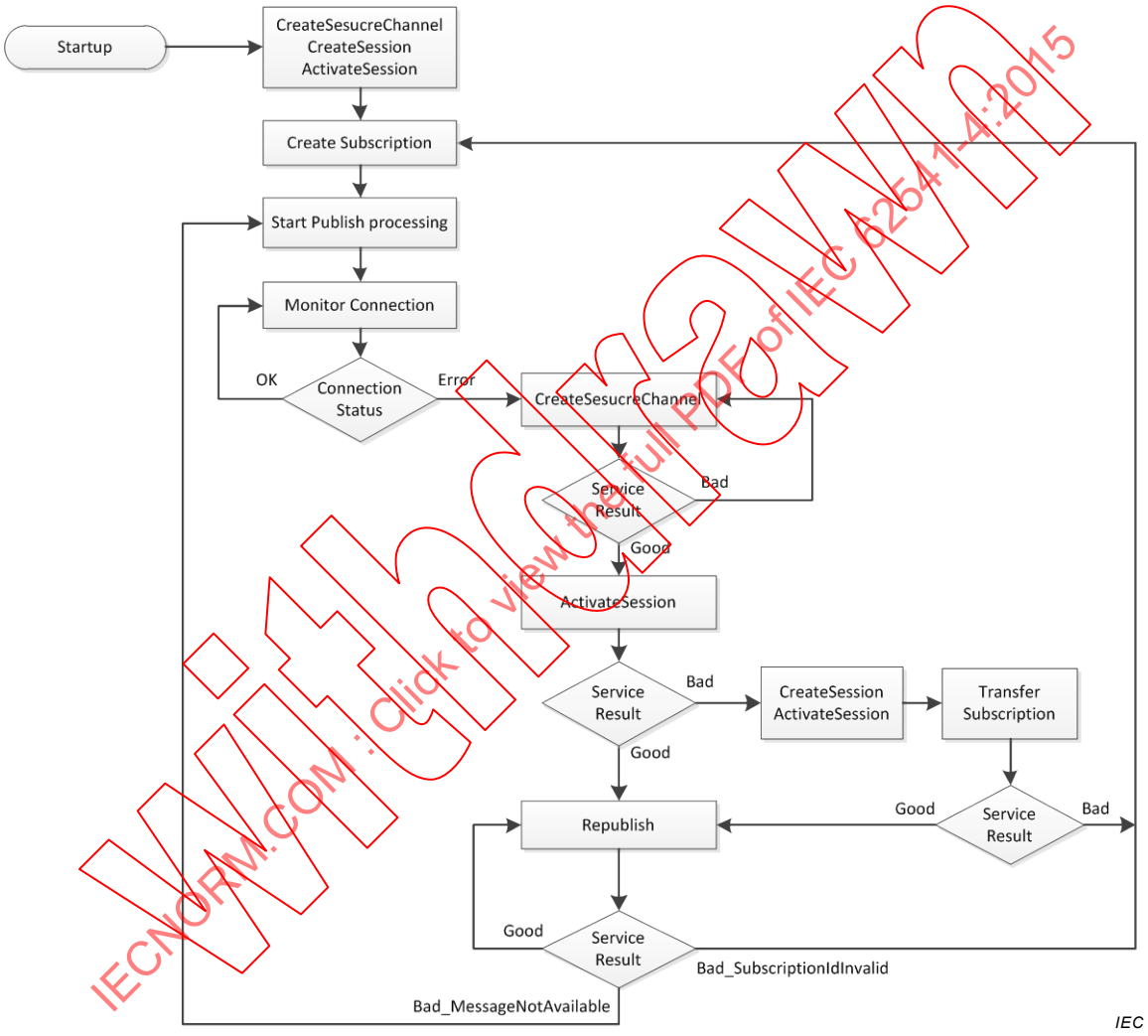


Figure 28 – Reconnect Sequence

When a *Client* loses the connection to the *Server*, the goal is to reconnect without losing information. To do this the *Client* shall re-establish the connection by creating a new *SecureChannel* and activating the *Session* with the *Service ActivateSession*. This assigns the new *SecureChannel* to the existing *Session* and allows the *Client* to reuse the *Session* and *Subscriptions* in the *Server*. This will result in the Client receiving data and event *Notifications* without losing information provided the queues in the MonitoredItems do not overflow.

The *Client* shall only create a new *Session* if *ActivateSession* fails. *TransferSubscriptions* is used to transfer the *Subscription* to the new *Session*. If *TransferSubscriptions* fails, the *Client* needs to create a new *Subscription*.

When the connection is lost, *Publish* responses may have been sent but not received by the *Client*.

After re-establishing the connection the *Client* shall call *Republish* in a loop, starting with the next expected sequence number and incrementing the sequence number until the *Server* returns the status Bad_MessageNotAvailable. After receiving this status, the *Client* shall start sending *Publish* requests with the normal *Publish* handling. This sequence ensures that the lost *NotificationMessages* queued in the *Server* are not overwritten by new *Publish* responses.

If the *Client* detects missing sequence numbers in the *Publish* and is not able to get the lost *NotificationMessages* through *Republish*, the *Client* should read the values of all data *MonitoredItems* to make sure the *Client* has the latest values for all *MonitoredItems*.

Independent of the detailed recovery strategy, the *Client* should make sure that it does not overwrite newer data in the *Client* with older values provided through *Republish*.

If the *Republish* returns Bad_SubscriptionIdInvalid, then the *Client* needs to create a new *Subscription*.

7 Common parameter type definitions

7.1 ApplicationDescription

The components of this parameter are defined in Table 103.

Table 103 – ApplicationDescription

Name	Type	Description
ApplicationDescription	structure	Specifies an application that is available.
applicationUri	String	The globally unique identifier for the application instance. This URI is used as *ServerUri* in *Services* if the application is a *Server*.
productUri	String	The globally unique identifier for the product.
applicationName	LocalizedText	A localized descriptive name for the application.
applicationType	Enum ApplicationType	The type of application. This value is an enumeration with one of the following values: SERVER_0 The application is a *Server*. CLIENT_1 The application is a *Client*. CLIENTANDSERVER_2 The application is a *Client* and a *Server*. DISCOVERYSERVER_3 The application is a *DiscoveryServer*.
gatewayServerUri	String	A URI that identifies the *Gateway Server* associated with the *discoveryUrls*. This value is not specified if the *Server* can be accessed directly. This field is not used if the *applicationType* is CLIENT_1.
discoveryProfileUri	String	A URI that identifies the discovery profile supported by the URLs provided. This field is not used if the *applicationType* is CLIENT_1. If this value is not specified then the Endpoints shall support the Discovery Services defined in 5.4. Alternate discovery profiles are defined in IEC 62541-7.
discoveryUrls []	String	A list of URLs for the discovery *Endpoints* provided by the application. If the *applicationType* is CLIENT_1, this field shall contain an empty list.

7.2 ApplicationInstanceCertificate

An *ApplicationInstanceCertificate* is a *ByteString* containing an encoded *Certificate*. The encoding of an *ApplicationInstanceCertificate* depends on the security technology mapping and is defined completely in IEC 62541-6. Table 104 specifies the information that shall be contained in an *ApplicationInstanceCertificate*.

Table 104 – ApplicationInstanceCertificate

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

7.3 BrowseResult

The components of this parameter are defined in Table 105.

Table 105 – BrowseResult

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

7.4 ContentFilter

7.4.1 ContentFilter structure

The ContentFilter structure defines a collection of elements that define filtering criteria. Each element in the collection describes an operator and an array of operands to be used by the operator. The operators that can be used in a ContentFilter are described in Table 110. The filter is evaluated by evaluating the first entry in the element array starting with the first operand in the operand array. The operands of an element may contain References to sub-elements resulting in the evaluation continuing to the referenced elements in the element array.The evaluation shall not introduce loops. For example evaluation starting from element "A" shall never be able to return to element "A". However there may be more than one path leading to another element "B".If an element cannot be traced back to the starting element it is ignored. Extra operands for any operator shall result in an error. Annex B provides examples using the ContentFilter structure.

Table 106 defines the *ContentFilter* structure.

Table 106 – ContentFilter Structure

Name	Type	Description
ContentFilter	structure	
elements []	ContentFilterElement	List of operators and their operands that compose the filter criteria. The filter is evaluated by starting with the first entry in this array. This structure is defined in-line with the following indented items.
filterOperator	Enum FilterOperator	Filter operator to be evaluated. The *FilterOperator* enumeration is defined in Table 110.
filterOperands []	Extensible Parameter FilterOperand	Operands used by the selected operator. The number and use depend on the operators defined in Table 110. This array needs at least one entry. This extensible parameter type is the *FilterOperand* parameter type specified in 7.4.4. It specifies the list of valid *FilterOperand* values.

7.4.2 ContentFilterResult

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

Table 107 – ContentFilterResult Structure

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

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

Table 108 – ContentFilterResult Result Codes

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

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

Table 109 – ContentFilterResult Operand Result Codes

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

7.4.3 FilterOperator

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

Table 110 – Basic FilterOperator Definition

Operator	Number of Operands	Description
Equals_0	2	TRUE if operand[0] is equal to operand[1]. If the operands are of different types, the system shall perform any implicit conversion to a common type. This operator resolves to FALSE if no implicit conversion is available and the operands are of different types. This operator returns FALSE if the implicit conversion fails. See the discussion on data type precedence in Table 114 for more information how to convert operands of different types.
IsNull_1	1	TRUE if operand[0] is a null value.
GreaterThan_2	2	TRUE if operand[0] is greater than operand[1]. The following restrictions apply to the operands: [0]: Any operand that resolves to an ordered value. [1]: Any operand that resolves to an ordered value. The same conversion rules as defined for *Equals* apply.
LessThan_3	2	TRUE if operand[0] is less than operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
GreaterThanOrEqual_4	2	TRUE if operand[0] is greater than or equal to operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
LessThanOrEqual_5	2	TRUE if operand[0] is less than or equal to operand[1]. The same conversion rules and restrictions as defined for *GreaterThan* apply.
Like_6	2	TRUE if operand[0] matches a pattern defined by operand[1]. See Table 112 for the definition of the pattern syntax. The following restrictions apply to the operands: [0]: Any operand that resolves to a String. [1]: Any operand that resolves to a String. This operator resolves to FALSE if no operand can be resolved to a string.
Not_7	1	TRUE if operand[0] is FALSE. The following restrictions apply to the operands: [0]: Any operand that resolves to a Boolean. If the operand cannot be resolved to a Boolean, the result is a NULL. See below for a discussion on the handling of NULL.
Between_8	3	TRUE if operand[0] is greater or equal to operand[1] and less than or equal to operand[2]. The following restrictions apply to the operands: [0]: Any operand that resolves to an ordered value. [1]: Any operand that resolves to an ordered value. [2]: Any operand that resolves to an ordered value. If the operands are of different types, the system shall perform any implicit conversion to match all operands to a common type. If no implicit conversion is available and the operands are of different types, the particular result is FALSE. See the discussion on data type precedence in Table 114 for more information how to convert operands of different types.
InList_9	2..n	TRUE if operand[0] is equal to one or more of the remaining operands. The Equals Operator is evaluated for operand[0] and each remaining operand in the list. If any Equals evaluation is TRUE, InList returns TRUE.
And_10	2	TRUE if operand[0] and operand[1] are TRUE. The following restrictions apply to the operands: [0]: Any operand that resolves to a Boolean.

Operator	Number of Operands	Description
		[1]: Any operand that resolves to a Boolean. If any operand cannot be resolved to a Boolean it is considered a NULL. See below for a discussion on the handling of NULL.
Or_11	2	TRUE if operand[0] or operand[1] are TRUE. The following restrictions apply to the operands: [0]: Any operand that resolves to a Boolean. [1]: Any operand that resolves to a Boolean. If any operand cannot be resolved to a Boolean it is considered a NULL. See below for a discussion on the handling of NULL.
Cast_12	2	Converts operand[0] to a value with a data type with a NodeId identified by operand[1]. The following restrictions apply to the operands: [0]: Any operand. [1]: Any operand that resolves to a NodeId or ExpandedNodeId where the *Node* is of the *NodeClass DataType*. If there is any error in conversion or in any of the parameters then the Cast Operation evaluates to a NULL. See below for a discussion on the handling of NULL.
BitwiseAnd_16	2	The result is an integer which matches the size of the largest operand and contains a bitwise And operation of the two operands where both have been converted to the same size (largest of the two operands). The following restrictions apply to the operands: [0]: Any operand that resolves to a integer. [1]: Any operand that resolves to a integer. If any operand cannot be resolved to a integer it is considered a NULL. See below for a discussion on the handling of NULL.
BitwiseOr_17	2	The result is an integer which matches the size of the largest operand and contains a bitwise Or operation of the two operands where both have been converted to the same size (largest of the two operands). The following restrictions apply to the operands: [0]: Any operand that resolves to a Integer. [1]: Any operand that resolves to a Integer. If any operand cannot be resolved to a Integer it is considered a NULL. See below for a discussion on the handling of NULL.

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

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

Table 111 – Complex FilterOperator Definition

Operator	Number of Operands	Description
InView_13	1	TRUE if the target *Node* is contained in the *View* defined by operand[0]. The following restrictions apply to the operands: [0]: Any operand that resolves to a *NodeId* that identifies a View Node. If operand[0] does not resolve to a *NodeId* that identifies a View Node, this operation shall always be False.
OfType_14	1	TRUE if the target *Node* is of type operand[0] or of a subtype of operand[0]. The following restrictions apply to the operands: [0]: Any operand that resolves to a *NodeId* that identifies an ObjectType or VariableType Node. If operand[0] does not resolve to a *NodeId* that identifies an ObjectType or VariableType Node, this operation shall always be False.
RelatedTo_15	6	TRUE if the target *Node* is of type Operand[0] and is related to a *NodeId* of the type defined in Operand[1] by the *Reference* type defined in Operand[2]. Operand[0] or Operand[1] can also point to an element *Reference* where the referred to element is another RelatedTo operator. This allows chaining of relationships (e.g. A is related to B is related to C), where the relationship is defined by the *ReferenceType* defined in Operand[2]. In this case, the referred to element returns a list of *NodeIds* instead of TRUE or FALSE. In this case if any errors occur or any of the operands cannot be resolved to an appropriate value, the result of the chained relationship is an empty list of nodes. Operand[3] defines the number of hops for which the relationship should be followed. If Operand[3] is 1, then objects shall be directly related. If a hop is greater than 1, then a *NodeId* of the type described in Operand[1] is checked for at the depth specified by the hop. In this case, the type of the intermediate *Node* is undefined, and only the *Reference* type used to reach the end *Node* is defined. If the requested number of hops cannot be followed, then the result is FALSE, i.e., an empty *Node* list. If Operand[3] is 0, the relationship is followed to its logical end in a forward direction and each *Node* is checked to be of the type specified in Operand[1]. If any *Node* satisfies this criterion, then the result is TRUE, i.e., the *NodeId* is included in the sub-list. Operand [4] defines if Operands [0] and [1] should include support for subtypes of the types defined by these operands. A TRUE indicates support for subtypes Operand [5] defines if Operand [2] should include support for subtypes of the reference type. A TRUE indicates support for subtypes. The following restrictions apply to the operands: [0]: Any operand that resolves to a *NodeId* or *ExpandedNodeId* that identifies an ObjectType or VariableType Node or a reference to another element which is a RelatedTo operator. [1]: Any operand that resolves to a *NodeId* or *ExpandedNodeId* that identifies an ObjectType or VariableType Node or a reference to another element which is a RelatedTo operator. [2]: Any operand that resolves to a *NodeId* that identifies a ReferenceType Node. [3]: Any operand that resolves to a value implicitly convertible to Uint32. [4]: Any operand that resolves to a value implicitly convertible to a boolean; if this operand does not resolve to a Boolean, then a value of FALSE is used. [5]: Any operand that resolves to a value implicitly convertible to a boolean; if this operand does not resolve to a Boolean, then a value of FALSE is used. If none of the operands [0],[1],[2],[3] resolves to an appropriate value then the result of this operation shall always be False (or an Empty set in the case of a nested *RelatedTo* operand). See examples for RelatedTo in B.2.

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

The *Like* operator can be used to perform wildcard comparisons. Several special characters can be included in the second operand of the *Like* operator. The valid characters are defined in Table 112. The wildcard characters can be combined in a single string (i.e. 'Th[ia][ts]%' would match 'That is fine', 'This is fine', 'That as one', 'This it is', 'Then at any', etc.). The *Like* operator is case sensitive.

Table 112 – Wildcard characters

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

Table 113 defines the conversion rules for the operand values. The types are automatically converted if an implicit conversion exists (I). If an explicit conversion exists (E) then type can be converted with the cast operator. If no conversion exists (X) the then types cannot be converted, however, some servers may support application specific explicit conversions. The types used in the table are defined in IEC 62541-3. A data type that is not in the table does not have any defined conversions.

Table 113 – Conversion Rules

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

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

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

Guid, *NodeId* and *ExpandedNodeId* are converted to and from *String* using the syntax defined in IEC 62541-6.

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

Converting a negative value to an unsigned type causes a conversion error. If the conversion fails the result is a null value.

Converting a value that is outside the range of the target type causes a conversion error. If the conversion fails the result is a null value.

ByteString is converted to *String* by formatting the bytes as a sequence of hexadecimal digits.

LocalizedText values are converted to *Strings* by dropping the *Locale*. *Strings* are converted to *LocalizedText* values by setting the Locale to "".

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

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

Boolean values are converted to '1' when true and '0' when false. Non zero numeric values are converted to true *Boolean* values. Numeric values of 0 are converted to false *Boolean* values. *String* values containing "true", "false", "1" or "0" can be converted to *Boolean* values. Other string values cause a conversion error. In this case *Strings* are case-insensitive.

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

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

Table 114 – Data Precedence Rules

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

Operands may contain null values (i.e. values which do not exist). When this happens, the element always evaluates to NULL (unless the IsNull_1 operator has been specified). Table 115 defines how to combine elements that evaluate to NULL with other elements in a logical AND operation.

Table 115 – Logical AND Truth Table

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

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

Table 116 – Logical OR Truth Table

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

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

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

7.4.4 FilterOperand parameters

7.4.4.1 Overview

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

Table 117 – FilterOperand parameter TypeIds

Symbolic Id	**Description**
Element	Specifies an index into the array of elements. This type is used to build a logic tree of sub-elements by linking the operand of one element to a sub-element.
Literal	Specifies a literal value.
Attribute	Specifies any *Attribute* of an *Object* or *Variable Node* using a *Node* in the type system and relative path constructed from *ReferenceTypes* and *BrowseNames*.
SimpleAttribute	Specifies any *Attribute* of an *Object* or *Variable Node* using a *TypeDefinition* and a relative path constructed from *BrowseNames*.

7.4.4.2 ElementOperand

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

Table 118 defines the *ElementOperand* type.

Table 118 – ElementOperand

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

7.4.4.3 LiteralOperand

Table 119 defines the *LiteralOperand* type.

Table 119 – LiteralOperand

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

7.4.4.4 AttributeOperand

Table 120 defines the *AttributeOperand* type.

Table 120 – AttributeOperand

Name	Type	Description
AttributeOperand	structure	Attribute of a *Node* in the address space.
nodeId	NodeId	*NodeId* of a *Node* from the type system.
alias	String	An optional parameter used to identify or refer to an alias. An alias is a symbolic name that can be used to alias this operand and use it in other locations in the filter structure.
browsePath	RelativePath	Browse path relative to the *Node* identified by the *nodeId* parameter. See 7.25 for the definition of *RelativePath*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *AttributeId*. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in IEC 62541-6.
indexRange	NumericRange	This parameter is used to identify a single element of an array or a single range of indexes for an array. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is not used if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array and this parameter is not used, then all elements are to be included in the range. The parameter is null if not used.

7.4.4.5 SimpleAttributeOperand

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

Table 121 defines the *SimpleAttributeOperand* type.

Table 121 – SimpleAttributeOperand

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

7.5 Counter

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

$x + y = (x + y) \text{mod}(\text{max value} + 1)$

For example:

max value + 1 = 0

max value + 2 = 1

7.6 ContinuationPoint

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

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

Servers shall support at least one *ContinuationPoint* per *Session*. *Servers* specify a maximum number of *ContinuationPoints* per *Session* in the *ServerCapabilities Object* defined in IEC 62541-5. *ContinuationPoints* remain active until the *Client* retrieves the remaining results, the *Client* releases the *ContinuationPoint* or the *Session* is closed. A *Server* shall automatically free *ContinuationPoints* from prior requests if they are needed to process a new request. The *Server* returns a *Bad_ContinuationPointInvalid* error if a *Client* tries to use a *ContinuationPoint* that has been released.

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

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

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

7.7 DataValue

7.7.1 General

The components of this parameter are defined in Table 122.

Table 122 – DataValue

Name	Type	Description
DataValue	structure	The value and associated information.
value	BaseDataType	The data value. If the *StatusCode* indicates an error then the value is to be ignored and the Server shall set it to null.
statusCode	StatusCode	The *StatusCode* that defines with the *Server*'s ability to access/provide the value. The *StatusCode* type is defined in 7.33
sourceTimestamp	UtcTime	The source timestamp for the value.
sourcePicoSeconds	UInteger	Specifies the number of 10 picoseconds (1,0 e-11 seconds) intervals which shall be added to the sourceTimestamp.
serverTimestamp	UtcTime	The *Server* timestamp for the value.
serverPicoSeconds	UInteger	Specifies the number of 10 picoseconds (1,0 e-11 seconds) intervals which shall be added to the serverTimestamp.

7.7.2 PicoSeconds

Some applications require high resolution timestamps. The *PicoSeconds* fields allow applications to specify timestamps with a resolution of 10 picoseconds. The actual size of the *PicoSeconds* field depends on the resolution of the *UtcTime DataType*. For example, if the *UtcTime DataType* has a resolution of 100 nanoseconds then the *PicoSeconds* field would have to store values up to 10 000 in order to provide the resolution of 10 picoseconds. The resolution of the *UtcTime DataType* depends on the *Mappings* defined in IEC 62541-6.

7.7.3 SourceTimestamp

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

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

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

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

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

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

7.7.4 ServerTimestamp

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

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

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

If the *Server* subscribes 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 seconds,

b) the connection to the device is good,

c) the device sent an update 3 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.

7.7.5 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 7.33.

Overall condition (severity)

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

Rules

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

7.8 DiagnosticInfo

The components of this parameter are defined in Table 123.

Table 123 – DiagnosticInfo

Name	Type	Description
DiagnosticInfo	structure	Vendor-specific diagnostic information.
namespaceUri	Int32	The *symbolicId* is defined within the context of a namespace. This namespace is represented as a string and is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *namespaceIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified. The *namespaceUri* shall not be the standard OPC UA namespace. There are no *symbolicIds* provided for standard *StatusCodes*.
symbolicId	Int32	The *symbolicId* shall be used to identify a vendor-specific error or condition; typically the result of some server internal operation. 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 use this string as *symbolicId* (e.g., "0xC0040007" or "-4"). This symbolic identifier string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *symbolicId* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified. The *symbolicId* shall not contain *StatusCodes*. If the *localizedText* contains a translation for the description of a *StatusCode*, the *symbolicId* is -1.
locale	Int32	The locale part of the vendor-specific localized text describing the symbolic id. This localized text string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *localizedTextIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified.
localizedText	Int32	A vendor-specific localized text string describes the symbolic id. The maximum length of this text string is 256 characters. This localized text string is conveyed to the *Client* in the *stringTable* parameter of the *ResponseHeader* parameter defined in 7.27. The *localizedTextIndex* parameter contains the index into the *stringTable* for this string. -1 indicates that no string is specified. The localizedText refers to the symbolicId if present or the string that describes the standard StatusCode if the server provides translations. If the index is -1, the server has no translation to return and the client should use the invariant StatusCode description from the specification.
additionalInfo	String	Vendor-specific diagnostic information.
innerStatusCode	StatusCode	The *StatusCode* from the inner operation. Many applications will make calls into underlying systems during OPC UA request processing. An OPC UA *Server* has the option of reporting the status from the underlying system in the diagnostic info.
innerDiagnosticInfo	DiagnosticInfo	The diagnostic info associated with the inner *StatusCode*.

7.9 EndpointDescription

The components of this parameter are defined in Table 124.

Table 124 – EndpointDescription

Name	Type	Description
EndpointDescription	structure	Describes an *Endpoint* for a *Server*.
endpointUrl	String	The URL for the *Endpoint* described.
Server	ApplicationDescription	The description for the *Server* that the *Endpoint* belongs to. The *ApplicationDescription* type is defined in 7.1.
serverCertificate	ApplicationInstance Certificate	The *Application Instance Certificate* issued to the *Server*. The *ApplicationInstanceCertificate* type is defined in 7.2.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type *MessageSecurityMode* type is defined in 7.14. A *SecureChannel* may have to be created even if the *securityMode* is NONE. The exact behaviour depends on the mapping used and is described in the IEC 62541-6.
securityPolicyUri	String	The URI for *SecurityPolicy* to use when securing messages. The set of known URIs and the *SecurityPolicies* associated with them are defined in IEC 62541-7.
userIdentityTokens []	UserTokenPolicy	The user identity tokens that the *Server* will accept. The *Client* shall pass one of the *UserIdentityTokens* in the *ActivateSession* request. The *UserTokenPolicy* type is described in 7.36.
transportProfileUri	String	The URI of the *Transport Profile* supported by the *Endpoint*. IEC 62541-7 defines URIs for the *Transport Profiles*.
securityLevel	Byte	A numeric value that indicates how secure the EndpointDescription is compared to other EndpointDescriptions for the same Server. A value of 0 indicates that the *EndpointDescription* is not recommended and is only supported for backward compatibility. A higher value indicates better security.

7.10 ExpandedNodeId

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

Table 125 – ExpandedNodeId

Name	Type	Description
ExpandedNodeId	structure	The *NodeId* with the namespace expanded to its string representation.
serverIndex	Index	Index that identifies the *Server* that contains the *TargetNode*. This *Server* may be the local *Server* or a remote *Server*. This index is the index of that *Server* in the local *Server*'s *Server* table. The index of the local *Server* in the *Server* table is always 0. All remote *Servers* have indexes greater than 0. The *Server* table is contained in the *Server Object* in the *AddressSpace* (see IEC 62541-3 and IEC 62541-5). The *Client* may read the *Server* table *Variable* to access the description of the target *Server*.
namespaceUri	String	The URI of the namespace. If this parameter is specified then the namespace index is ignored. Subclause 5.4 and IEC 62541-12[3] describe the discovery mechanism that can be used to resolve URIs into URLs.
namespaceIndex	Index	The index in the *Server*'s namespace table. This parameter shall be 0 and is ignored in the *Server* if the namespace URI is specified.
identifierType	IdType	Type of the identifier element of the *NodeId*.
identifier	*	The identifier for a *Node* in the address space of an OPC UA *Server* (see *NodeId* definition in IEC 62541-3).

7.11 ExtensibleParameter

The extensible parameter types can only be extended by additional parts of series of standards.

3 Under consideration.

The *ExtensibleParameter* defines a data structure with two elements. The *parameterTypeId* specifies the data type encoding of the second element. Therefore the second element is specified as "--". The *ExtensibleParameter* base type is defined in Table 126.

Concrete extensible parameters that are common to OPC UA are defined in Clause 7. Additional parts of series of standards can define additional extensible parameter types.

Table 126 – ExtensibleParameter Base Type

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

7.12 Index

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

7.13 IntegerId

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

7.14 MessageSecurityMode

The *MessageSecurityMode* is an enumeration that specifies what security should be applied to messages exchanges during a Session. The possible values are described in Table 127.

Table 127 – MessageSecurityMode Values

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

7.15 MonitoringParameters

The components of this parameter are defined in Table 128.

Table 128 – MonitoringParameters

Name	Type	Description
MonitoringParameters	structure	Parameters that define the monitoring characteristics of a *MonitoredItem*.
clientHandle	IntegerId	*Client*-supplied id of the *MonitoredItem*. This id is used in *Notifications* generated for the list *Node*. The *IntegerId* type is defined in 7.13.
samplingInterval	Duration	The interval that defines the fastest rate at which the *MonitoredItem*(s) should be accessed and evaluated. This interval is defined in milliseconds. The value 0 indicates that the *Server* should use the fastest practical rate. The value -1 indicates that the default sampling interval defined by the publishing interval of the *Subscription* is requested. A different sampling interval is used if the publishing interval is not a supported sampling interval. Any negative number is interpreted as -1. The sampling interval is not changed if the publishing interval is changed by a subsequent call to the *ModifySubscription Service*. The *Server* uses this parameter to assign the *MonitoredItems* to a sampling interval that it supports. The assigned interval is provided in the *revisedSamplingInterval* parameter. The Server shall always return a *revisedSamplingInterval* that is equal or higher than the requested *samplingInterval*. If the requested *samplingInterval* is higher than the maximum sampling interval supported by the *Server*, the maximum sampling interval is returned.
filter	Extensible Parameter MonitoringFilter	A filter used by the *Server* to determine if the *MonitoredItem* should generate a *Notification*. If not used, this parameter is null. The *MonitoringFilter* parameter type is an extensible parameter type specified in 7.16. It specifies the types of filters that can be used.
queueSize	Counter	The requested size of the *MonitoredItem* queue. The following values have special meaning for data monitored items: Value Meaning 0 or 1 the server returns the default queue size which shall be 1 as *revisedQueueSize* for data monitored items. The queue has a single entry, effectively disabling queuing. For values larger than one a first-in-first-out queue is to be used. The *Server* may limit the size in *revisedQueueSize*. In the case of a queue overflow, the *Overflow* bit (flag) in the *InfoBits* portion of the *DataValue statusCode* is set in the new value. The following values have special meaning for event monitored items: Value Meaning 0 the *Server* returns the default queue size for *Event Notifications* as *revisedQueueSize* for event monitored items. 1 the *Server* returns the minimum queue size the Server requires for *Event Notifications* as *revisedQueueSize*. MaxUInt32 the *Server* returns the maximum queue size that the *Server* can support for *Event Notifications* as *revisedQueueSize*. If a *Client* chooses a value between the minimum and maximum settings of the *Server* the value shall be returned in the *revisedQueueSize*. If the requested *queueSize* is outside the minimum or maximum, the *Server* shall return the corresponding bounding value. In the case of a queue overflow, an *Event* of the type *EventQueueOverflowEventType* is generated.
discardOldest	Boolean	A boolean parameter that specifies the discard policy when the queue is full and a new *Notification* is to be queued. It has the following values: TRUE the oldest (first) *Notification* in the queue is discarded. The new *Notification* is added to the end of the queue. FALSE the last *Notification* added to the queue gets replaced with the new *Notification*.

7.16 MonitoringFilter parameters

7.16.1 Overview

The *CreateMonitoredItem Service* allows specifying a filter for each *MonitoredItem*. The *MonitoringFilter* is an extensible parameter whose structure depends on the type of item being monitored. The *parameterTypeIds* are defined in Table 129. Other types can be defined by additional parts of this multi-part specification or other specifications based on OPC UA. The *ExtensibleParameter* type is defined in 7.11.

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

Table 129 – MonitoringFilter parameterTypeIds

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

7.16.2 DataChangeFilter

The *DataChangeFilter* defines the conditions under which a *DataChange Notification* should be reported and, optionally, a range or band for value changes where no *DataChange Notification* is generated. This range is called *Deadband*. The *DataChangeFilter* is defined in Table 130.

Table 130 – DataChangeFilter

Name	Type	Description
DataChangeFilter	structure	
trigger	Enum DataChangeTrigger	Specifies the conditions under which a data change notification should be reported. It has the following values: STATUS_0 Report a notification ONLY if the *StatusCode* associated with the value changes. See Table 166 for *StatusCodes* defined in this standard. IEC 62541-8 specifies additional *StatusCodes* that are valid in particular for device data. STATUS_VALUE_1 Report a notification if either the *StatusCode* or the value change. The *Deadband* filter can be used in addition for filtering value changes. **This is the default setting if no filter is set.** STATUS_VALUE_TIMESTAMP_2 Report a notification if either StatusCode, value or the *SourceTimestamp* change. If a *Deadband* filter is specified, this trigger has the same behaviour as STATUS_VALUE_1. If the DataChangeFilter is not applied to the monitored item, STATUS_VALUE_1 is the default reporting behaviour.
deadbandType	UInt32	A value that defines the *Deadband* type and behaviour. Value deadbandType None_0 No *Deadband* calculation should be applied. Absolute_1 AbsoluteDeadband (see below) Percent_2 PercentDeadband (This type is specified in IEC 62541-8).
deadbandValue	Double	The *Deadband* is applied only if * the *trigger* includes value changes and * the *deadbandType* is set appropriately. Deadband is ignored if the status of the data item changes. ***DeadbandType = AbsoluteDeadband***: For this type the *deadbandValue* contains the absolute change in a data value that shall cause a *Notification* to be generated. This parameter applies only to *Variable*s with any *Number* data type. An exception that causes a *DataChange Notification* based on an AbsoluteDeadband is determined as follows: **Generate a *Notification* if (absolute value of (last cached value – current value) > *AbsoluteDeadband*)** The last cached value is defined as the last value pushed to the queue. If the item is an array of values, the entire array is returned if any array element exceeds the AbsoluteDeadband, or the size or dimension of the array changes. ***DeadbandType = PercentDeadband:*** This type is specified in IEC 62541-8

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

7.16.3 EventFilter

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

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

Each *Event Notification* shall include the fields defined by the *selectClauses* parameter of the *EventFilter*. The defined *EventTypes* are specified in IEC 62541-5.

The *selectClause* and *whereClause* parameters are specified with the *SimpleAttributeOperand* structure (see 7.4.4.5). This structure requires the *NodeId* of an *EventType* supported by the *Server* and a path to an *InstanceDeclaration*. An *InstanceDeclaration* is a *Node* which can be found by following forward hierarchical references from the fully inherited *EventType* where the *Node* is also the source of a *HasModellingRule* reference. *EventTypes*, *InstanceDeclarations* and *Modelling Rules* are described completely in IEC 62541-3.

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

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

The *SimpleAttributeOperand* structure allows the Client to specify any *Attribute*, however, the *Server* is only required to support the *Value Attribute* for *Variable Nodes* and the *NodeId Attribute* for *Object Nodes*. That said, profiles defined in IEC 62541-7 may make support for additional *Attributes* mandatory.

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

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

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

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

The *Server* shall validate the *whereClauses* when a *Client* creates or updates the *EventFilter*. Any structural errors in the construction of the filter and any errors which are true for all possible *Events* are returned in the *whereClauseResult* parameter described in Table 132. Errors that could occur depending on the state of the *Server* or the *Event* are not reported. Some *Servers*, like aggregating *Servers*, may not know all possible *EventTypes* at the time the *EventFilter* is set. These *Servers* do not return errors for unknown *EventTypes* or *BrowsePaths*.

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

Table 131 defines the EventFilter structure.

Table 131 – EventFilter structure

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

Table 132 defines the EventFilterResult structure. This is the *MonitoringFilterResult* associated with the *EventFilter MonitoringFilter*.

Table 132 – EventFilterResult structure

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

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

Table 133 – EventFilterResult Result Codes

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

7.16.4 AggregateFilter

The *AggregateFilter* defines the *Aggregate* function that should be used to calculate the values to be returned. See IEC 62541-13 for details on possible *Aggregate* functions. It specifies a startTime of the first *Aggregate* to be calculated. The samplingInterval of the MonitoringAttributes (see 7.15) defines how the server should internally sample the underlying data source. The processingInterval specifies the size of a time-period where the *Aggregate* is calculated. The queueSize from the MonitoringAttributes specifies the number of processed values that should be kept.

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

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

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

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

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

The *AggregateFilter* is defined in Table 134.

Table 134 – AggregateFilter structure

Name	Type	Description
AggregateFilter	structure	
startTime	UtcTime	Beginning of period to calculate the *Aggregate* the first time. The size of each period used to calculate the *Aggregate* is defined by the samplingInterval of the MonitoringAttributes (see 7.15).
aggregateType	NodeId	The NodeId of the *AggregateFunctionType* object that indicates the *Aggregate* to be used when retrieving processed data. See IEC 62541-13 for details.
processingInterval	Duration	The period be used to compute the *Aggregate*.
aggregateConfiguration	Aggregate Configuration	This parameter allows *Clients* to override the *Aggregate* configuration settings supplied by the *AggregateConfiguration* object on a per monitored item basis. See IEC 62541-13 for more information on *Aggregate* configurations. If the *Server* does not support the ability to override the *Aggregate* configuration settings it shall return a *StatusCode* of Bad_AggregateListMismatch. This structure is defined in-line with the following indented items.
useServerCapabilities Defaults	Boolean	If value = TRUE use Aggregate configuration settings as outlined by the AggregateConfiguration object. If value=FALSE use configuration settings as outlined in the following aggregateConfiguration parameters. Default is TRUE.
treatUncertainAsBad	Boolean	As described in IEC 62541-13.
percentDataBad	Byte	As described in IEC 62541-13.
percentDataGood	Byte	As described in IEC 62541-13.
useSloped Extrapolation	Boolean	As described in IEC 62541-13.

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

Table 135 – AggregateFilterResult structure

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

7.17 MonitoringMode

The *MonitoringMode* is an enumeration that specifies whether sampling and reporting are enabled or disabled for a *MonitoredItem*. The value of the publishing enabled parameter for a *Subscription* does not affect the value of the monitoring mode for a *MonitoredItem* of the *Subscription*. The values of this parameter are defined in Table 136.

Table 136 – MonitoringMode Values

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

7.18 NodeAttributes parameters

7.18.1 Overview

The *AddNodes Service* allows specifying the *Attributes* for the *Nodes* to add. The *NodeAttributes* is an extensible parameter whose structure depends on the type of the *Attribute* being added. It identifies the *NodeClass* that defines the structure of the *Attributes* that follow. The *parameterTypeIds* are defined in Table 137. The *ExtensibleParameter* type is defined in 7.11.

Table 137 – NodeAttributes parameterTypeIds

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

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

Table 138 – Bit mask for specified Attributes

Field	Bit	Description
AccessLevel	0	Indicates if the AccessLevel Attribute is set.
ArrayDimensions	1	Indicates if the ArrayDimensions Attribute is set.
Reserved	2	Reserved to be consistent with WriteMask defined in IEC 62541-3.
ContainsNoLoops	3	Indicates if the ContainsNoLoops Attribute is set.
DataType	4	Indicates if the DataType Attribute is set.
Description	5	Indicates if the Description Attribute is set.
DisplayName	6	Indicates if the DisplayName Attribute is set.
EventNotifier	7	Indicates if the EventNotifier Attribute is set.
Executable	8	Indicates if the Executable Attribute is set.
Historizing	9	Indicates if the Historizing Attribute is set.
InverseName	10	Indicates if the InverseName Attribute is set.
IsAbstract	11	Indicates if the IsAbstract Attribute is set.
MinimumSamplingInterval	12	Indicates if the MinimumSamplingInterval Attribute is set.
Reserved	13	Reserved to be consistent with WriteMask defined in IEC 62541-3.
Reserved	14	Reserved to be consistent with WriteMask defined in IEC 62541-3.
Symmetric	15	Indicates if the Symmetric Attribute is set.
UserAccessLevel	16	Indicates if the UserAccessLevel Attribute is set.
UserExecutable	17	Indicates if the UserExecutable Attribute is set.
UserWriteMask	18	Indicates if the UserWriteMask Attribute is set.
ValueRank	19	Indicates if the ValueRank Attribute is set.
WriteMask	20	Indicates if the WriteMask Attribute is set.
Value	21	Indicates if the Value Attribute is set.
Reserved	22:32	Reserved for future use. Shall always be zero.

7.18.2 ObjectAttributes parameter

Table 139 defines the *ObjectAttributes* parameter.

Table 139 – ObjectAttributes

Name	Type	Description
ObjectAttributes	structure	Defines the *Attributes* for the *Object NodeClass*.
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
eventNotifier	Byte	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.3 VariableAttributes parameter

Table 140 defines the *VariableAttributes* parameter.

Table 140 – VariableAttributes

Name	Type	Description
VariableAttributes	structure	Defines the *Attributes* for the *Variable NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
value	Defined by the *DataType Attribute*	See IEC 62541-3 for the description of this *Attribute*.
dataType	NodeId	See IEC 62541-3 for the description of this *Attribute*.
valueRank	Int32	See IEC 62541-3 for the description of this *Attribute*.
arrayDimensions	UInt32 []	See IEC 62541-3 for the description of this *Attribute*.
accessLevel	Byte	See IEC 62541-3 for the description of this *Attribute*.
userAccessLevel	Byte	See IEC 62541-3 for the description of this *Attribute*.
minimumSamplingInterval	Duration	See IEC 62541-3 for the description of this *Attribute*.
historizing	Boolean	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.4 MethodAttributes parameter

Table 141 defines the *MethodAttributes* parameter.

Table 141 – MethodAttributes

Name	Type	Description
BaseAttributes	structure	Defines the *Attributes* for the *Method NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
executable	Boolean	See IEC 62541-3 for the description of this *Attribute*.
userExecutable	Boolean	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.5 ObjectTypeAttributes parameter

Table 142 defines the *ObjectTypeAttributes* parameter.

Table 142 – ObjectTypeAttributes

Name	Type	Description
ObjectTypeAttributes	structure	Defines the *Attributes* for the *ObjectType NodeClass*.
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
isAbstract	Boolean	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.6 VariableTypeAttributes parameter

Table 143 defines the *VariableTypeAttributes* parameter.

Table 143 – VariableTypeAttributes

Name	Type	Description
VariableTypeAttributes	structure	Defines the *Attributes* for the *VariableType NodeClass*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
value	Defined by the *DataType Attribute*	See IEC 62541-3 for the description of this *Attribute*.
dataType	NodeId	See IEC 62541-3 for the description of this *Attribute*.
valueRank	Int32	See IEC 62541-3 for the description of this *Attribute*.
arrayDimensions	UInt32 []	See IEC 62541-3 for the description of this *Attribute*.
isAbstract	Boolean	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.7 ReferenceTypeAttributes parameter

Table 144 defines the *ReferenceTypeAttributes* parameter.

Table 144 – ReferenceTypeAttributes

Name	Type	Description
ReferenceTypeAttributes	structure	Defines the *Attributes* for the *ReferenceType NodeClass.*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
isAbstract	Boolean	See IEC 62541-3 for the description of this *Attribute*.
symmetric	Boolean	See IEC 62541-3 for the description of this *Attribute*.
inverseName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.8 DataTypeAttributes parameter

Table 145 defines the *DataTypeAttributes* parameter.

Table 145 – DataTypeAttributes

Name	Type	Description
DataTypeAttributes	structure	Defines the *Attributes* for the *DataType NodeClass.*
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
isAbstract	Boolean	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.18.9 ViewAttributes parameter

Table 146 defines the *ViewAttributes* parameter.

Table 146 – ViewAttributes

Name	Type	Description
ViewAttributes	structure	Defines the *Attributes* for the *View NodeClass*.
specifiedAttributes	UInt32	A bit mask that indicates which fields contain valid values. A field shall be ignored if the corresponding bit is set to 0. The bit values are defined in Table 138.
displayName	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
description	LocalizedText	See IEC 62541-3 for the description of this *Attribute*.
containsNoLoops	Boolean	See IEC 62541-3 for the description of this *Attribute*.
eventNotifier	Byte	See IEC 62541-3 for the description of this *Attribute*.
writeMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.
userWriteMask	UInt32	See IEC 62541-3 for the description of this *Attribute*.

7.19 NotificationData parameters

7.19.1 Overview

The *NotificationMessage* structure used in the *Subscription Service* set allows specifying different types of *NotificationData*. The *NotificationData* parameter is an extensible parameter whose structure depends on the type of *Notification* being sent. This parameter is defined in Table 147. Other types can be defined by additional parts of series of standards or other specifications based on OPC UA. The *ExtensibleParameter* type is defined in 7.11.

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

Table 147 – NotificationData parameterTypeIds

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

7.19.2 DataChangeNotification parameter

Table 148 defines the *NotificationData* parameter used for data change notifications. This structure contains the monitored data items that are to be reported. Monitored data items are reported under two conditions:

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

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

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

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

Table 148 – DataChangeNotification

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

7.19.3 EventNotificationList parameter

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

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

Table 149 – EventNotificationList

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

7.19.4 StatusChangeNotification parameter

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

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

Table 150 – StatusChangeNotification

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

7.20 NotificationMessage

The components of this parameter are defined in Table 151.

Table 151 – NotificationMessage

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

7.21 NumericRange

This parameter is defined in Table 152. A formal BNF definition of the numeric range can be found in Clause A.3.

The syntax for the string contains one of the following two constructs. The first construct is the string representation of an individual integer. For example, "6" is valid, but "6,0" and "3,2" are not. The minimum and maximum values that can be expressed are defined by the use of this parameter and not by this parameter type definition. The second construct is a range represented by two integers separated by the colon (":") character. The first integer shall always have a lower value than the second. For example, "5:7" is valid, while "7:5" and "5:5" are not. The minimum and maximum values that can be expressed by these integers are defined by the use of this parameter, and not by this parameter type definition. No other characters, including white-space characters, are permitted.

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

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

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

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

Bad_IndexRangeInvalid is only used for invalid syntax of the *NumericRange*. All other invalid requests with a valid syntax shall result in *Bad_IndexRangeNoData*.

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

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

Table 152 – NumericRange

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

7.22 QueryDataSet

The components of this parameter are defined in Table 153.

Table 153 – QueryDataSet

Name	Type	Description
QueryDataSet	structure	Data related to a *Node* returned in a Query response.
nodeId	ExpandedNodeId	The *NodeId* for this *Node* description.
typeDefinitionNode	ExpandedNodeId	The *NodeId* for the type definition for this *Node* description.
values []	BaseDataType	Values for the selected *Attributes*. The order of returned items matches the order of the requested items. There is an entry for each requested item for the given *TypeDefinitionNode* that matches the selected instance, this includes any related nodes that were specified using a relative path from the selected instance's *TypeDefinitionNode*. If no values where found for a given requested item a null value is returned for that item. If a value has a bad status, the *StatusCode* is returned instead of the value. 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 *ReferenceDescription* is returned for that item. If the *QueryDataSet* is returned in a *QueryNext* to continue a list of *ReferenceDescription*, the *values* array will have the same size but the other values already returned are null.

7.23 ReadValueId

The components of this parameter are defined in Table 154.

Table 154 – ReadValueId

Name	Type	Description
ReadValueId	structure	Identifier for an item to read or to monitor.
nodeId	NodeId	*NodeId* of a *Node*.
attributeId	IntegerId	Id of the *Attribute*. This shall be a valid *Attribute* id. The *IntegerId* is defined in 7.13. The IntegerIds for the Attributes are defined in IEC 62541-6.
indexRange	NumericRange	This parameter is used to identify a single element of an array, or a single range of indexes for arrays. If a range of elements is specified, the values are returned as a composite. The first element is identified by index 0 (zero). The *NumericRange* type is defined in 7.21. This parameter is null if the specified *Attribute* is not an array. However, if the specified *Attribute* is an array, and this parameter is null, then all elements are to be included in the range.
dataEncoding	QualifiedName	This parameter specifies the *BrowseName* of the *DataTypeEncoding* that the *Server* should use when returning the Value *Attribute* of a *Variable*. It is an error to specify this parameter for other *Attributes*. A *Client* can discover what *DataTypeEncodings* are available by following the *HasEncoding Reference* from the *DataType Node* for a *Variable*. OPC UA defines *BrowseNames* which *Servers* shall recognize even if the *DataType Nodes* are not visible in the *Server* address space. These *BrowseNames* are: Default Binary The default or native binary (or non-XML) encoding. Default XML The default XML encoding. Each *DataType* shall support at least one of these encodings. *DataTypes* that do not have a true binary encoding (e.g. they only have a non-XML text encoding) should use the 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 shall identify one of the encodings as the Default XML encoding. Other standards bodies may define other well-known data encodings that could be supported. If this parameter is not specified then the *Server* shall choose either the Default Binary or Default XML encoding according to what *Message* encoding (see IEC 62541-6) is used for the *Session*. If the *Server* does not support the encoding that matches the *Message* encoding then the *Server* shall choose the default encoding that it does support. If this parameter is specified for a *MonitoredItem*, the *Server* shall set the *StructureChanged* bit in the *StatusCode* (see 7.33) if the *DataTypeEncoding* changes. The *DataTypeEncoding* changes if the *DataTypeVersion* of the *DataTypeDescription* or the *DataTypeDictionary* associated with the *DataTypeEncoding* changes.

7.24 ReferenceDescription

The components of this parameter are defined in Table 155.

Table 155 – ReferenceDescription

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

[1)] If the *Server* index indicates that the *TargetNode* is a remote *Node*, then the *browseName*, nodeClass and typeDefinition may be null or empty. If they are not, they might not be up to date because the local *Server* might not continuously monitor the remote *Server* for changes.

7.25 RelativePath

The components of this parameter are defined in Table 156.

Table 156 – RelativePath

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

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

A text format for the *RelativePath* can be found in Clause A.2. This format is used in examples that explain the *Services* that make use of the *RelativePath* structure.

7.26 RequestHeader

The components of this parameter are defined in Table 157.

Table 157 – RequestHeader

Name	Type	Description
RequestHeader	structure	Common parameters for all requests submitted on a *Session*.
authenticationToken	Session AuthenticationToken	The secret *Session* identifier used to verify that the request is associated with the *Session*. The *SessionAuthenticationToken* type is defined in 7.29.
timestamp	UtcTime	The time the *Client* sent the request. The parameter is only used for diagnostic and logging purposes in the server.
requestHandle	IntegerId	A *requestHandle* associated with the request. This client defined handle can be used to cancel the request. It is also returned in the response.
returnDiagnostics	UInt32	A bit mask that identifies the types of vendor-specific diagnostics to be returned in *diagnosticInfo* response parameters. The value of this parameter may consist of zero, one or more of the following values. No value indicates that diagnostics are not to be returned. Bit Value – Diagnostics to return 0x0000 0001 – ServiceLevel / SymbolicId 0x0000 0002 – ServiceLevel / LocalizedText 0x0000 0004 – ServiceLevel / AdditionalInfo 0x0000 0008 – ServiceLevel / Inner *StatusCode* 0x0000 0010 – ServiceLevel / Inner Diagnostics 0x0000 0020 – OperationLevel / SymbolicId 0x0000 0040 – OperationLevel / LocalizedText 0x0000 0080 – OperationLevel / AdditionalInfo 0x0000 0100 – OperationLevel / Inner *StatusCode* 0x0000 0200 – OperationLevel / Inner Diagnostics Each of these values is composed of two components, *level* and *type*, as described below. If none are requested, as indicated by a 0 value, or if no diagnostic information was encountered in processing of the request, then diagnostics information is not returned. *Level*: ServiceLevel – return diagnostics in the *diagnosticInfo* of the *Service*. OperationLevel – return diagnostics in the *diagnosticInfo* defined for individual operations requested in the *Service*. *Type*: SymbolicId – return a namespace-qualified, symbolic identifier for an error or condition. The maximum length of this identifier is 32 characters. LocalizedText – return up to 256 bytes of localized text that describes the symbolic id. AdditionalInfo – return a byte string that contains additional diagnostic information, such as a memory image. The format of this byte string is vendor-specific, and may depend on the type of error or condition encountered. InnerStatusCode – return the inner *StatusCode* associated with the operation or *Service*. InnerDiagnostics – return the inner diagnostic info associated with the operation or *Service*. The contents of the inner diagnostic info structure are determined by other bits in the mask. Note that setting this bit could cause multiple levels of nested diagnostic info structures to be returned.
auditEntryId	String	An identifier that identifies the *Client*'s security audit log entry associated with this request. An empty string value means that this parameter is not used. The *AuditEntryId* typically contains who initiated the action and from where it was initiated. The *AuditEventId* is included in the *AuditEvent* to allow human readers to correlate an *Event* with the initiating action. More details of the *Audit* mechanisms are defined in 6.2 and in IEC 62541-3.
timeoutHint	UInt32	This timeout in milliseconds is used in the *Client* side *Communication Stack* to set the timeout on a per-call base. For a *Server* this timeout is only a hint and can be used to cancel long running operations to free resources. If the Server detects a timeout, he can cancel the operation by sending the *Service* result *Bad_Timeout*. The *Server* should wait at minimum the timeout after he received the request before cancelling the operation. The *Server* shall check the *timeoutHint* parameter of a *PublishRequest* before processing a *PublishResponse*. If the request timed out, a Bad_Timeout *Service* result is sent and another *PublishRequest* is used. The value of 0 indicates no timeout.
additionalHeader	Extensible Parameter AdditionalHeader	Reserved for future use. Applications that do not understand the header should ignore it.

7.27 ResponseHeader

The components of this parameter are defined in Table 158.

Table 158 – ResponseHeader

Name	Type	Description
ResponseHeader	structure	Common parameters for all responses.
timestamp	UtcTime	The time the *Server* sent the response.
requestHandle	IntegerId	The requestHandle given by the *Client* to the request.
serviceResult	StatusCode	OPC UA-defined result of the *Service* invocation. The *StatusCode* type is defined in 7.33.
serviceDiagnostics	DiagnosticInfo	Diagnostic information for the *Service* invocation. This parameter is empty if diagnostics information was not requested in the request header. The *DiagnosticInfo* type is defined in 7.8.
stringTable []	String	There is one string in this list for each unique namespace, symbolic identifier, and localized text string contained in all of the diagnostics information parameters contained in the response (see 7.8). Each is identified within this table by its zero-based index.
additionalHeader	Extensible Parameter AdditionalHeader	Reserved for future use. Applications that do not understand the header should ignore it.

7.28 ServiceFault

The components of this parameter are defined in Table 159.

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

The exact use of this parameter depends on the mappings defined in IEC 62541-6.

Table 159 – ServiceFault

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

7.29 SessionAuthenticationToken

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

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

For the purposes of this discussion, a *Server* consists of application (code) and a *Communication Stack* as shown in Figure 29. The security provided by the *SessionAuthenticationToken* depends on a trust relationship between the *Server* application and the *Communication Stack*. The *Communication Stack* shall be able to verify the sender of the message and it uses the *SecureChannelId* or the *Client Certificate* to identify the sender to the *Server*. In these cases, the *SessionAuthenticationToken* is a UInt32 identifier that allows the *Server* to distinguish between different *Sessions* created by the same sender.

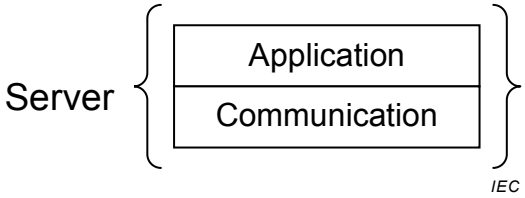


Figure 29 – Logical layers of a *Server*

In some cases, the application and the *Communication Stack* cannot exchange information at runtime which means the application will not have access to the *SecureChannelId* or the *Certificate* used to create the *SecureChannel.* In these cases the application shall create a random *ByteString* value that is at least 32 bytes long. This value shall be kept secret and shall always be exchanged over a *SecureChannel* with encryption enabled. The Administrator is responsible for ensuring that encryption is enabled. The *Profiles* in IEC 62541-7 may define additional requirements for a *ByteString SessionAuthenticationToken*.

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

Figure 30 illustrates the information exchanged between the Client, the Server and the Server *Communication Stack* when the *Client* obtains a *SessionAuthenticationToken.* In this figure the GetSecureChannelInfo step represents an API that depends on the *Communication Stack* implementation.

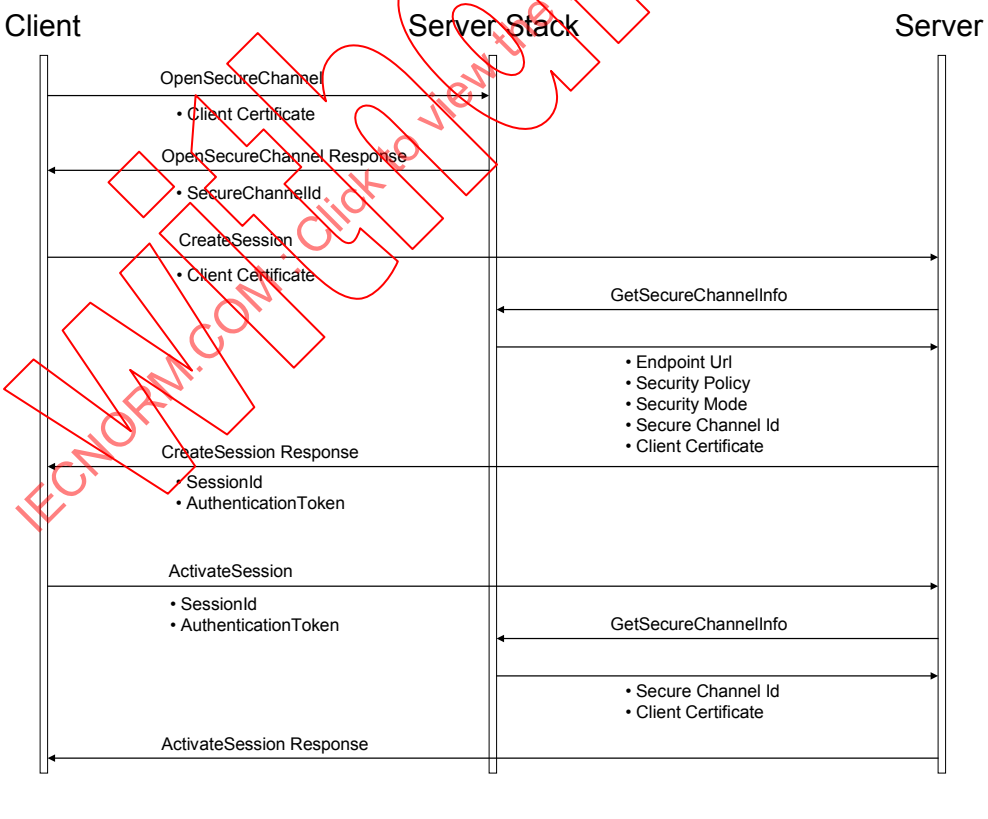


Figure 30 – Obtaining a SessionAuthenticationToken

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

7.30 SignatureData

The components of this parameter are defined in Table 160.

Table 160 – SignatureData

Name	Type	Description
SignatureData	structure	Contains a digital signature created with a *Certificate*.
signature	ByteString	This is a signature generated with the private key associated with a *Certificate*.
algorithm	String	A string containing the URI of the *algorithm*. The URI string values are defined as part of the security profiles specified in IEC 62541-7.

7.31 SignedSoftwareCertificate

A *SignedSoftwareCertificate* is a *ByteString* containing an encoded *Certificate*. The encoding of a *SignedSoftwareCertificate* depends on the security technology mapping and is defined completely in IEC 62541-6. Table 161 specifies the information that shall be contained in a *SignedSoftwareCertificate*.

Table 161 – SignedSoftwareCertificate

Name	Type	Description
SignedSoftwareCertificate	structure	A *SoftwareCertificate* with a signature created by Certifying Authority.
version	String	An identifier for the version of the *Certificate* encoding.
serialNumber	ByteString	A unique identifier for the *Certificate* assigned by the Issuer.
signatureAlgorithm	String	The algorithm used to sign the *Certificate*. The syntax of this field depends on the *Certificate* encoding.
signature	ByteString	The signature created by the Issuer.
issuer	Structure	A name that identifies the *Issuer Certificate* used to create the signature.
validFrom	UtcTime	When the *Certificate* becomes valid.
validTo	UtcTime	When the *Certificate* expires.
subject	Structure	A name that identifies the product which the *Certificate* describes. This field shall contain the *productName* and *vendorName* from the *SoftwareCertificate*.
subjectAltName []	Structure	A list of alternate names for the product. This list shall include the *productUri* specified in the *SoftwareCertificate*.
publicKey	ByteString	The public key associated with the *Certificate*.
keyUsage []	String	Specifies how the *Certificate* key may be used. *SignedSoftwareCertificate* may only be used for creating Digital Signatures and Non-Repudiation. The contents of this field depends on the *Certificate* encoding.
softwareCertificate	ByteString	The XML encoded form of the *SoftwareCertificate* stored as UTF8 text. IEC 62541-6 describes the XML representation for the *SoftwareCertificate*.

7.32 SoftwareCertificate

The components of this parameter are defined in Table 162.

Table 162 – SoftwareCertificate

Name	Type	Description
SoftwareCertificate	Structure	A *Certificate* describing a product.
productName	String	The name of the product that is certified. This field shall be specified.
productUri	String	A globally unique identifier for the product that is certified. This field shall be specified.
vendorName	String	The name of the vendor responsible for the product. This field shall be specified.
vendorProductCertificate	ByteString	The DER encoded form of the X509 *Certificate* which is assigned to the product by the vendor. This field may be omitted.
softwareVersion	String	Software version. This field shall be specified.
buildNumber	String	Build number. This field shall be specified.
buildDate	UtcTime	Date and time of the build. This field shall be specified.
issuedBy	String	URI of the certifying authority. This field shall be specified.
issueDate	UtcTime	Specifies when the *Certificate* was issued by the certifying authority. This field shall be specified.
supportedProfiles []	structure	List of supported *Profiles*. This structure is defined in-line with the following indented items.
organizationUri	String	A URI that identifies the organization that defined the profile.
profileId	String	A string that identifies the *Profile*.
complianceTool	String	A string that identifies the tool or certification method used for compliance testing.
complianceDate	UtcTime	Date and time of the compliance test.
complianceLevel	Enum ComplianceLevel	An enumeration that specifies the compliance level of the *Profile*. It has the following values: UNTESTED_0 the profiled capability has not been tested successfully PARTIAL_1 the profiled capability has been partially tested and has passed critical tests, as defined by the certifying authority. SELFTESTED_2 the profiled capability has been successfully tested using a self-test system authorized by the certifying authority. CERTIFIED_3 the profiled capability has been successfully tested by a testing organisation authorized by the certifying authority.
unsupportedUnitIds []	String	The identifiers for the optional conformance units that were not tested. See IEC 62541-7 for a detailed explanation.

7.33 StatusCode

7.33.1 General

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

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

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

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

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

The exact bit assignments are shown in Table 163.

Table 163 – StatusCode Bit Assignments

Field	Bit Range	Description
Severity	30:31	Indicates whether the *StatusCode* represents a good, bad or uncertain condition. These bits have the following meanings: Good Success 00 Indicates that the operation was successful and the associated results may be used. Uncertain Warning 01 Indicates that the operation was partially successful and that associated results might not be suitable for some purposes. Bad Failure 10 Indicates that the operation failed and any associated results cannot be used. Reserved 11 Reserved for future use. All *Clients* should treat a *StatusCode* with this severity as “Bad”.
Reserved	29:29	Reserved for use in OPC UA application specific APIs. This bit shall always be zero on the wire but may be used by OPC UA application specific APIs for API specific status codes.
Reserved	2828	Reserved for future use. Shall always be zero.
SubCode	16:27	The code is a numeric value assigned to represent different conditions. Each code has a symbolic name and a numeric value. All descriptions in the OPC UA specification refer to the symbolic name. IEC 62541-6 maps the symbolic names onto a numeric value.
StructureChanged	15:15	Indicates that the structure of the associated data value has changed since the last *Notification*. *Clients* should not process the data value unless they re-read the metadata. *Servers* shall set this bit if the *DataTypeEncoding* used for a *Variable* changes. 7.23 describes how the *DataTypeEncoding* is specified for a *Variable*. *Servers* shall also set this bit if the *EnumStrings Property* of the *DataType* of the *Variable* changes. This bit is provided to warn *Clients* that parse complex data values that their parsing routines could fail because the serialized form of the data value has changed. This bit has meaning only for *StatusCodes* returned as part of a data change *Notification* or the *HistoryRead*. *StatusCodes* used in other contexts shall always set this bit to zero.
SemanticsChanged	14:14	Indicates that the semantics of the associated data value have changed. *Clients* should not process the data value until they re-read the metadata associated with the *Variable*. *Servers* should set this bit if the metadata has changed in way that could cause application errors if the *Client* does not re-read the metadata. For example, a change to the engineering units could create problems if the *Client* uses the value to perform calculations. IEC 62541-8 defines the conditions where a *Server* shall set this bit for a DA *Variable*. Other specifications may define additional conditions. A *Server* may define other conditions that cause this bit to be set. This bit has meaning only for *StatusCodes* returned as part of a data change *Notification* or the *HistoryRead*. *StatusCodes* used in other contexts shall always set this bit to zero.
Reserved	12:13	Reserved for future use. Shall always be zero.
InfoType	10:11	The type of information contained in the info bits. These bits have the following meanings: NotUsed 00 The info bits are not used and shall be set to zero. DataValue 01 The *StatusCode* and its info bits are associated with a data value returned from the *Server*. This flag is only used in combination with *StatusCodes* defined in IEC 62541-8. Reserved 1X Reserved for future use. The info bits shall be ignored.
InfoBits	0:9	Additional information bits that qualify the *StatusCode*. The structure of these bits depends on the Info Type field.

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

Table 164 – DataValue InfoBits

Info Type	Bit Range	Description
LimitBits	8:9	The limit bits associated with the data value. The limits bits have the following meanings: **Limit** **Bits** **Description** None 00 The value is free to change. Low 01 The value is at the lower limit for the data source. High 10 The value is at the higher limit for the data source. Constant 11 The value is constant and cannot change.
Overflow	7	This bit shall only be set if the *MonitoredItem* queue size is greater than 1. If this bit is set, not every detected change has been returned since the *Server*'s queue buffer for the *MonitoredItem* reached its limit and had to purge out data.
Reserved	5:6	Reserved for future use. Shall always be zero.
HistorianBits	0:4	These bits are set only when reading historical data. They indicate where the data value came from and provide information that affects how the *Client* uses the data value. The historian bits have the following meaning: Raw XXX00 A raw data value. Calculated XXX01 A data value which was calculated. Interpolated XXX10 A data value which was interpolated. Reserved XXX11 Undefined. Partial XX1XX A data value which was calculated with an incomplete interval. Extra Data X1XXX A raw data value that hides other data at the same timestamp. Multi Value 1XXXX Multiple values match the *Aggregate* criteria (i.e. multiple minimum values at different timestamps within the same interval). IEC 62541-11 describes how these bits are used in more detail.

7.33.2 Common StatusCodes

Table 165 defines the common *StatusCodes* for all *Service* results used in more than one service. It does not provide a complete list. These *StatusCodes* may also be used as operation level result code. IEC 62541-6 maps the symbolic names to a numeric value and provides a complete list of StatusCodes including codes defines in other parts.

Table 165 – Common Service Result Codes

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

Symbolic Id	Description
Bad_ViewTimestampInvalid	The view timestamp is not available or not supported.
Bad_ViewParameterMismatchInvalid	The view parameters are not consistent with each other.
Bad_ViewVersionInvalid	The view version is not available or not supported.

Table 166 defines the common *StatusCodes* for all operation level results used in more than one service. It does not provide a complete list. IEC 62541-6 maps the symbolic names to a numeric value and provides a complete list of StatusCodes including codes defines in other parts. The common *Service* result codes can be also contained in the operation level.

Table 166 – Common Operation Level Result Codes

Symbolic Id	Description
Good_Clamped	The value written was accepted but was clamped.
Good_Overload	Sampling has slowed down due to resource limitations.
Uncertain	The value is uncertain but no specific reason is known.
Bad	The value is bad but no specific reason is known.
Bad_AttributeIdInvalid	The attribute is not supported for the specified node.
Bad_BrowseDirectionInvalid	The browse direction is not valid.
Bad_BrowseNameInvalid	The browse name is invalid.
Bad_ContentFilterInvalid	The content filter is not valid.
Bad_ContinuationPointInvalid	The continuation point provided is not longer valid. This status is returned if the continuation point was deleted or the address space was changed between the browse calls.
Bad_DataEncodingInvalid	The data encoding is invalid. This result is used if no *dataEncoding* can be applied because an *Attribute* other than *Value* was requested or the *DataType* of the *Value Attribute* is not a subtype of the *Structure DataType*.
Bad_DataEncodingUnsupported	The server does not support the requested data encoding for the node. This result is used if a *dataEncoding* can be applied but the passed data encoding is not known to the *Server*.
Bad_EventFilterInvalid	The event filter is not valid.
Bad_FilterNotAllowed	A monitoring filter cannot be used in combination with the attribute specified.
Bad_FilterOperandInvalid	The operand used in a content filter is not valid.
Bad_HistoryOperationInvalid	The history details parameter is not valid.
Bad_HistoryOperationUnsupported	The server does not support the requested operation.
Bad_IndexRangeInvalid	The syntax of the index range parameter is invalid.
Bad_IndexRangeNoData	No data exists within the range of indexes specified.
Bad_MonitoredItemFilterInvalid	The monitored item filter parameter is not valid.
Bad_MonitoredItemFilterUnsupported	The server does not support the requested monitored item filter.
Bad_MonitoredItemIdInvalid	The monitoring item id does not refer to a valid monitored item.
Bad_MonitoringModeInvalid	The monitoring mode is invalid.
Bad_NoCommunication	Communication with the data source is defined, but not established, and there is no last known value available. This status/sub-status is used for cached values before the first value is received or for Write and Call if the communication is not established.
Bad_NoContinuationPoints	The operation could not be processed because all continuation points have been allocated.
Bad_NodeClassInvalid	The node class is not valid.
Bad_NodeIdInvalid	The syntax of the node id is not valid.
Bad_NodeIdUnknown	The node id refers to a node that does not exist in the server address space.
Bad_NoDeleteRights	The *Server* will not allow the node to be deleted.
Bad_NodeNotInView	The nodesToBrowse is not part of the view.
Bad_NotFound	A requested item was not found or a search operation ended without success.
Bad_NotImplemented	Requested operation is not implemented.
Bad_NotReadable	The access level does not allow reading or subscribing to the *Node*.
Bad_NotSupported	The requested operation is not supported.
Bad_NotWritable	The access level does not allow writing to the *Node*.
Bad_ObjectDeleted	The object cannot be used because it has been deleted.
Bad_OutOfRange	The value was out of range.
Bad_ReferenceTypeIdInvalid	The reference type id does not refer to a valid reference type node.
Bad_SourceNodeIdInvalid	The source node id does not refer to a valid node.
Bad_StructureMissing	A mandatory structured parameter was missing or null.

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

7.34 TimestampsToReturn

The *TimestampsToReturn* is an enumeration that specifies the *Timestamp Attributes* to be transmitted for *MonitoredItems* or *Nodes* in *Read* and *HistoryRead*. The values of this parameter are defined in Table 167.

Table 167 – TimestampsToReturn Values

Value	Description
SOURCE_0	Return the source timestamp.
SERVER_1	Return the *Server* timestamp.
BOTH_2	Return both the source and *Server* timestamps.
NEITHER_3	Return neither timestamp. This is the default value for *MonitoredItems* if a *Variable* value is not being accessed. For *HistoryRead* this is not a valid setting.

7.35 UserIdentityToken parameters

7.35.1 Overview

The *UserIdentityToken* structure used in the *Server Service Set* allows *Clients* to specify the identity of the user they are acting on behalf of. The exact mechanism used to identify users depends on the system configuration. The different types of identity tokens are based on the most common mechanisms that are used in systems today. Table 168 defines the current set of user identity tokens. The *ExtensibleParameter* type is defined in 7.11.

Table 168 – UserIdentityToken parameterTypeIds

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

The *Client* shall always prove possession of a *UserIdentityToken* when it passes it to the *Server*. Some tokens include a secret such as a password which the *Server* will accept as proof. In order to protect these secrets the *Token* may be encrypted before it is passed to the *Server*. The profiles defined in IEC 62541-7 specify where encryption shall be applied. Other types of tokens allow the *Client* to create a signature with the secret associated with the *Token*. In these cases, the *Client* proves possession of a *UserIdentityToken* by appending the last *ServerNonce* to the *ServerCertificate* and uses the secret to produce a *Signature* which is passed to the *Server*.

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

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

Table 169 – UserIdentityToken Encrypted Token Format

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

7.35.2 AnonymousIdentityToken

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

Table 170 defines the *AnonymousIdentityToken* parameter.

Table 170 – AnonymousIdentityToken

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

7.35.3 UserNameIdentityToken

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

This token shall be encrypted by the *Client* if required by the *Server* in the *SecurityPolicy* of the *UserTokenPolicy* in the *EndpointDescription*. The *Server* should specify a *SecurityPolicy* for the *UserTokenPolicy* if the *SecureChannel* has a *SecurityPolicy* of None and no transport layer encryption is available. If None is specified for the *UserTokenPolicy* and *SecurityPolicy* is None then the password only contains the UTF-8 encoded password. This configuration should not be used unless the network is encrypted in some other manner such as a VPN. The use of this configuration without network encryption would result in a serious security fault, in that it would cause the appearance of a secure user access, but it would make the password visible in clear text.

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

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

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

Table 171 defines the *UserNameIdentityToken* parameter.

Table 171 – UserNameIdentityToken

Name	Type	Description
UserNameIdentityToken	structure	UserName value.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.
userName	String	A string that identifies the user.
password	ByteString	The password for the user. The password can be an empty string. This parameter shall be encrypted with the Server's Certificate using the algorithm specified by the *SecurityPolicy*.
encryptionAlgorithm	String	A string containing the URI of the *AsymmetricEncryptionAlgorithm*. The URI string values are defined names that may be used as part of the security profiles specified in IEC 62541-7. This parameter is null if the password is not encrypted.

Table 172 describes the dependencies for selecting the *AsymmetricEncryptionAlgorithm* for the *UserNameIdentityToken*. The *SecureChannel SecurityPolicy* URI is specified in the *EndpointDescription* and used in subsequent OpenSecureChannel request. The *UserTokenPolicy SecurityPolicy* URI is specified in the *EndpointDescription*. The *encryptionAlgorithm* is specified in the *UserNameIdentityToken* or *IssuedIdentityToken* provided by the *Client* in the *ActivateSession* call.

Table 172 – EncryptionAlgorithm selection

SecureChannel SecurityPolicy	UserTokenPolicy SecurityPolicy	UserNameIdentityToken EncryptionAlgorithm
Security Policy – None	Null	Null
Security Policy – None	Security Policy – None	Null
Security Policy – None	Security Policy – Other	Asymmetric algorithm for "Other"
Security Policy – Other	Null	Asymmetric algorithm for "Other"
Security Policy – Other	Security Policy – Yet another	Asymmetric algorithm for "Yet another"
Security Policy – Other	Security Policy – Other	Asymmetric algorithm for "Other"
Security Policy – Other	Security Policy – None	Null

7.35.4 X509IdentityTokens

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

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

Table 173 defines the X509IdentityToken parameter.

Table 173 – X509 Identity Token

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

7.35.5 IssuedIdentityToken

The *IssuedIdentityToken* is used to pass WS-Security compliant *SecurityTokens* to the *Server*.

WS-Security defines a number of token profiles that may be used to represent different types of *SecurityTokens*. For example, Kerberos and SAML tokens have WSS token profiles and shall be exchanged in OPC UA as XML Security Tokens.

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

These tokens may be encrypted or require a signature. IEC 62541-7 defines profiles that include user related security; they also include any requirements for encryption and signatures. Additional security profiles specify encryption and signature algorithms.

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

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

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

Table 174 defines the *IssuedIdentityToken* parameter.

Table 174 – IssuedIdentityToken

Name	Type	Description
IssuedIdentityToken	structure	WSS value.
policyId	String	An identifier for the *UserTokenPolicy* that the token conforms to. The *UserTokenPolicy* structure is defined in 7.36.
tokenData	ByteString	The XML representation of the token. This parameter may be encrypted with the *Server's Certificate*.
encryptionAlgorithm	String	A string containing the URI of the *AsymmetricEncryptionAlgorithm*. The list of OPC UA-defined names that may be used is specified in IEC 62541-7. This parameter is null if the token is not encrypted. The *encryptionAlgorithm* is *AsymmetricEncryptionAlgorithm* for the security policy identified by the *securityPolicyUri* for the *UserTokenPolicy*.

7.36 UserTokenPolicy

The components of this parameter are defined in Table 175.

Table 175 – UserTokenPolicy

Name	Type	Description
UserTokenPolicy	structure	Specifies a *UserIdentityToken* that a *Server* will accept.
policyId	String	An identifier for the UserTokenPolicy assigned by the Server. The Client specifies this value when it constructs a UserIdentityToken that conforms to the policy. This value is only unique within the context of a single Server.
tokenType	Enum UserIdentity TokenType	The type of user identity token required. This value is an enumeration with one of the following values: ANONYMOUS_0 No token is required. USERNAME_1 A username/password token. CERTIFICATE_2 An X509v3 *Certificate* token. ISSUEDTOKEN_3 Any WS-Security defined token. A tokenType of ANONYMOUS indicates that the *Server* does not require any user identification. In this case the *Client Application Instance Certificate* is used as the user identification.
issuedTokenType	String	A URI for the type of token. IEC 62541-6 defines URIs for common issued token types. Vendors may specify their own token. This field may only be specified if *TokenType* is ISSUEDTOKEN_3.
issuerEndpointUrl	String	An optional URL for the token issuing service. The meaning of this value depends on the *issuedTokenType.*
securityPolicyUri	String	The security policy to use when encrypting or signing the *UserIdentityToken* when it is passed to the *Server* in the *ActivateSession* request. Clause 7.35 describes how this parameter is used. The security policy for the SecureChannel is used if this value is omitted.

7.37 ViewDescription

The components of this parameter are defined in Table 176.

Table 176 – ViewDescription

Name	Type	Description
ViewDescription	structure	Specifies a *View*.
viewId	NodeId	*NodeId* of the *View* to *Query*. A null value indicates the entire *AddressSpace*.
timestamp	UtcTime	The time date desired. The corresponding version is the one with the closest previous creation timestamp. Either the *Timestamp* or the *viewVersion* parameter may be set by a *Client*, but not both. If *ViewVersion* is set this parameter shall be null.
viewVersion	UInt32	The version number for the *View* desired. When *Nodes* are added to or removed from a *View*, the value of a View's *ViewVersion Property* is updated. Either the *Timestamp* or the *viewVersion* parameter may be set by a *Client*, but not both. The ViewVersion *Property* is defined in IEC 62541-3. If *timestamp* is set this parameter shall be 0. The current view is used if timestamp is null and viewVersion is 0.

Annex A
(informative)

BNF definitions

A.1 Overview over BNF

The BNF (Backus-Naur form) used in this annex uses `<´ and `>´ to mark symbols, `[´ and `]´ to identify optional paths and `|´ to identify alternatives. If the ‘(‘ and ‘)’ symbols are used, it indicates sets.

A.2 BNF of RelativePath

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

The components of a *RelativePath* text format are specified in Table A.1.

Table A.1 – RelativePath

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

Table A.2 provides *RelativePaths* examples in text format.

Table A.2 – *RelativePath* Examples

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

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

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

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

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

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

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

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

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

<name-char>      ::= All valid characters for a String (see IEC 62541-3) excluding
                     reserved-chars.
```

A.3 BNF of NumericRange

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

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

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

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

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

Annex B
(informative)

Content Filter and Query Examples

B.1 Simple ContentFilter examples

B.1.1 Overview

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

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

- AttributeOperand: Refers to a *Node,* an *Attribute* of a *Node* or the *Value Attribute* of a *Property* associated with a *Node*. In the examples, the character names of NodeIds are used instead of an actual nodeId, this also applies to Attribute Ids.
- The string representation of relative paths is used instead of the actual structure.
- The NamespaceIndex used in all examples is 12 (it could just as easily have been 4 or 23 or any value). For more information about NamespaceIndex, see IEC 62541-3. The use of the NamespaceIndex illustrates that the information model being used in the examples is not a model defined by this standard, but one created for the examples.

B.1.2 Example 1

For example the logic describe by '(((AType.A = 5) or InList(BType.B, 3,5,7)) and BaseObjectType.displayName LIKE "Main%")' would result in a logic tree as shown in Figure B.1 and a ContentFilter as shown in Table B.1. For this example to return anything AType and BType shall both be subtypes of BaseObjectType, or the resulting "And" operation would always be false.

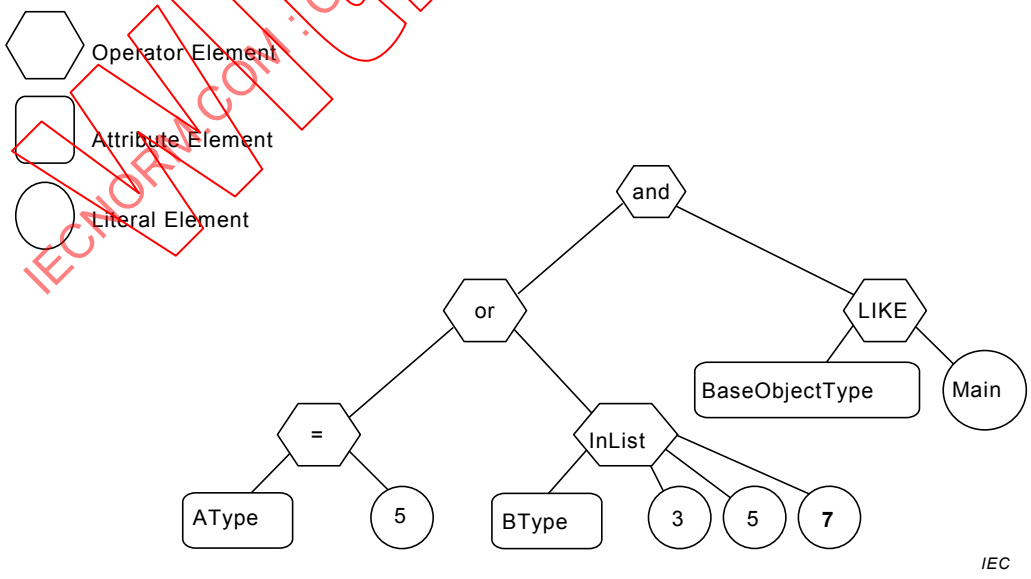


Figure B.1 – Filter Logic Tree Example

Table B.1 describes the elements, operators and operands used in the example.

Table B.1 – ContentFilter Example

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	And	ElementOperand = 1	Element Operand = 4		
1	Or	ElementOperand = 2	Element Operand = 3		
2	Equals	AttributeOperand = NodeId: AType, BrowsePath: ".12:A", Attribute:value	LiteralOperand = '5'		
3	InList	AttributeOperand = NodeId: BType, BrowsePath: ".12:B", Attribute:value	LiteralOperand = '3'	LiteralOperand = '5'	LiteralOperand = '7'
4	Like	AttributeOperand = NodeId: BaseObjectType, BrowsePath: ".", Attribute: displayName	LiteralOperand = "Main%"		

B.1.3 Example 2

As another example a filter to select all *SystemEvents* (including derived types) that are contained in the Area1 *View* or the Area2 *View* would result in a logic tree as shown in Figure B.2 and a ContentFilter as shown in Table B.2.

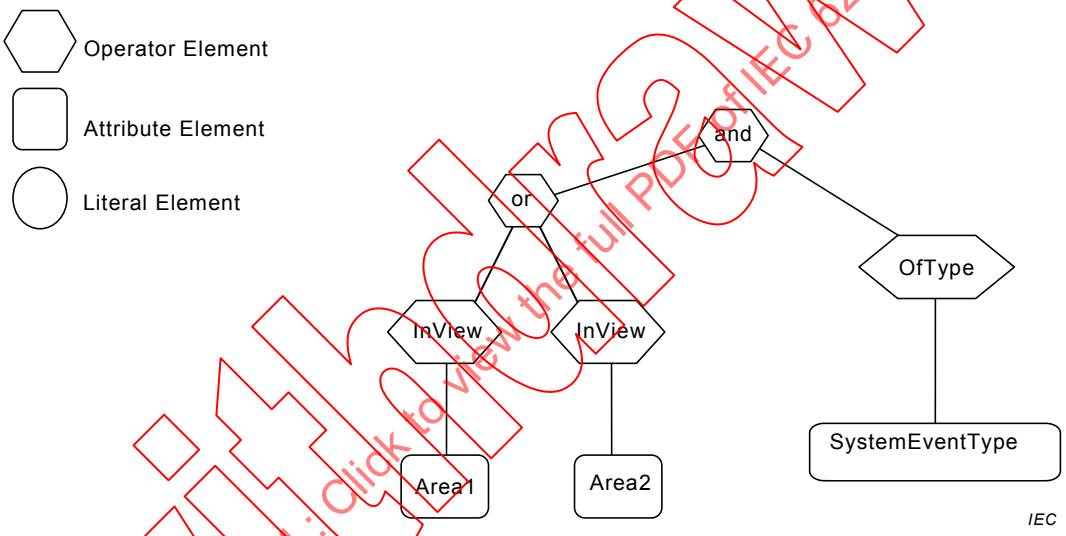


Figure B.2 – Filter Logic Tree Example

Table B.2 describes the elements, operators and operands used in the example.

Table B.2 – ContentFilter Example

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

B.2 Complex Examples of Query Filters

B.2.1 Overview

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

- AttributeOperand: Refers to a *Node*, an *Attribute* of a *Node* or the *Value Attribute* of a *Property* associated with a *Node*. In the examples character names of *ExpandedNodeId* are used instead of an actual *ExpandedNodeId*, this also applies to *Attribute* Ids.
- The string representation of relative paths is used instead of the actual structure.
- The *NamespaceIndex* used in all examples is 12 (it could just as easily have been 4 or 23 or any value). For more information about *NamespacesIndex*, see IEC 62541-3. The use of the *NamespaceIndex* illustrates that the information model being used in the examples is not a model defined by this standard, but one created for the examples.

B.2.2 Used type model

The following examples use the type model described below. All *Property* values are assumed to be string unless otherwise noted

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 subtype 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" (Stored as an *Int32*).

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

This example type system is shown in Figure B.3. In this Figure, the OPC UA notation is used for all *References* to *ObjectTypes*, *Variables*, *Properties* and subtypes. 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 are subtypes of the listed *Reference*.

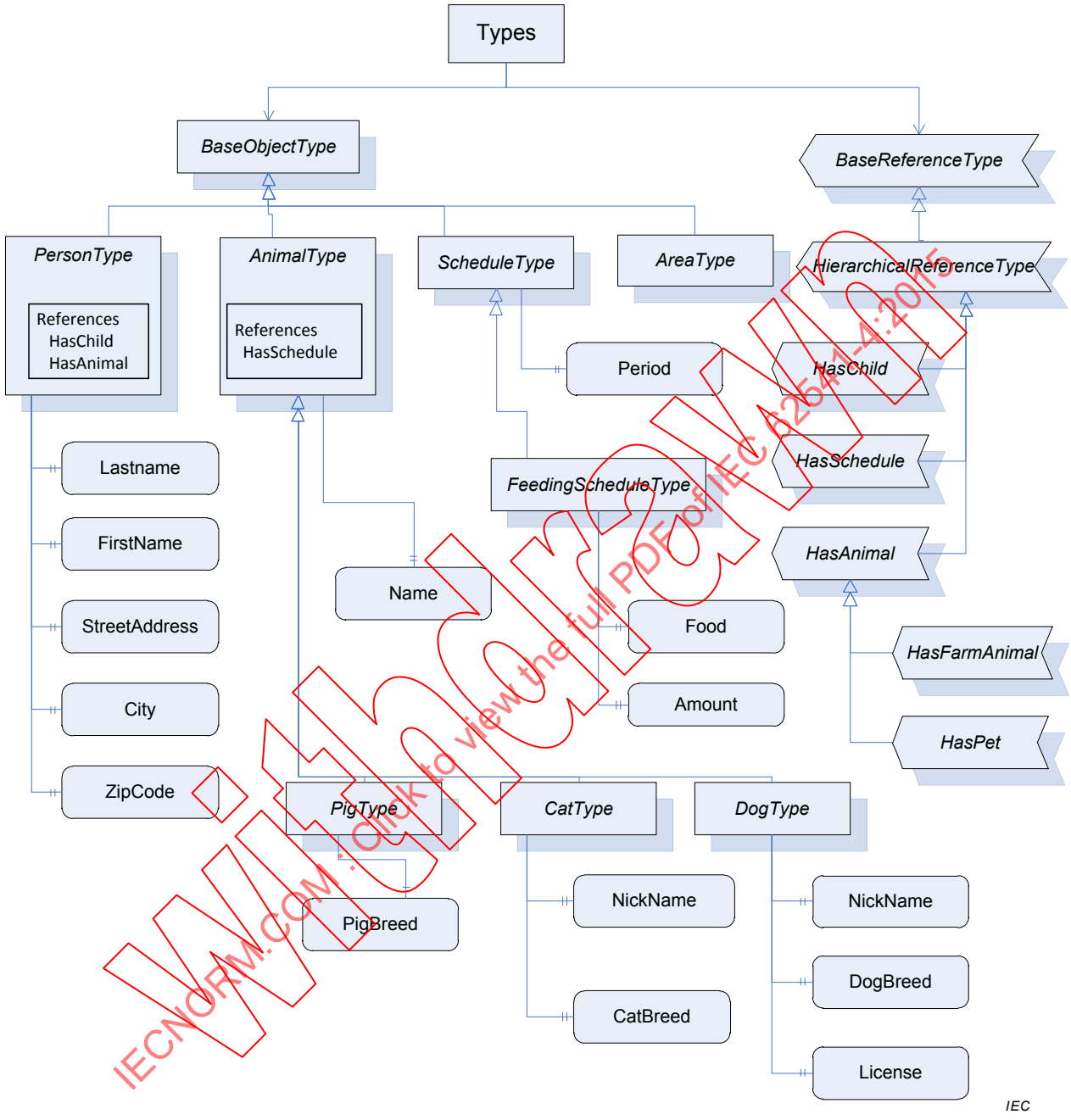


Figure B.3 – Example Type Nodes

A corresponding example set of instances is shown in Figure B.4. 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 *ReferenceTypes* use the OPC UA notation, custom *ReferenceTypes* are listed as 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 coloured box. Two Area nodes are included for grouping of the existing person nodes. All custom nodes are defined in namespace 12 which is not included in Figure B.4.

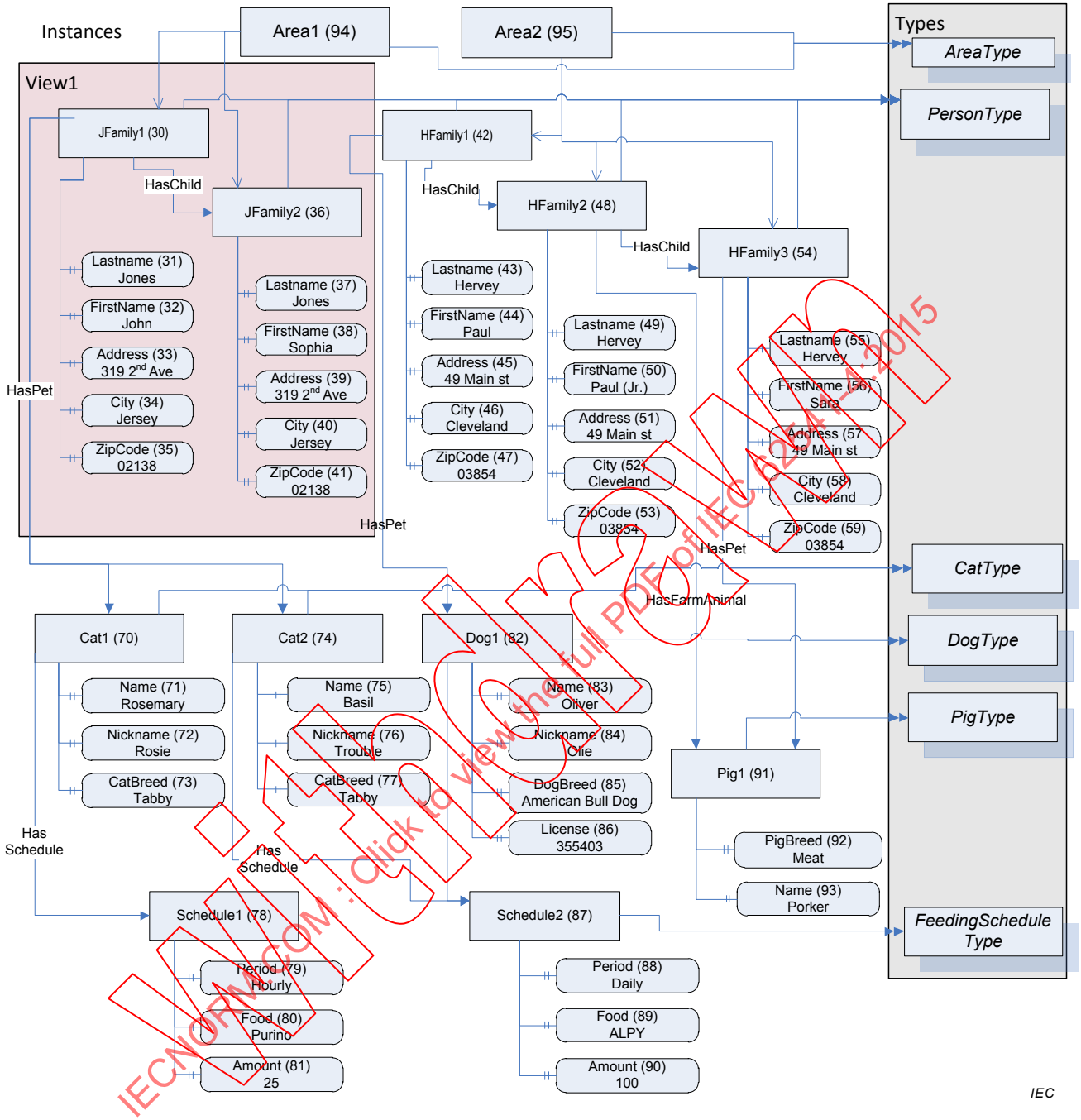


Figure B.4 – Example Instance Nodes

B.2.3 Example Notes

For all of the examples in 7.4.4, the type definition *Node* is listed in its symbolic form, in the actual call it would be the *ExpandedNodeId* assigned to the *Node*. The *Attribute* is the symbolic name of the *Attribute*, in the actual call they would be translated to the *IntegerId* of the *Attribute*. Also in all of the examples the *BrowseName* is included in the result table for clarity; normally this would not be returned.

All of the examples include the following items:

- an English description of the object of the query,
- an SQL like description of the query,

- a table that has a NodeTypeDescription of the items that are to be returned
- a figure illustrating the query filter.
- A table describing the content filer
- A table describing the resulting dataset

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

B.2.4 Example 1

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

Example 1: Get PersonType.LastName, AnimalType.Name, ScheduleType.Period where the Person Has a Pet and that Pet Has a Schedule.

The *NodeTypeDescription* parameters used in the example are described in Table B.3.

Table B.3 – Example 1 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.5.

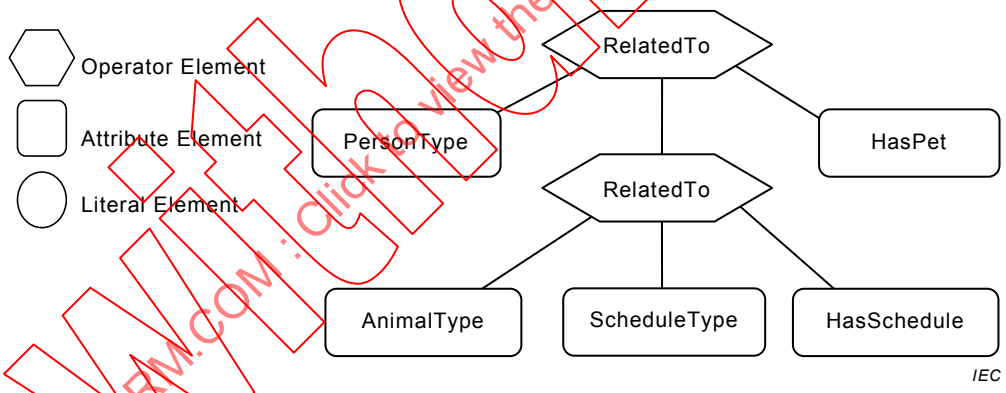


Figure B.5 – Example 1 Filter

Table B.4 describes the *ContentFilter* elements, operators and operands used in the example.

Table B.4 – Example 1 ContentFilter

Element[]	filterOperator	filterOperand[0]	filterOperand[1]	filterOperand[2]	filterOperand[3]
1	RelatedTo_15	AttributeOperand = Nodeid: PersonType, BrowsePath ".", Attribute: NodeId	ElementOperand = 2	AttributeOperand = NodeId: HasPet, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'
2	RelatedTo_15	AttributeOperand = NodeId: AnimalType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: ScheduleType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: HasSchedule, BrowsePath ".", Attribute: NodeId	LiteralOperand= '1'

Table B.5 describes the *QueryDataSet* that results from this query if it were executed against the instances described in Figure B.4

Table B.5 – Example 1 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (JFamily1)	PersonType	".12:LastName"	Jones
		"<12:HasPet>12:AnimalType. 12:Name"	Rosemary
			Basil
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule.12:Period"	Hourly
			Daily
12:42(HFamily1)	PersonType	".12:LastName"	Hervey
		"<12:HasPet>12:AnimalType. 12:Name"	Oliver
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule.12:Period"	Daily

NOTE The RelativePath 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.

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 for the .<HasPet>.AnimalType.Name item. They are show as separate rows for ease of viewing. The actual value array for JFamily1 would be ("Jones", {"RoseMary", "Basil"}, {"Hourly", "Daily"})

B.2.5 Example 2

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

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

The NodeTypeDescription parameters used in the example are described in Table B.6.

Table B.6 – Example 2 NodeTypeDescription

Type Definition Node	Include Subtypes	QueryDataDescription		
		Relative Path	Attribute	Index Range
PersonType	FALSE	".12:LastName"	Value	N/A
AnimalType	TRUE	".12:Name"	Value	N/A

The corresponding ContentFilter is illustrated in Figure B.6.

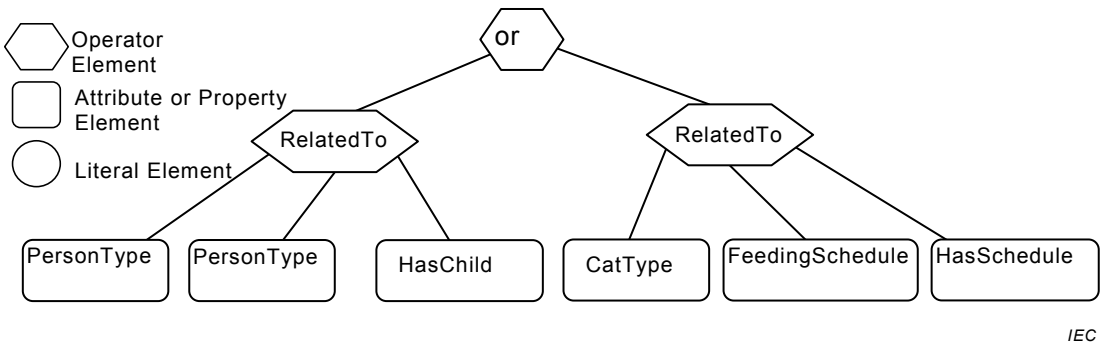


Figure B.6 – Example 2 Filter Logic Tree

Table B.7 describes the elements, operators and operands used in the example. It is worth noting that a CatType is a subtype of AnimalType.

Table B.7 – Example 2 ContentFilter

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

The results from this query would contain the *QueryDataSets* shown in Table B.8.

Table B.8 – Example 2 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (Jfamily1)	PersonType	. 12:LastName	Jones
12:42 (HFamily1)	PersonType	. 12:LastName	Hervey
12:48 (HFamily2)	PersonType	. 12:LastName	Hervey
12:70 (Cat1)	CatType	. 12:Name	Rosemary
12:74 (Cat2)	CatType	. 12:Name	Basil

NOTE The relative path column and browse name (in parentheses in the *NodeId* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinitionNodeId* would be a *NodeId* not the symbolic name that is included in the table.

B.2.6 Example 3

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

Example 3: Get PersonType.LastName, AnimalType.Name, ScheduleType.Period where a person has a pet and the animal has a feeding schedule and the person has a Zipcode = '02138' and (the Schedule.Period is Daily or Hourly) and Amount to feed is > 10.

Table B.9 describes the NodeTypeDescription parameters used in the example.

Table B.9 – Example 3 – NodeTypeDescription

Type Definition Node	Include Subtypes	QueryDataDescription		
		RelativePath	Attribute	Index Range
PersonType	FALSE	"12:LastName"	Value	N/A
		"<12:HasPet>12:AnimalType. 12:Name"	Value	N/A
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:FeedingSchedule.Period"	Value	N/A

The corresponding *ContentFilter* is illustrated in Figure B.7.

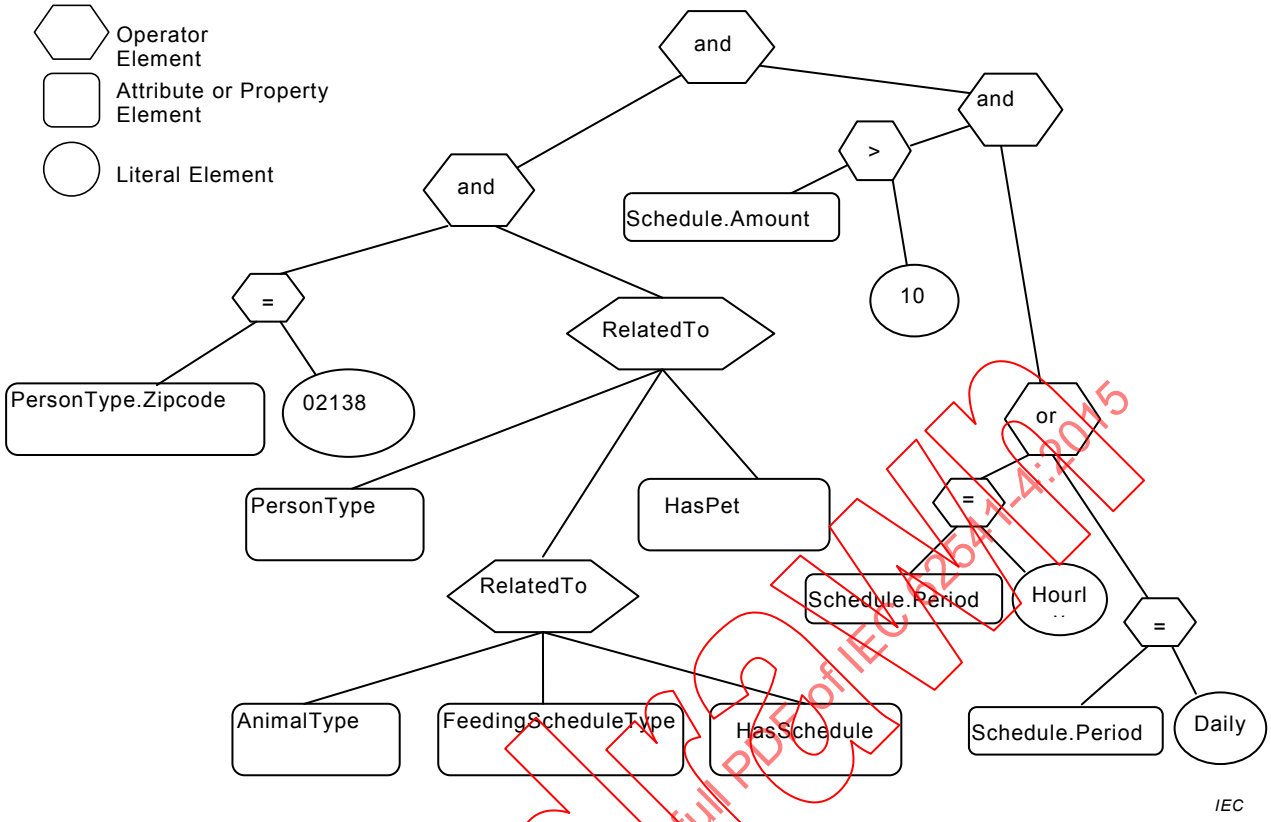


Figure B.7 – Example 3 Filter Logic Tree

Table B.10 describes the elements, operators and operands used in the example.

Table B.10 – Example 3 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	And	Element Operand= 1	ElementOperand = 2		
1	And	ElementOperand = 4	ElementOperand = 6		
2	And	ElementOperand = 3	ElementOperand = 9		
3	Or	ElementOperand = 7	ElementOperand = 8		
4	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", Attribute: NodeId	ElementOperand = 5	AttributeOperand = NodeId: 12:HasPet, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'
5	RelatedTo	AttributeOperand = Node: 12:AnilmalType, BrowsePath ".", Attribute: NodeId Alias: AT	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath ".", Attribute: NodeId Alias: FST	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'
6	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath 12:Zipcode ".", Attribute: Value	LiteralOperand = '02138'		
7	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath "12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule/12:Period", Attribute: Value Alias: FST	LiteralOperand = 'Daily'		
8	Equals	AttributeOperand = NodeId: 12:PersonType BrowsePath "12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule/12:Period", Attribute: Value Alias: FST	LiteralOperand = 'Hourly'		
9	Greater Than	AttributeOperand = NodeId: 12:PersonType BrowsePath "12:HasPet>12:AnimalType<12: HasSchedule>12: FeedingSchedule/12:Amount", Attribute: Value Alias: FST	ElementOperand = 10		
10	Cast	LiteralOperand = 10	AttributeOperand = NodeId: Int32, BrowsePath ".", Attribute: NodeId		

The results from this query would contain the *QueryDataSets* shown in Table B.11.

Table B.11 – Example 3 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:30 (JFamily1)	PersonType	".12:LastName"	Jones
		"<12:HasPet>12:PersonType. 12:Name"	Rosemary
			Basil
		"<12:HasPet>12:AnimalType<12:HasSchedule>12:FeedingSchedule. 12:Period"	Hourly
			Daily

NOTE The RelativePath column and browse name (in parentheses in the *NodeId* column) are not in the *QueryDataSet* and are only shown here for clarity. The TypeDefinitionNodeId would be an integer not the symbolic name that is included in the table.

B.2.7 Example 4

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

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

Table B.12 describes the NodeTypeDescription parameters used in the example.

Table B.12 – Example 4 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.8.

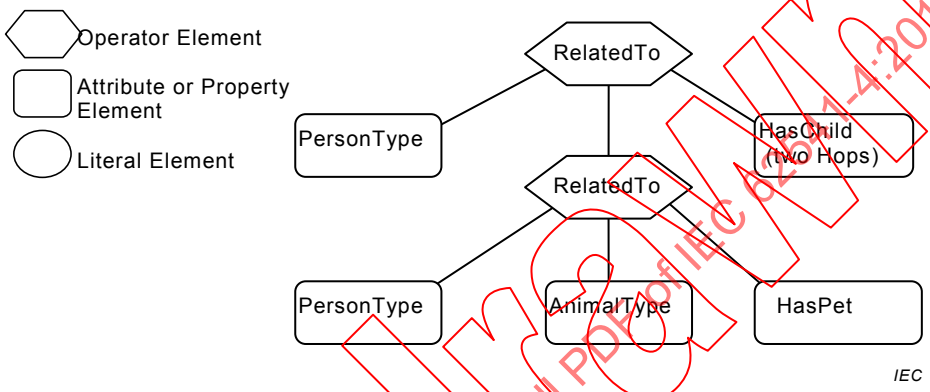


Figure B.8 – Example 4 Filter Logic Tree

Table B.13 describes the elements, operators and operands used in the example.

Table B.13 – Example 4 ContentFilter

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

The results from this query would contain the *QueryDataSets* shown in Table B.14. It is worth noting that the pig "Pig1" is referenced as a pet by Sara, but is referenced as a farm animal by Sara's parent Paul.

Table B.14 – Example 4 QueryDataSets

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

NOTE The RelativePath column and browse name (in parentheses in the *NodeId* column) are not in the *QueryDataSet* and are only shown here for clarity. The TypeDefinition NodeId would be an integer not the symbolic name that is included in the table.

B.2.8 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 B.15 describes the NodeTypeDescription parameters used in the example.

Table B.15 – Example 5 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.9.

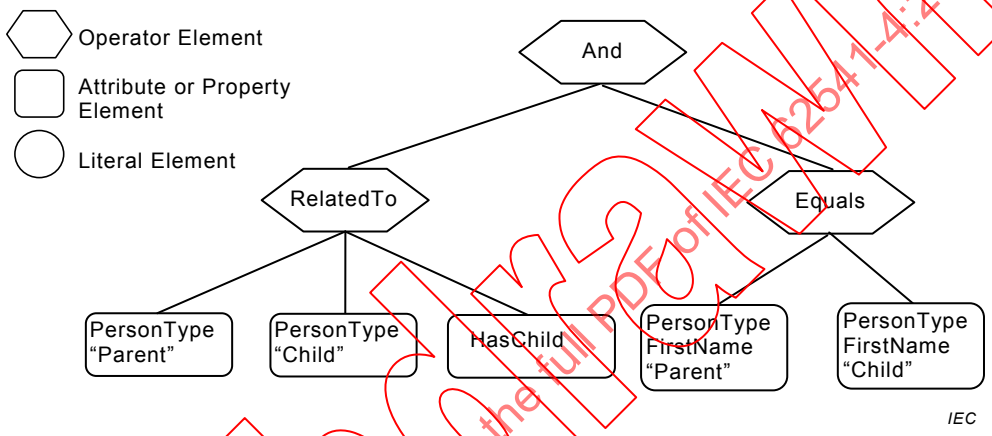


Figure B.9 – Example 5 Filter Logic Tree

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 B.16 describes the elements, operators and operands used in the example.

Table B.16 – Example 5 ContentFilter

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

The results from this query would contain the *QueryDataSets* shown in Table B.17.

Table B.17 – Example 5 QueryDataSets

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

NOTE The RelativePath column and browse name (in parentheses in the *NodeId* column) are not in the *QueryDataSet* and are only shown here for clarity. The *TypeDefinitionNodeId* would be an integer not the symbolic name that is included in the table.

B.2.9 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 B.18 describes the NodeTypeDescription parameters used in the example.

Table B.18 – Example 6 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.10.

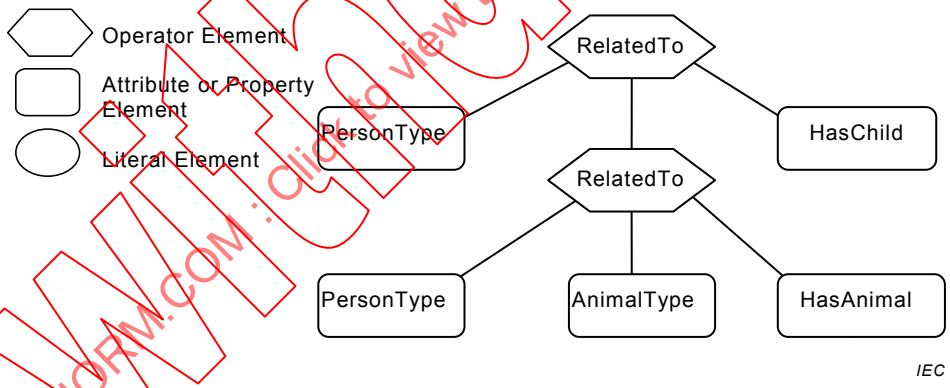


Figure B.10 – Example 6 Filter Logic Tree

Table B.19 describes the elements, operators and operands used in the example.

Table B.19 – Example 6 ContentFilter

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

The results from this query would contain the *QueryDataSets* shown in Table B.20.

Table B.20 – Example 6 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	".NodeId"	12:42 (HFamily1)
		<12:HasChild>12:PersonType<12:HasAnimal> 12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasChild>12:PersonType<12:HasAnimal>	HasFarmAnimal *ReferenceDescription*
12:48 (HFamily2)	PersonType	".NodeId"	12:48 (HFamily2)
		<12:HasChild>12:PersonType<12:HasAnimal> 12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasChild>12:PersonType<12:HasAnimal>	HasPet *ReferenceDescription*

NOTE The RelativePath 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 B.20, the value would be a *ReferenceDescription* which are described in 7.24.

Table B.21 provides an example of the same QueryDataSet as shown in Table B.20 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 B.4.

Table B.21 – Example 6 QueryDataSets without Additional Information

NodeId	TypeDefinition NodeId	Value
12:42	PersonType	12:42
		12:91
		[HasChild,TRUE,[48,HFamily2,HFamily2,PersonType]],
		[HasFarmAnimal,TRUE[91,Pig1,Pig1,PigType]
12:48	PersonType	12:54
		12:91
		[HasChild,TRUE,[54,HFamily3,HFamily3,PersonType]]
		[HasPet, TRUE,[91,Pig1,Pig1,PigType]]

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

B.2.10 Example 7

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

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

Table B.22 describes the NodeTypeDescription parameters used in the example.

Table B.22 – Example 7 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.11. Note that 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.

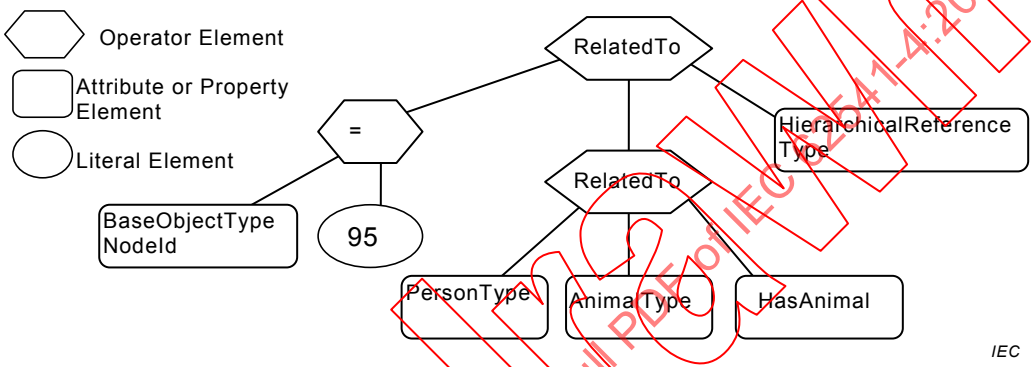


Figure B.11 – Example 7 Filter Logic Tree

Table B.23 describes the elements, operators and operands used in the example.

Table B.23 – Example 7 ContentFilter

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

The results from this *Query* would contain the *QueryDataSets* shown in Table B.24.

Table B.24 – Example 7 QueryDataSets

NodeId	TypeDefinition NodeId	RelativePath	Value
12:42 (HFamily1)	PersonType	".NodeId"	12:42 (HFamily1)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasAnimal>12:AnimalType.NodeId	NULL
		<12:HasAnimal>	HasFarmAnimal *ReferenceDescription*
12:48 (HFamily2)	PersonType	".NodeId"	12:48 (HFamily2)
		<12:HasChild>	HasChild *ReferenceDescription*
		<12:HasAnimal>12:AnimalType.NodeId	12:91 (Pig1)
		<12:HasAnimal>	HasFarmAnimal *ReferenceDescription*

NOTE The RelativePath 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 B.24, the value would be a *ReferenceDescription* which are described in 7.24.

B.2.11 Example 8

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

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

The NodeTypeDescription parameters used in the example are described in Table B.25

Table B.25 – Example 8 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.12.

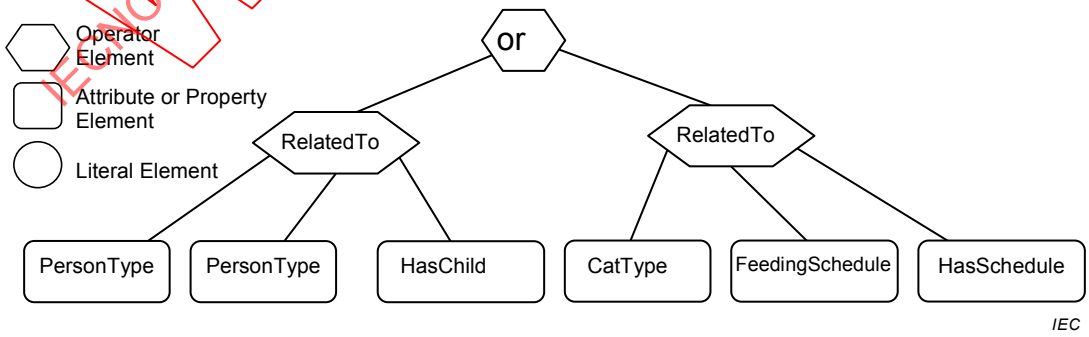


Figure B.12 – Example 8 Filter Logic Tree

Table B.26 describes the elements, operators and operands used in the example. It is worth noting that a CatType is a subtype of AnimalType.

Table B.26 – Example 8 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:HasChild, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = NodeId: 12:CatType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'

The results from this query would contain the *QueryDataSets* shown in Table B.27. If this is compared to the result set from example 2, the only difference is the omission of the Cat *Nodes*. These *Nodes* are not in the *View* and thus are not included in the result set.

Table B.27 – Example 8 QueryDataSets

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

NOTE The RelativePath column and browse name (in parentheses in the *NodeId* column) are not in the *QueryDataSet* and are only shown here for clarity. The TypeDefinition NodeId would be an integer not the symbolic name that is included in the table.

B.2.12 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 B.28 describes the NodeTypeDescription parameters used in the example.

Table B.28 – Example 9 NodeTypeDescription

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

The corresponding *ContentFilter* is illustrated in Figure B.13.

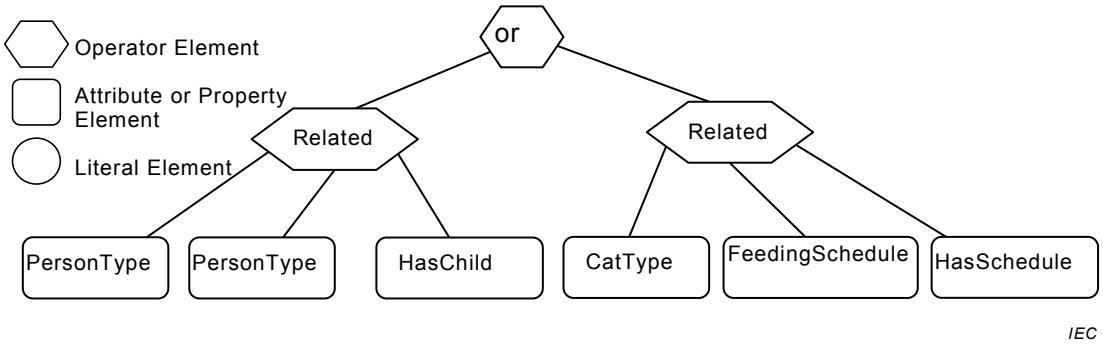


Figure B.13 – Example 9 Filter Logic Tree

Table B.29 describes the elements, operators and operands used in the example.

Table B.29 – Example 9 ContentFilter

Element[]	Operator	Operand[0]	Operand[1]	Operand[2]	Operand[3]
0	Or	ElementOperand=1	ElementOperand = 2		
1	RelatedTo	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:PersonType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:HasChild, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'
2	RelatedTo	AttributeOperand = NodeId: 12:CatType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:FeedingScheduleType, BrowsePath ".", Attribute: NodeId	AttributeOperand = NodeId: 12:HasSchedule, BrowsePath ".", Attribute: NodeId	LiteralOperand = '1'

The results from this *Query* would contain the *QueryDataSets* shown in Table B.30. 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 B.30 – Example 9 QueryDataSets

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

NOTE The RelativePath 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.

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

Architecture unifiée OPC –

Partie 4: Services

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. À cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation d'éditions référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale IEC 62541-4 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Cette deuxième édition annule et remplace la première édition parue en 2011. Cette édition constitue une révision technique.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) Mise à jour de 6.4 Redondance.
   Ajour de l'option de redondance non transparente HotAndMirrored et reformulation dans l'ensemble de la description de la redondance.

b) Clarifications pour les scénarios Publish et Reconnect.
Reformulation de différentes parties de la spécification pour s'assurer qu'aucune donnée n'est perdue pendant de courtes interruptions de la communication et que les clients peuvent toujours détecter la durée de la perte d'informations pendant une interruption de la connexion. Ajout d'un nouveau paragraphe 6.5 Rétablir les connexions qui décrit la séquence exacte de reconnexion pour les clients n'ayant plus de connexion à un serveur. Modification de l'exigence minimale relative à la file d'attente de retransmission pour l'envoi des NotificationMessages entre un intervalle d'entretien et au minimum deux fois le nombre minimal de demandes Publish par Session. Clarification concernant la valeur de données dans laquelle le bit de dépassement de capacité est établi en fonction du réglage de rejet le plus ancien. Modification de la gestion de rejet lorsque discardOldest est FALSE. La nouvelle valeur remplace l'ancienne mise en file d'attente pour FALSE. Ajout de l'exception selon laquelle le bit de dépassement de capacité n'est pas établi si la taille de la file d'attente est un.

c) Gestion des modifications de MonitoredItem dans les scénarios de courte interruption du réseau.
Ajout d'une nouvelle méthode GetMonitoredItems dans la Partie 5. Cette méthode peut être utilisée pour établir la liste des éléments surveillés dans un abonnement en cas d'échec de CreateMonitoredItems en raison d'une interruption du réseau et lorsque le client ne sait pas si la création a été réussie dans le serveur.

d) Mise à jour de 6.1.3 Déterminer si un Certificat est fiable
Révision des règles de validation du certificat.

e) Révision de la définition des paramètres semaphoreFile et isOnline dans le Service RegisterServer

f) Services ModifySubscription et ModifyMonitoredItems
Clarification précisant que les modifications s'appliquent directement et prennent effet le plus tôt possible sans toutefois dépasser une durée égale à deux fois le nouvel intervalle de temps.

g) Établissement d'une longue liste de modifications mineures permettant de lever les ambiguïtés.

Le texte de cette norme est issu des documents suivants:

CDV	Rapport de vote
65E/375/CDV	65E/403/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme.

Cette publication a été rédigée selon les Directives ISO/IEC, Partie 2.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture Unifiée OPC*, peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "http://webstore.iec.ch" dans les données relatives à la publication recherchée. A cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IMPORTANT – Le logo *"colour inside"* qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

Architecture unifiée OPC –

Partie 4: Services

1 Domaine d'application

La présente partie de l'IEC 62541 définit les *Services* de l'Architecture Unifiée OPC (OPC UA). Les *Services* décrits sont le recueil d'appels de procédures abstraites distantes (RPC – Remote Procedure Calls) qui sont mises en œuvre par les *Serveur*s OPC UA et qui sont appelées par les *Clients* OPC UA. Toutes les interactions entre *Clients* et *Serveurs* OPC UA ont lieu via ces *Services*. Les *Services* définis sont considérés comme abstraits, car pour leur mise en œuvre aucun mécanisme RPC particulier n'est spécifié dans la présente partie. L'IEC 62541-6 spécifie le(les) mécanisme(s) concret(s) de mise en correspondance (mapping) pour la mise en œuvre. Par exemple, dans l'IEC 62541-6, un des mécanismes de mise en correspondance est basé sur l'utilisation des Services Web XML. Dans ce cas, les *Services* décrits dans la présente partie apparaissent comme les méthodes de service web dans le contrat WSDL.

Tous les *Serveur*s OPC UA n'auront pas besoin de mettre en œuvre la totalité des *Services* définis. L'IEC 62541-7 définit les *Profiles* (profils) qui dictent les Services qui auront besoin d'être mis en œuvre afin d'être conformes à un *Profile* particulier.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts* (disponible en anglais seulement)

IEC TR 62541-2, *OPC Unified Architecture – Part-2: Security Model* (disponible en anglais seulement)

IEC 62541-3, *OPC Unified Architecture – Part 3: Address Space Model* (disponible en anglais seulement)

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'Information*

IEC 62541-6, *Architecture unifiée OPC – Partie 6: Correspondances*

IEC 62541-7, *Architecture unifiée OPC – Partie 7: Profils*

IEC 62541-8, *Architecture unifiée OPC – Partie 8: Accès aux données*

IEC 62541-11, *OPC Unified Architecture – Part 11: Historical Access* (disponible en anglais seulement)

IEC 62541-13, *OPC Unified Architecture – Part 13: Aggregates* (disponible en anglais seulement)

3 Termes, définitions et conventions

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans l'IEC TR 62541-1, l'IEC TR 62541-2 et l'IEC 62541-3 ainsi que les suivants s'appliquent.

3.1.1
bande morte (Deadband)
plage permise pour les variations de valeur qui ne déclencheront pas une *Notification* de changement de données

Note 1 à l'article: *Bande morte* peut s'appliquer comme un filtre lors de l'abonnement à des *Variables* et sert à empêcher que des signaux bruyants ne mettent inutilement un *Client* à jour. La présente Norme définit *AbsoluteDeadband* comme un filtre commun. L'IEC 62541-8 définit un filtre *Deadband* supplémentaire.

3.1.2
point d'extrémité (Endpoint)
adresse physique disponible sur un réseau qui permet à des *Clients* d'accéder à un ou plusieurs *Services* fournis par un *Serveur* (server)

Note 1 à l'article: Chaque *Serveur* peut avoir plusieurs *Points d'extrémité*. L'adresse d'un *Point d'extrémité* doit inclure un *HostName* (un nom d'hôte).

3.1.3
serveur passerelle (Gateway Server)
serveur qui agit comme un intermédiaire pour un ou plusieurs *Serveurs*

Note 1 à l'article: Les *Serveurs passerelles* peuvent être déployés pour limiter l'accès externe, assurer la conversion de protocole ou fournir des caractéristiques que les *Serveurs* sous-jacents ne prennent pas en charge.

3.1.4
nom d'hôte (HostName)
identificateur unique pour une machine sur un réseau

Note 1 à l'article: Cet identificateur doit être unique dans un réseau local, mais il peut toutefois être également unique au niveau global. L'identificateur peut être une adresse IP.

3.1.5
jeton de sécurité (Security Token)
identificateur pour un jeu de clés cryptographiques

Note 1 à l'article: Tous les *Security Tokens* (Jetons de sécurité) appartiennent à un contexte de sécurité qui est, dans le cas d'OPC UA, le canal sécurisé (*Secure Channel*).

3.1.6
certificat de logiciel (SoftwareCertificate)
certificat numérique pour un produit logiciel qui peut être installé sur plusieurs hôtes pour décrire les fonctions du produit logiciel

Note 1 à l'article: Des installations différentes du même produit logiciel peuvent avoir le même certificat de logiciel. Les certificats de logiciel ne sont pas pertinents pour la sécurité. Ils sont utilisés pour identifier un produit logiciel et ses caractéristiques prises en charge. Les certificats de logiciel sont décrits en 6.2.

3.2 Abréviations et symboles

API Application Programming Interface (Interface de programmation d'application)

BNF Backus-Naur Form (Notation Backus-Naur)

CA Certificate Authority (Autorité de certification)

CRL Certificate Revocation List (Liste d'annulation de certificats)

CTL Certificate Trust List (Liste de certificats de confiance)

DA Data Access (Accès aux données)

NAT Network Address Translation (Traduction d'adresse réseau)

UA Unified Architecture (Architecture Unifiée)

URI Uniform Resource Identifier (Identificateur uniforme de ressource)

URL Uniform Resource Locator (Localisateur uniforme de ressource)

3.3 Conventions pour les définitions des Services

Les *Services* OPC UA ont des paramètres qui sont acheminés entre le *Client* et le *Serveur*. Les spécifications des *Services* OPC UA utilisent des tableaux pour décrire les paramètres des *Services* comme montré au Tableau 1. Dans ce tableau, les paramètres sont organisés en paramètres de demande et paramètres de réponse.

Tableau 1 – Tableau de définition des services

Nom	Type	Description
Request (Demande)		Définit les paramètres de demande relatifs au *Service*
Nom de paramètre simple		Description de ce paramètre
Nom de paramètre construit		Description du paramètre construit
Nom de paramètre composant		Description du paramètre composant
Response (Réponse)		Définit les paramètres de réponse relatifs au *Service*

Les colonnes Nom, Type et Description donnent le nom, le type de données et la description de chaque paramètre. Tous les paramètres sont obligatoires, même si certains peuvent être inutilisés dans certaines circonstances. La colonne Description spécifie la valeur à fournir lorsqu'un paramètre est inutilisé.

Deux types de paramètres sont définis dans ces tableaux, à savoir simple et construit. Les paramètres simples ont un type de données simple, tel que *Boolean* (booléen) ou *String* (chaîne).

Les paramètres construits sont des paramètres composés de deux paramètres ou plus, qui peuvent être simples ou construits. Les noms des paramètres composants sont indentés sous le nom du paramètre construit.

Les types de données utilisés dans ces tableaux peuvent être des types de base, des types communs à plusieurs *Services* ou des types spécifiques à chaque *Service*. Les types de données de base sont définis dans l'IEC 62541-3. Les types de base utilisés dans les *Services* sont énumérés dans le Tableau 2. Les types de données qui sont communs à plusieurs *Services* sont définis à l'Article 7. Les types de données qui sont spécifiques à un *Service* sont définis dans le tableau de paramètres du *Service* en question.

Tableau 2 – Types de paramètre définis dans l'IEC 62541-3

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

Les paramètres des primitives du service Request et Indication sont représentés au Tableau 1 comme des paramètres Request. De même, les paramètres des primitives du service Response et Confirmation sont représentés au Tableau 1 comme des paramètres Response. Tous les paramètres de demande et de réponse sont inchangés et acheminés entre l'expéditeur et le récipiendaire. Par conséquent, des colonnes distinctes pour les valeurs des paramètres de demande, d'indication, de réponse et de confirmation sont inutiles et ont été intentionnellement omises afin d'améliorer la lisibilité.

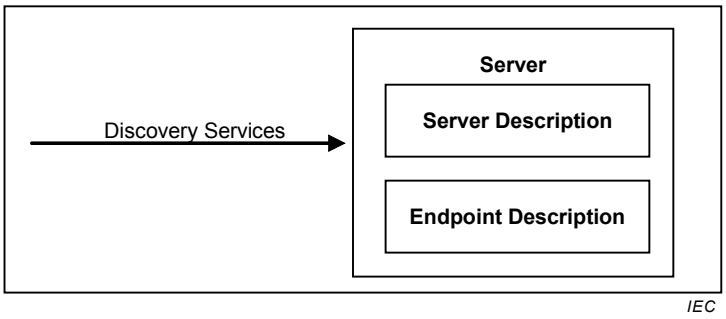
4 Vue d'ensemble

4.1 Modèle de jeux de services

Le présent article spécifie les *Services* OPC UA. Les définitions des *Services* OPC UA sont des descriptions abstraites, mais ne sont pas une spécification de mise en œuvre (implémentation). La mise en correspondance entre les descriptions abstraites et la *Communication Stack* (*Pile de communication)* associée à ces *Services* sont définies dans l'IEC 62541-6. Dans le cas d'une implémentation basée sur des services web, les *Services* OPC UA correspondent au service web et un *Service* OPC UA correspond à une opération du service web.

Ces *Services* sont organisés en *Service Sets* (Jeux de services). Chaque *Service Set* définit un jeu de *Services* connexes. L'organisation en *Service Sets* est un regroupement logique présenté dans la présente Norme, mais qui n'est pas utilisé pour l'implémentation.

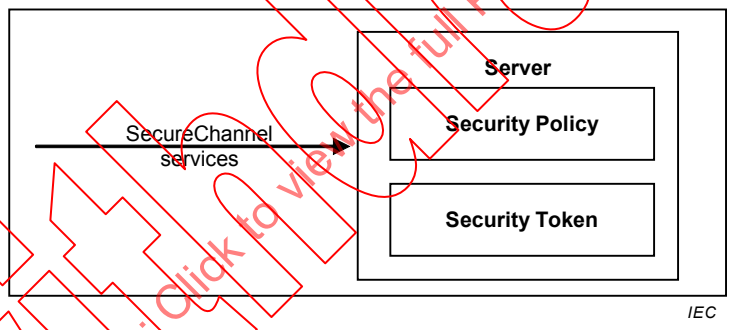
Le *Discovery Service Set* (Jeu de services de découverte), illustré à la Figure 1, définit les *Services* qui permettent à un *Client* de découvrir les *Endpoints* implémentés par un *Serveur* et de connaitre la configuration de sécurité de chacun de ces *Endpoints*.



Anglais	Français
Discovery services	Services Discovery
Server	Serveur
Server description	Description du serveur
Endpoint description	Description du point d'extrémité

Figure 1 – Jeu de services de découverte (Discovery Service Set)

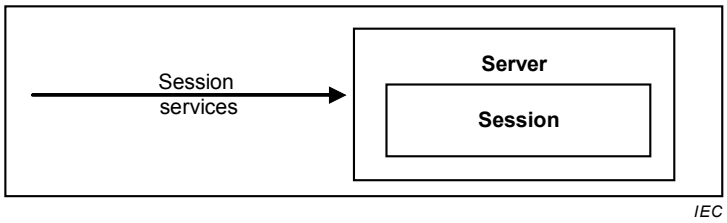
Le *SecureChannel Service Set* (Jeu de services de canal sécurisé), illustré à la Figure 2, définit les *Services* qui permettent à un *Client* d'établir un canal de communication pour assurer la *Confidentiality* (Confidentialité) et l'*Integrity* (Intégrité) des *Messages* échangés avec le *Serveur*.



Anglais	Français
SecureChannel services	Services SecureChannel
Server	Serveur
Security Policy	Politique de sécurité
Security Token	Jeton de sécurité

Figure 2 – Jeu de services de canal sécurisé (SecureChannel Service Set)

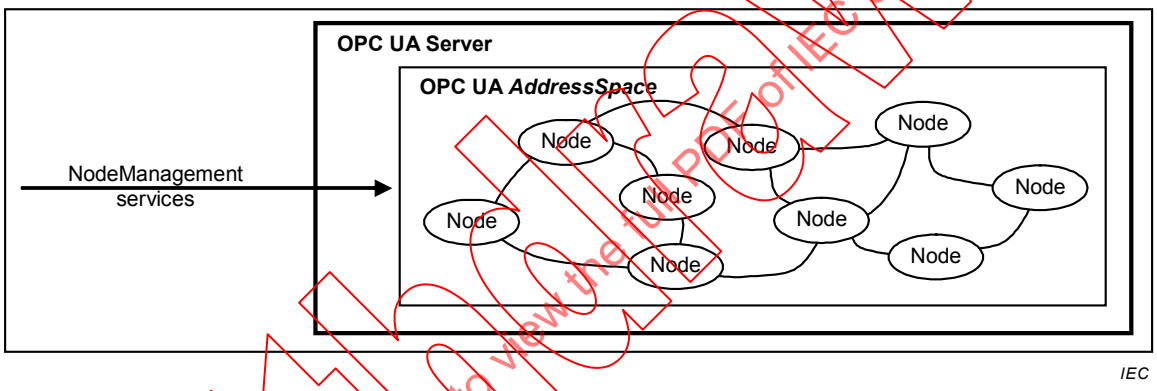
Le *Session Service Set* (Jeu de services de session), illustré à la Figure 3, définit les *Services* qui permettent au *Client* d'authentifier l'utilisateur au nom duquel il agit et de gérer des *Sessions*.



Anglais	Français
Session services	Services Session
Server	Serveur
Session	Session

Figure 3 – Jeu de services de session (Session Service Set)

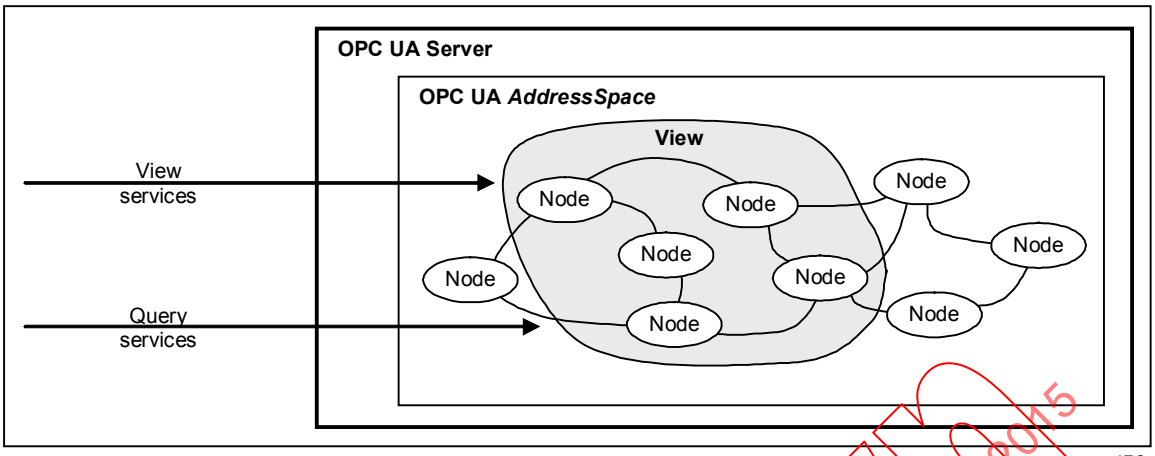
Le *NodeManagement Service Set* (Jeu de services de gestion de nœuds), illustré à la Figure 4, définit les *Services* qui permettent au *Client* d'ajouter, de modifier et de supprimer des *Nodes* (Nœuds) dans l'*AddressSpace* (Espace d'adresses).



Anglais	Français
NodeManagement services	Services NodeManagement
OPC UA Server	Serveur OPC UA
OPC UA AddressSpace	Espace d'adresses OPC UA
Node	Nœud

Figure 4 – Jeu de services de gestion de nœuds (NodeManagement Service Set)

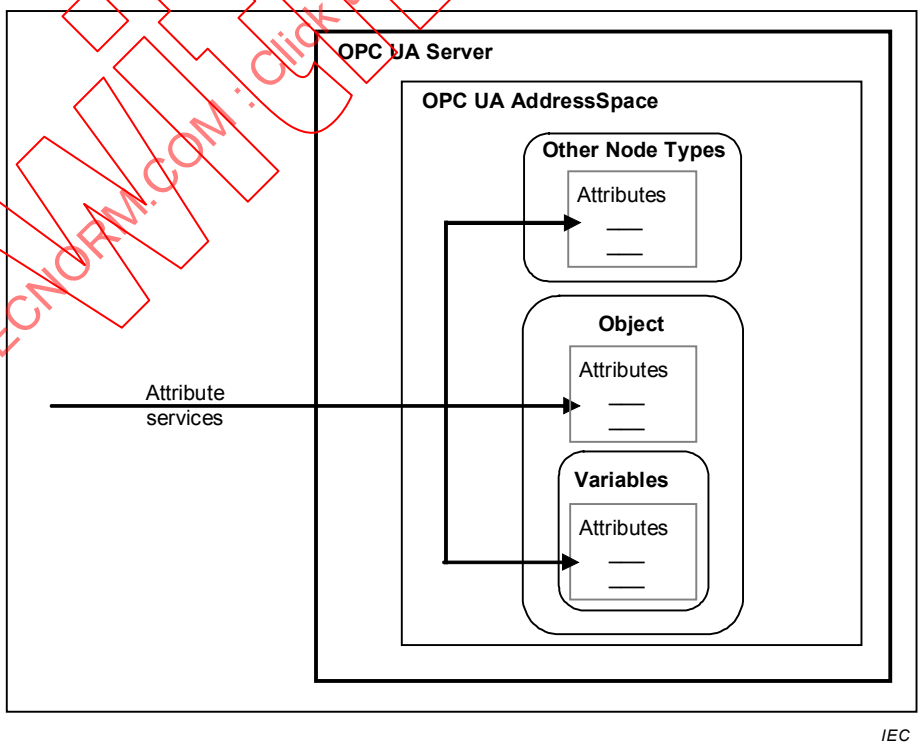
Le *View Service Set* (Jeu de services de vues), illustré à la Figure 5, définit les *Services* qui permettent aux *Clients* de parcourir l'*Espace d'Adresses* ou des sous-ensembles de l'*Espace d'Adresses* appelés *Views* (Vues). Le *Query Service Set* (Jeu de services d'interrogation) permet aux *Clients* de récupérer un sous-ensemble de données de l'*Espace d'Adresses* ou de la *Vue*.



Anglais	Français
View services	Services View (vues)
Query services	Services Query (interrogation)
OPC UA Server	Serveur OPC UA
OPC UA AddressSpace	Espace d'adresses OPC UA
Node	Nœud
View	Vue

Figure 5 – Jeu de services de vues (View Service Set)

L'*Attribute Service Set* (Jeu de services d'attributs) est illustré à la Figure 6. Il définit les *Services* qui permettent aux *Clients* de lire et écrire les *Attributs* de *Nœuds*, y compris les valeurs de l'historique. La valeur d'une *Variable* étant modélisée comme *Attribut*, ces *Services* permettent aux *Clients* de lire et écrire les valeurs des *Variables*.

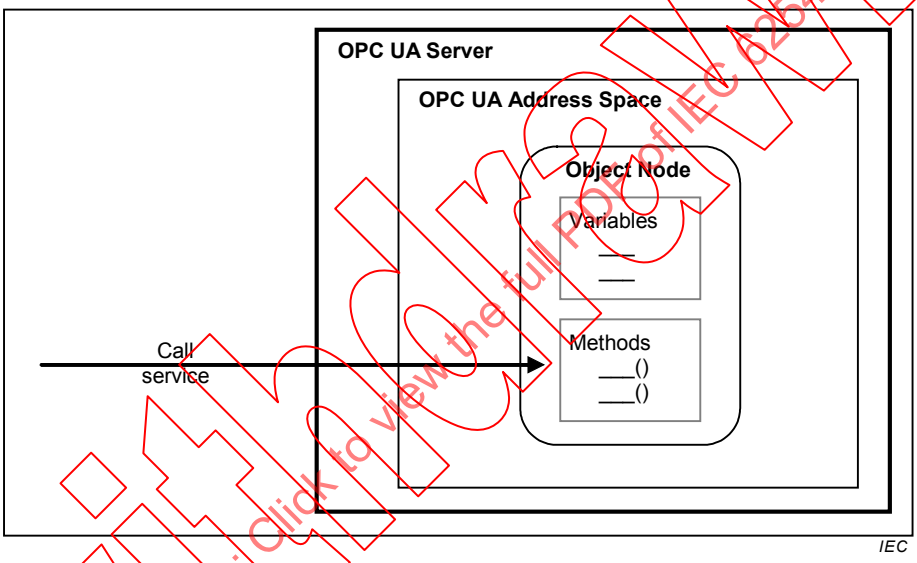


Anglais	Français
Attribute services	Services d'attributs

Anglais	Français
Other Node Types	Autres types de nœuds
OPC UA Server	Serveur OPC UA
OPC UA AddressSpace	Espace d'adresses OPC UA
Attributes	Attributs
Object	Objet
Variables	Variables

Figure 6 – Jeu de services d'attributs (Attribute Service Set)

Le *Method Service Set* (Jeu de services de méthodes) est illustré à la Figure 7. Il définit les *Services* qui permettent aux *Clients* d'appeler des méthodes. Les méthodes s'exécutent totalement lorsqu'elles sont appelées. Elles peuvent être appelées avec des paramètres d'entrée spécifiques à la méthode et peuvent retourner des paramètres de sortie spécifiques à la méthode.



Anglais	Français
Call service	Service Call
Object Node	Object Node
OPC UA Server	Serveur OPC UA
OPC UA AddressSpace	Espace d'adresses OPC UA
Methods	Méthodes
Variables	Variables

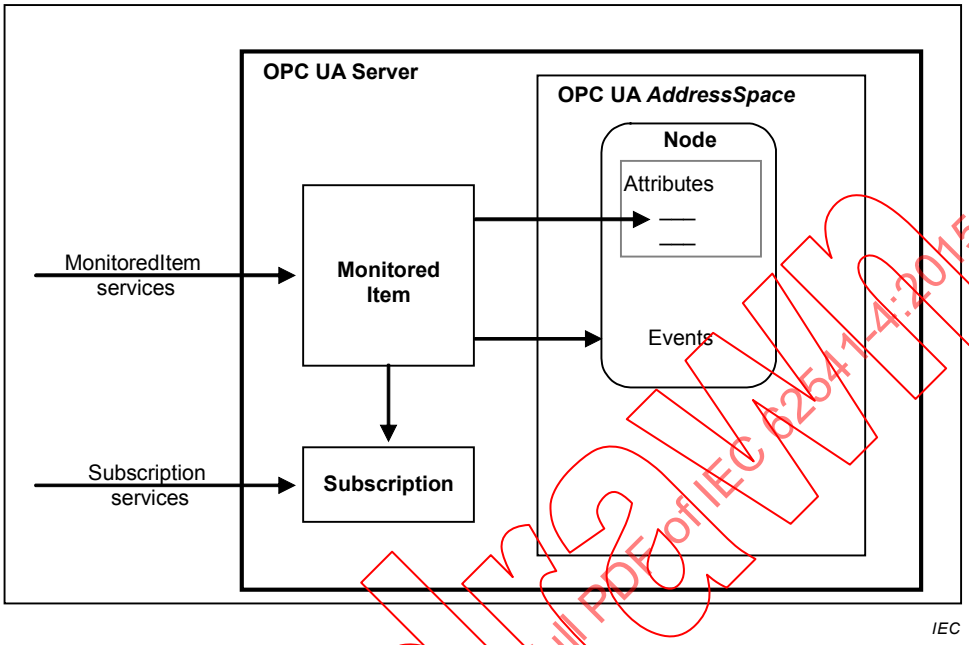
Figure 7 – Jeu de services de méthodes (Method Service Set)

Le *MonitoredItem Service Set* (Jeu de services d'éléments surveillés) et le *Subscription Service Set* (Jeu de services d'abonnements), illustrés à la Figure 8, sont utilisés ensemble pour s'abonner à des *Nœuds* de l'*Espace d'Adresses* OPC UA.

Le *MonitoredItem Service Set* définit les *Services* qui permettent aux *Clients* de créer, modifier et supprimer des *MonitoredItems* (Éléments surveillés) utilisés pour surveiller les *Attributs* afin de déceler des changements de valeur et les *Objects* afin de détecter des *Événements*.

Ces *Notifications* sont mises en file d'attente en vue de leur transfert vers le *Client* en utilisant le mécanisme de *Subscriptions* (Abonnements).

Le *Subscription Service Set* (Jeu de services d'abonnements) définit les *Services* qui permettent aux *Clients* de créer, modifier et supprimer des *Subscriptions*. Les *Subscriptions* envoient des *Notifications* générées par des *MonitoredItems* au *Client*. Les *Subscription Services* (Services d'abonnements) prévoient aussi la reprise *Client* après des *Messages* manqués et des défaillances de communications.



Anglais	Français
MonitoredItem services	Services MonitoredItem (éléments surveillés)
Subscription services	Services Subscription (abonnements)
Node	Nœud
OPC UA Server	Serveur OPC UA
OPC UA AddressSpace	Espace d'adresses OPC UA
Attributes	Attributs
Events	Événements
MonitoredItem	Élément surveillé
Subscription	Abonnement

Figure 8 – Jeux de services d'éléments surveillés et d'abonnements (MonitoredItem et Subscription Service Sets)

4.2 Procédures des services Request/Response (Demande/Réponse)

Les procédures des *Services* Request/Response décrivent le traitement des demandes reçues par le *Serveur* et des réponses renvoyées ensuite. Les procédures débutent par la présentation au *Serveur* d'un *Message* de demande de *Service* de la part du *Client* demandeur.

À la réception de la demande, le *Serveur* traite le *Message* en deux étapes. Dans la première étape, il tente de décoder et de localiser le *Service* à exécuter. La gestion des erreurs pour cette étape est spécifique à la technologie de communication utilisée et elle est décrite dans l'IEC 62541-6.

S'il réussit, il tente d'accéder à chaque opération identifiée dans la demande et exécute l'opération demandée. Pour chaque opération indiquée dans la demande, il génère un code de succès/échec distinct qu'il insère dans un *Message* de réponse positive accompagné des éventuelles données qui sont à retourner.

Pour exécuter ces opérations, le *Client* et le *Serveur* peuvent se servir de l'API de la *Pile de communication* pour construire et interpréter les *Messages* et accéder à l'opération demandée.

La mise en œuvre de la gestion de chaque demande de service ou de chaque réponse doit vérifier que chaque paramètre de service s'inscrit bien dans la plage spécifiée pour chaque paramètre.

5 Jeux de services (Service Sets)

5.1 Généralités

Le présent article définit les *Service Sets* d'OPC UA et leurs *Services*. L'Article 7 contient les définitions des paramètres communs utilisés par ces *Services*. Les exigences relatives à l'audit sont décrites en 6.2 pour tous les services.

Le *Server Profile* définit si un *Serveur* prend ou non en charge un *Service Set*, ou un *Service* inclus dans un *Service Set*. Les *Profiles* sont décrits dans l'IEC 62541-7.

5.2 En-tête de demande de services et de réponse

Chaque demande de *Service* a un *RequestHeader* (En-tête de demande) et chaque réponse relative à un *Service* a un *ResponseHeader* (En-tête de réponse).

La structure du *RequestHeader* est définie en 7.26 et contient les paramètres communs aux demandes tels qu'*authenticationToken* (Jeton d'authentification), *timestamp* (Horodatage) et *requestHandle* (Gestion de demande).

La structure du *ResponseHeader* est définie en 7.27 et contient des paramètres communs aux réponses tels que *serviceResult* (Résultat de service) et *diagnosticInfo* (Informations de diagnostic).

5.3 Résultats des services

Les résultats d'un *Service* sont communiqués à deux niveaux dans les réponses OPC UA: l'un indique le statut de l'appel de *Service* tandis que l'autre indique le statut de chaque opération demandée par le *Service*.

Les résultats de *Service* sont définis via le *StatusCode* (Code de statut) (voir 7.33).

Le statut de l'appel de *Service* est représenté par le *serviceResult* contenu dans le *ResponseHeader* (voir 7.27). Le mécanisme pour retourner ce paramètre est spécifique à la technologie de communication utilisée pour acheminer la réponse de *Service* et il est défini dans l'IEC 62541-6.

Le statut des opérations individuelles d'une demande est représenté par des *StatusCodes* individuels.

Les cas ci-après définissent l'utilisation de ces paramètres.

a) Un code bad (mauvais) est retourné dans le *serviceResult* si le *Service* lui-même a failli. Dans ce cas, un *ServiceFault* (Défaut de service) est renvoyé. Le *ServiceFault* est défini en 7.28.

b) Le code good (bon) est retourné dans le *serviceResult* si le *Service* a partiellement ou totalement réussi. Dans ce cas, d'autres paramètres de réponse sont retournés. Le *Client* doit toujours vérifier les paramètres de réponse, notamment tous les *StatusCodes* associés à chaque opération. Ces *StatusCodes* peuvent indiquer des résultats incorrects ou incertains pour une ou plusieurs opérations demandées lors de l'appel de *Service*.

Tous les *Services* avec des matrices d'opérations dans la demande doivent retourner un code bad dans le *serviceResult* si la matrice est vide.

Les *Services* définissent divers *StatusCodes* spécifiques et un *Serveur* doit utiliser ces *StatusCodes* spécifiques comme décrit dans le *Service*. Il convient qu'un *Client* puisse gérer ces *StatusCodes* spécifiques au *Service*. En outre, un *Client* doit prendre en compte d'autres *StatusCodes* communs définis au Tableau 165 et au Tableau 166. Des détails complémentaires pour la gestion de *StatusCodes* spécifiques par le *Client* peuvent être définis dans l'IEC 62541-7.

Si le *Serveur* découvre, par des mécanismes hors bande, que l'intégrité des authentifiants de l'application ou de l'utilisateur utilisés pour créer une *Session* ou un *Canal Sécurisé* a été compromise, il convient que le *Serveur* interrompe immédiatement toutes les sessions et les canaux qui utilisent ces authentifiants. Dans ce cas, il convient que le code de résultat du *Service* soit *Bad_IdentityTokenRejected* ou *Bad_CertificateUntrusted*.

L'analyse de message peut échouer en raison d'erreurs de syntaxe ou si les données du message sortent de la plage prise en charge par le récipiendaire. Lorsque cela se produit, les messages doivent être rejetés par le récipiendaire. Si le récipiendaire est un *Serveur*, il doit retourner un *ServiceFault* avec le code de résultat *Bad_DecodingError* ou *Bad_EncodingLimitsExceeded*. Si le récipiendaire est le *Client*, il convient que la *Pile de Communication* rende compte de ces erreurs à l'application *Client*.

De nombreuses applications imposent des limites à la taille des messages et/ou des éléments de données contenus dans ces messages. Par exemple, un *Serveur* peut rejeter des demandes contenant des valeurs chaînes plus longues qu'une certaine longueur. Ces limites sont typiquement établies par des administrateurs et s'appliquent à toutes les connexions entre un *Client* et un *Serveur*.

Les *Clients* qui reçoivent des défauts *Bad_EncodingLimitsExceeded* de la part du *Serveur* auront vraisemblablement à reformuler leurs demandes. L'administrateur peut avoir besoin d'augmenter les limites pour le *Client* s'il reçoit une réponse du *Serveur* comportant ce défaut.

Dans certains cas, les erreurs d'analyse sont fatales et il n'est pas possible de retourner un défaut. Par exemple, le message entrant peut dépasser la capacité du tampon du récipiendaire. Dans ces cas, ces erreurs peuvent être traitées comme un défaut de communication qui exige le rétablissement du *Canal Sécurisé* (voir 5.5).

Le *Client* et le *Serveur* réduisent les risques d'erreur fatale en échangeant leurs limites de taille de message dans le service *CreateSession*. Cela permet à chaque partie prenante d'éviter d'envoyer un message qui génère un défaut de communication. Il convient que le *Serveur* retourne un défaut *Bad_ResponseTooLarge* si un message de réponse sérialisé dépasse la taille de message spécifiée par le *Client*. De même, il convient que la *Pile de communication* (*Communication Stack*) du *Client* rapporte une erreur *Bad_RequestTooLarge* à l'application avant d'envoyer un message qui dépasse la limite du *Serveur*.

À signaler que les limites de taille de message s'appliquent seulement au corps de message et n'incluent pas les en-têtes ou les effets d'une application d'une quelconque politique de sécurité. Cela signifie qu'un corps de message qui est plus petit que le maximum spécifié peut tout de même générer une erreur fatale.

5.4 Jeu de services de découverte (Discovery Service Set)

5.4.1 Vue d'ensemble

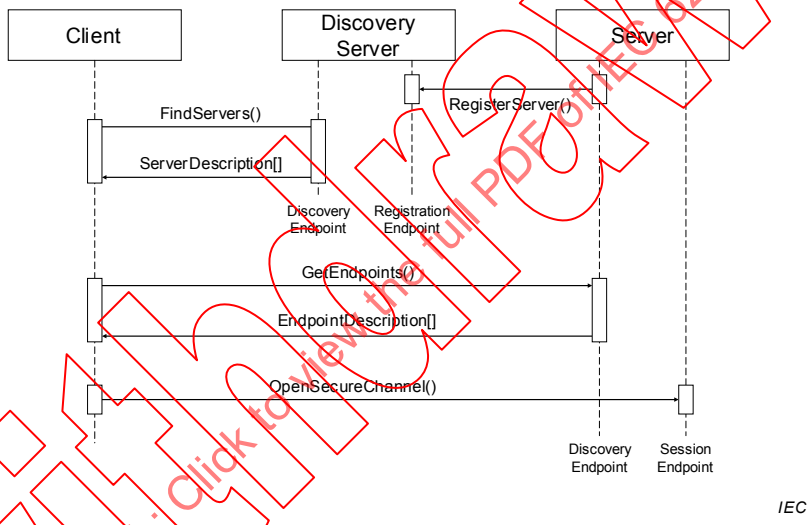
Le *Service Set* définit les *Services* utilisés pour découvrir les *Endpoints* (*Points d'extrémité)* implémentés par un *Serveur* et pour lire la configuration de sécurité pour ces *Endpoints*. Les *Discovery Services* (Services de Découverte) sont mis en œuvre par des *Serveurs* individuels

et par des *Discovery Servers* (Serveurs de découverte) spécialisés. L'IEC 62541-12[1] décrit comment les *Discovery Servers* spécialisés utilisent les *Discovery Services*.

Chaque *Serveur* doit avoir un *Discovery Endpoint* (Point d'extrémité de découverte) auquel les *Clients* accèdent sans établir une *Session*. Cet *Endpoint* peut ou peut ne pas être le même *Session Endpoint* (Point d'extrémité de session) que les *Clients* utilisent pour établir un *Canal Sécurisé.* Les clients lisent les informations de sécurité nécessaires pour établir un *Canal Sécurisé* en appelant le *GetEndpoints Service* (Service de récupération des points d'extrémité) sur le *Discovery Endpoint*.

En outre, les *Serveurs* peuvent s'enregistrer auprès d'un *Discovery Server* connu en utilisant le service *RegisterServer* (Enregistrement de serveur). Les Clients peuvent découvrir plus tard tous les éventuels *Serveurs* enregistrés en appelant le *FindServers Service* (Service Trouver des serveurs) sur le *Discovery Server*.

Le processus de découverte complet est illustré à la Figure 9.



Anglais	Français
Discovery serveur	Serveur de découverte
Server	Serveur
Discovery endpoint	Point d'extrémité de découverte
Registration endpoint	Point d'extrémité d'enregistrement
Session endpoint	Point d'extrémité de session

Figure 9 – Processus de découverte

L'URL (localisateur uniforme de ressource, *Uniform Resource Locator*) pour un *Discovery Endpoint* doit fournir toutes les informations dont le *Client* a besoin pour se connecter au *Discovery Endpoint*.

Une fois qu'un *Client* récupère les *Endpoints*, le *Client* peut sauvegarder cette information et l'utiliser pour se reconnecter directement au *Serveur* sans passer par le processus de découverte. Si le *Client* s'aperçoit qu'il ne peut pas se connecter, la configuration du *Serveur* a pu changer et il est nécessaire que le *Client* repasse par le processus de découverte.

[1] A l'étude.

Les *Discovery Endpoints* ne doivent nécessiter aucune sécurité pour les messages, mais ils peuvent requérir une sécurité de la couche transport. Dans les systèmes de production, les Administrateurs peuvent désactiver la découverte pour des raisons de sécurité et les *Clients* doivent compter sur des *EndpointDescriptions* (descriptions de points d'extrémité) placées en cache. Pour prendre en charge des systèmes ayant les *Discovery Services* désactivés, les *Clients* doivent permettre aux *Administrateurs* de mettre à jour manuellement les *EndpointDescriptions* utilisées pour se connecter à un *Serveur*. Les *Serveurs* doivent permettre aux *Administrateurs* de désactiver le *Discovery Endpoint*.

Le *Client* doit être très prudent pour utiliser les informations retournées par un *Discovery Endpoint,* car elles ne sont pas sûres. Le *Client* le fait en comparant les informations retournées par le *Discovery Endpoint* aux informations retournées dans la réponse *CreateSession*. Le *Client* doit s'assurer que:

a) L'*ApplicationUri* spécifié dans le *Serveur Certificate* est le même que l'ApplicationUri fourni dans l'*EndpointDescription*.

b) Le *Server Certificate* retourné dans la réponse *CreateSession* est le même que le *Certificate* utilisé pour créer le *Canal Sécurisé.*

c) Les *EndpointDescriptions* retournées par le *Discovery Endpoint* sont les mêmes que les *EndpointDescriptions* retournées dans la réponse *CreateSession*.

Si le *Client* détecte que l'une des exigences ci-dessus n'est pas remplie, le *Client* doit alors fermer le *Canal Sécurisé* et émettre un rapport d'erreur.

Le *Client* doit vérifier que le *HostName* spécifié dans le *Server Certificate* est le même que le *HostName* contenu dans l'*endpointUri* fourni dans l'*EndpointDescription* retournée par *CreateSession*. Si ce n'est pas le cas, le *Client* doit rapporter la différence et peut fermer le *Canal Sécurisé*. Les *Serveurs* doivent ajouter tous les *HostNames* possibles tels que MyHost et MyHost.Local dans le *Server Certificate*. Ceci comprend les adresses IP de l'hôte ou le *HostName* présenté par un routeur NAT utilisé pour se connecter au *Serveur*.

5.4.2 Trouver des serveurs (FindServers)

5.4.2.1 Description

Ce *Service* retourne les *Serveurs* connus auprès d'un *Serveur* ou d'un *Discovery Server*. Le comportement des *Discovery Servers* est décrit en détail dans l'IEC 62541-12[2].

Le *Client* peut réduire le nombre de résultats retournés en spécifiant des critères de filtrage. Un *Discovery Server* retourne une liste vide si aucun *Serveur* ne correspond aux critères spécifiés par le client. Les critères de filtrage pris en charge par ce *Service* sont décrits en 5.4.2.2.

Chaque *Serveur* doit fournir un *Discovery Endpoint* qui prend en charge ce *Service*. Le *Serveur* doit toujours retourner un enregistrement qui le décrit, cependant, dans certains cas, plusieurs enregistrements peuvent être retournés. Les *Gateway Servers* (serveurs passerelles) doivent retourner un enregistrement pour chaque *Serveur* auquel ils accordent l'accès et en plus de manière facultative un enregistrement qui permet au *Gateway Server* (serveur passerelle) d'être accessible comme un *Serveur* OPC UA ordinaire. Les *Serveurs* redondants non transparents doivent fournir un enregistrement pour chaque *Serveur* dans le jeu redondant.

Chaque *Serveur* doit avoir un identificateur globalement unique appelé *ServerUri*. Il convient que cet identificateur soit un nom de domaine totalement qualifié; cependant, cela peut être un GUID ou construct similaire qui assure l'unicité au niveau global. Le *ServerUri* retourné par ce *Service* doit avoir la valeur qui apparaît en indice 0 de la propriété *ServerArray* (matrice de

[2] A l'étude.

serveurs) (voir l'IEC 62541-5). Le *ServerUri* est retourné comme le champ *applicationUri* dans l'*ApplicationDescription* (voir 7.1)

Chaque *Serveur* doit aussi avoir un identificateur interprétable par l'utilisateur appelé *ServerName* (nom de serveur) qui au niveau global n'est pas nécessairement unique. Cet identificateur peut être disponible dans plusieurs paramètres de lieu.

Un *Serveur* peut avoir plusieurs *HostNames*. C'est pourquoi le *Client* doit passer à ce *Service* l'URL qu'il a utilisé pour se connecter à l'*Endpoint*. L'implémentation de ce *Service* doit utiliser cette information pour retourner des réponses qui sont accessibles au *Client* via l'URL fourni.

Ce *Service* ne doit nécessiter aucune sécurité pour les messages, mais il peut requérir une sécurité de la couche transport.

Certains *Serveurs* peuvent être accessibles via un *Gateway Server* et doivent avoir une valeur spécifiée pour *gatewayServerUri* (URI de serveur passerelle) dans leur *ApplicationDescription* (voir 7.1). Les *discoveryUrls* fournis dans *ApplicationDescription* doivent appartenir au *Gateway Server.* Certains *Discovery Servers* peuvent retourner plusieurs enregistrements pour le même *Serveur* si le *Serveur* en question peut être accessible par plusieurs chemins.

Ce *Service* peut être utilisé sans sécurité et il est donc vulnérable aux attaques par déni de service (DOS, *Denial Of Service*). Il convient qu'un *Serveur* réduise au maximum les traitements requis pour envoyer une réponse via ce *Service*. Cela peut être fait en préparant le résultat à l'avance. Il convient que le *Serveur* ajoute aussi un faible retard avant de commencer le traitement d'une demande s'il y a beaucoup de demandes.

Le *DiscoveryUrl* retourné par ce *Service* est ambigu s'il y a plusieurs *TransportProfiles* (par exemple, codage UA XML ou binaire UA) associés au schéma d'URL. Il convient que les *Clients* qui prennent en charge plusieurs *TransportProfiles* tentent d'utiliser des *TransportProfiles* de remplacement si le premier choix échoue.

5.4.2.2 Paramètres

Le Tableau 3 définit les paramètres pour le *Service*.

Tableau 3 – Paramètres du service FindServers

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. L'*authenticationToken* (jeton d'authentification) est toujours omis. S'il est fourni, l'*authenticationToken* doit être ignoré. Le type *RequestHeader* est défini en 7.26.
endpointUrl	String	L'adresse réseau que le *Client* a utilisée pour accéder au *Discovery Endpoint*. Le *Serveur* utilise cette information pour des diagnostics et pour déterminer les URL à retourner dans la réponse. Il convient que le *Serveur* retourne une URL par défaut adéquat s'il ne reconnaît pas le *HostName* dans l'URL.
localeIds []	LocaleId	Liste des paramètres de lieu à utiliser. Il convient que le serveur retourne le *ServerName* en utilisant un des paramètres de lieu spécifiés. Si le serveur prend en charge plus d'un des paramètres de lieu demandés, il doit utiliser celui qui apparaît le premier dans la liste. Si le serveur ne prend en charge aucun des paramètres de lieu demandés, il choisit un paramètre de lieu adéquat par défaut. Le serveur choisit un paramètre de lieu adéquat par défaut si cette liste est vide.
serverUris []	String	Liste des serveurs à retourner. Tous les serveurs connus sont retournés si la liste est vide. Un serverUri correspond à l'*applicationUri* de l'*ApplicationDescription* défini en 7.1.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses. Le type *ResponseHeader* est défini en 7.27.
servers []	ApplicationDescription	Liste des *Serveurs* qui répondent aux critères spécifiés dans la demande. Cette liste est vide si aucun serveur ne répond aux critères. Le type *ApplicationDescription* est défini en 7.1.

5.4.2.3 Résultats des services

Les *StatusCodes* communs sont définis au Tableau 165.

5.4.3 Déterminer les points d'extrémité (GetEndpoints)

5.4.3.1 Description

Ce *Service* retourne les *Endpoints* pris en charge par un *Serveur* et la totalité des informations de configuration nécessaires pour établir un *Canal Sécurisé* et une *Session*.

Cela ne doit nécessiter aucune sécurité pour les messages, mais peut requérir une sécurité de la couche transport.

Un *Client* peut réduire le nombre de résultats retournés en spécifiant des critères de filtrage sur la base de *LocaleIds* et d'URI de *Transport Profile*. Le *Serveur* retourne une liste vide si aucun *Endpoint* ne correspond aux critères spécifiés par le client. Les critères de filtrage pris en charge par *Service* sont décrits en 5.4.3.2.

Un *Serveur* peut prendre en charge plusieurs configurations de sécurité pour le même *Endpoint*. Dans cette situation, le *Serveur* doit retourner des enregistrements d'*EndpointDescription* distincts pour chaque configuration disponible. Il convient que les *Clients* traitent chacune de ces configurations comme des *Endpoints* distincts même si l'URL physique se trouve être le même.

La configuration de sécurité pour un *Endpoint* comporte quatre composants:

Server Application Instance Certificate (Certificat d'instance d'application serveur)

Message Security Mode (Mode de sécurité de message)

Security Policy (Politique de sécurité)

Supported User Identity Tokens (Jetons d'identité d'utilisateur pris en charge)

L'*ApplicationInstanceCertificate* est utilisé pour sécuriser la demande d'*OpenSecureChannel* (voir 5.5.2). Le *MessageSecurityMode* et le *SecurityPolicy* indiquent au *Client* comment sécuriser les messages envoyés via le *Canal Sécurisé*. Les *UserIdentityTokens* (jetons d'identité d'utilisateur) indiquent au client le type d'authentifiant d'utilisateur qui doit être passé au *Serveur* dans la demande *ActivateSession* (voir 5.6.3).

Si le *securityPolicyUri* est NONE et qu'aucune *UserTokenPolicy* ne nécessite de chiffrement, le *Client* doit ignorer l'*ApplicationInstanceCertificate*.

Chaque *EndpointDescription* spécifie aussi un URI pour le *Transport Profile* (profil de transport) que l'*Endpoint* prend en charge. Les *Transport Profiles* (Profils de transport) spécifient les informations telles que le format de codage de message et la version de protocole et sont définis dans l'IEC 62541-7. Les *Clients* doivent chercher les *SoftwareCertificates* (Certificats de logiciel) du *Serveur* s'ils veulent découvrir la liste complète des *Profiles* pris en compte par le *Serveur* (voir 7.30).

Les messages sont sécurisés en appliquant aux messages avant de les envoyer sur le réseau des algorithmes de chiffrement normalisés. Le jeu exact d'algorithmes utilisés dépend de la *SecurityPolicy* pour l'*Endpoint*. L'IEC 62541-7 définit des *Profiles* pour des *SecurityPolicies* communes et leur affecte un URI unique. Il est prévu que les applications connaissent les *SecurityPolicies* (Politiques de sécurité) qu'elles prennent en charge et, par conséquent, seul l'URI de profil pour la *SecurityPolicy* est spécifié dans l'*EndpointDescription*. Un *Client* ne peut pas se connecter à un *Endpoint* qui ne prend pas en charge une *SecurityPolicy* qu'il connaît.

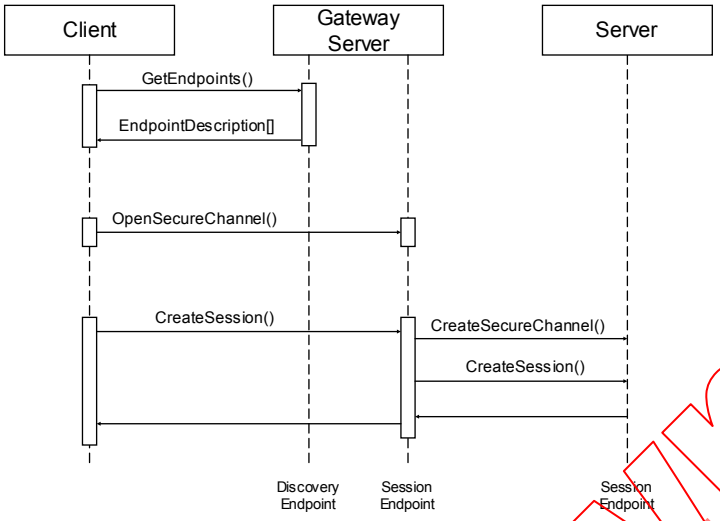
Une *EndpointDescription* peut spécifier que le mode de sécurité de message est NONE (Aucun/Rien). Cette configuration n'est pas recommandée sauf si les applications communiquent sur un réseau physiquement isolé dans lequel le risque d'intrusion est extrêmement faible. Si la sécurité de message est NONE, il est possible pour les *Clients* de pirater, délibérément ou accidentellement, des *Sessions* créées par d'autres *Clients*.

Un *Serveur* peut avoir plusieurs *HostNames*. Pour cette raison, le *Client* doit passer à ce *Service* l'URL qu'il a utilisée pour se connecter à l'*Endpoint*. L'implémentation de ce *Service* doit utiliser cette information pour retourner des réponses qui sont accessibles au *Client* via l'URL fournie.

Ce *Service* peut être utilisé sans sécurité et il est donc vulnérable aux attaques par déni de service (DOS, Denial Of Service). Il convient qu'un *Serveur* réduise au maximum la quantité de traitement nécessaire pour envoyer la réponse à ce *Service*. Cela peut être fait en préparant le résultat à l'avance. Il convient que le *Serveur* ajoute aussi un faible retard avant de commencer le traitement d'une demande s'il y a beaucoup de demandes.

Certaines des *EndpointDescriptions* retournées dans une réponse doivent spécifier les informations d'*Endpoint* pour un *Gateway Server* qui peut être utilisé pour accéder à un autre *Serveur*. Dans ce cas, le *gatewayServerUri* (l'URI de serveur passerelle) est spécifié dans l'*EndpointDescription* et tous les contrôles de sécurité utilisés pour vérifier les *Certificates* doivent utiliser le *gatewayServerUri* (voir 6.1.3) au lieu du *serverUri* (URI du serveur).

Pour se connecter à un *Serveur* via la passerelle, le *Client* doit d'abord établir un *Canal Sécurisé* avec le *Gateway Server*. Le *Client* doit appeler le service *CreateSession* et passe au *Gateway Server* le *serverUri* spécifié dans l'*EndpointDescription*. Le *Gateway Server* doit alors se connecter au *Serveur* sous-jacent pour le compte du *Client*. Le processus de connexion au *Serveur* via un *Gateway Server* est illustré à la Figure 10.



Anglais	Français
Gateway server	Serveur passerelle
Server	Serveur
Discovery endpoint	Point d'extrémité de découverte
Session endpoint	Point d'extrémité de session

Figure 10 – Utilisation d'un Gateway Server (Serveur passerelle)

5.4.3.2 Paramètres

Le Tableau 4 définit les paramètres pour le *Service*.

Tableau 4 – Paramètres du service GetEndpoints

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. L'*authenticationToken* (jeton d'authentification) est toujours omis. S'il est fourni, l'*authenticationToken* doit être ignoré. Le type *RequestHeader* est défini en 7.26.
endpointUrl	String	L'adresse réseau que le *Client* a utilisée pour accéder au *Discovery Endpoint*. Le *Serveur* utilise cette information pour des diagnostics et pour déterminer les URL à retourner dans la réponse. Il convient que le *Serveur* retourne une URL adéquate par défaut s'il ne reconnaît pas le *HostName* dans l'URL.
localeIds []	LocaleId	Liste des paramètres de lieu à utiliser. Spécifie le paramètre de lieu à utiliser pour retourner des chaînes lisibles par l'homme. Ce paramètre est décrit en 5.4.2.2.
profileUris []	String	Liste de *Transport Profile* que doivent prendre en charge les *Endpoints* retournés. L'IEC 62541-7 définit les URI pour les *Transport Profiles* Tous les *Endpoints* sont retournés si la liste est vide.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses. Le type *ResponseHeader* est défini en 7.27.
Endpoints []	EndpointDescription	Liste des *Endpoints* qui répondent aux critères spécifiés dans la demande. Cette liste est vide si aucun *Endpoint* ne répond aux critères. Le type *EndpointDescription* est défini en 7.9.

5.4.3.3 Résultats des services

Les *StatusCodes* communs sont définis au Tableau 165.

5.4.4 Enregistrer un serveur (RegisterServer)

5.4.4.1 Description

Ce *Service* enregistre un *Serveur* auprès d'un *Discovery Server* (Serveur de découverte). Ce *Service* est appelé par un *Serveur* ou par un utilitaire de configuration distinct. Les *Clients* n'utilisent pas ce *Service*.

Un *Serveur* doit établir un *Canal Sécurisé* avec le *Discovery Server* avant d'appeler ce *Service*. Le *Canal Sécurisé* est décrit en 5.5. L'*Administrateur* du *Serveur* doit fournir au *Serveur* une *EndpointDescription* pour le *Discovery Server,* cela fait partie intégrante du processus de configuration. Les *Discovery Servers* doivent rejeter les enregistrements si le *serverUri* fourni ne correspond pas à l'*applicationUri* contenue dans le *Server Certificate* utilisé pour créer le *Canal Sécurisé*.

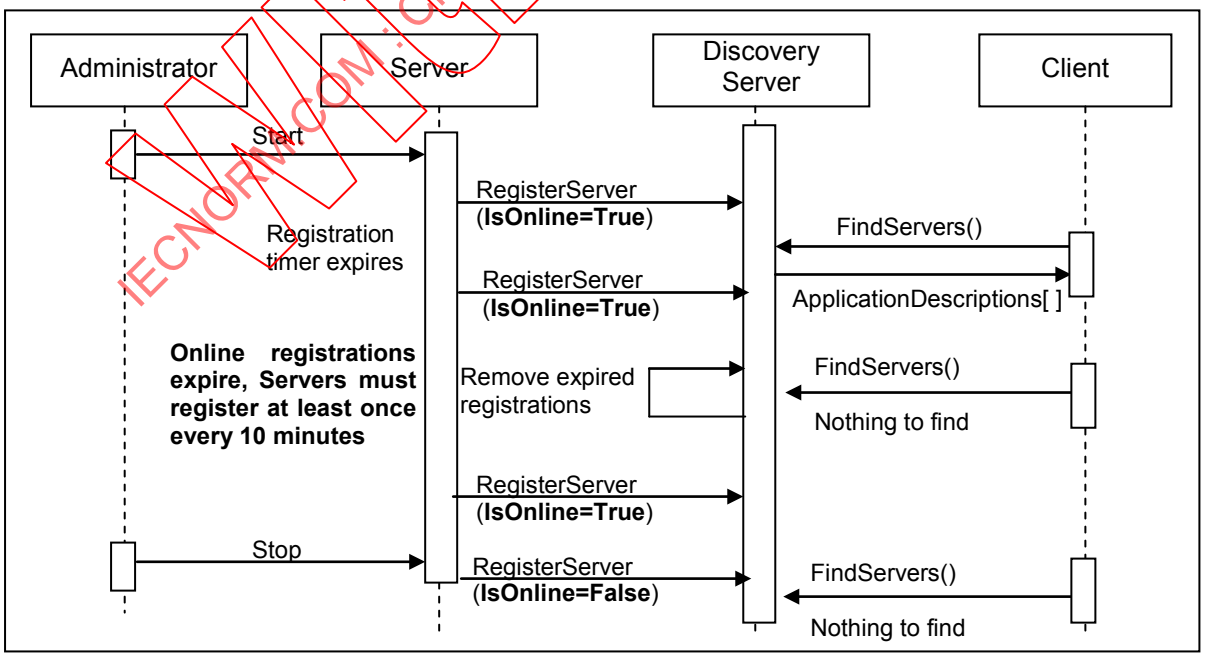
Un *Serveur* fournit uniquement son *serverUri* et les URL des *Discovery Endpoints* au *Discovery Server*. Les *Clients* doivent utiliser le service *GetEndpoints* pour rechercher directement auprès du *Serveur* les informations de configurations les plus récentes.

Le *Serveur* doit fournir un nom localisé pour lui-même dans tous paramètres de lieu qu'il prend en charge.

Les *Serveurs* doivent être à même de s'enregistrer auprès d'un *Discovery Server* tournant sur la même machine. Les mécanismes exacts dépendent de l'implémentation du *Discovery Server* et sont décrits dans l'IEC 62541-6.

Il existe deux types d'applications d'un *Serveur*: celles qui sont lancées manuellement, y compris par un démarrage par le système d'exploitation à l'amorçage et celles qui sont lancées automatiquement lorsqu'un *Client* tente de se connecter. Le processus d'enregistrement qu'un *Serveur* doit utiliser dépend de la catégorie dans laquelle il s'inscrit.

Le processus d'enregistrement pour les *Serveurs* lancés manuellement est illustré à la Figure 11.



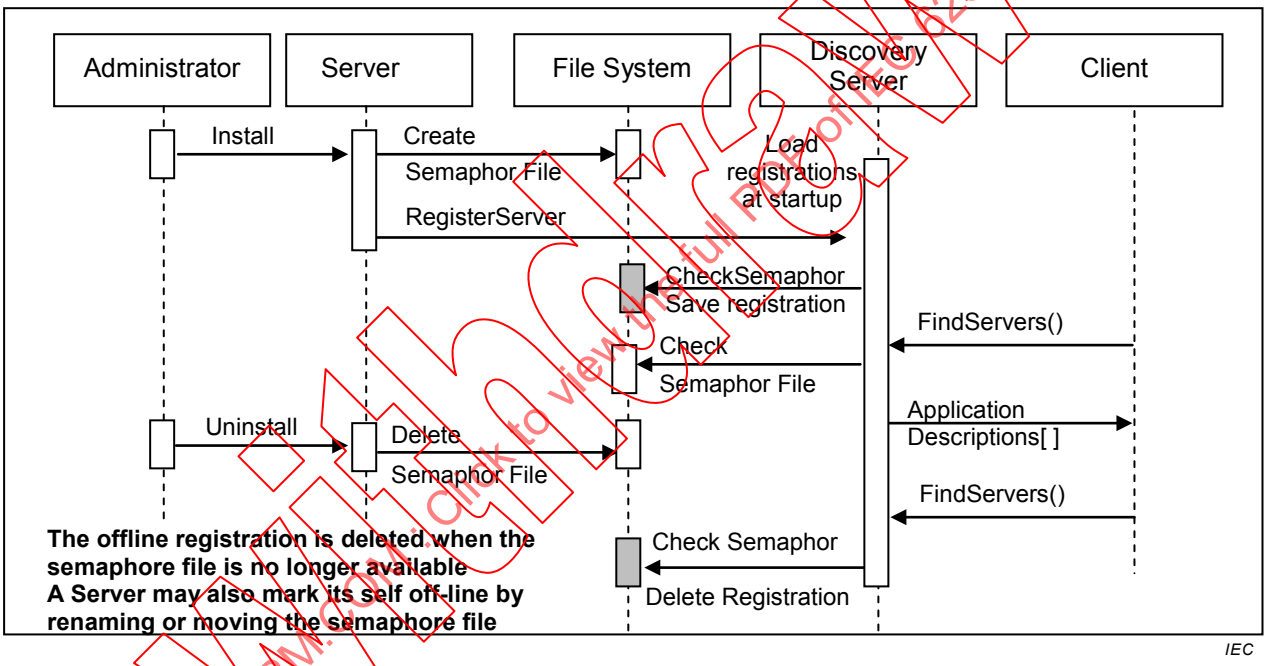
IEC

Anglais	Français
Administrator	Administrateur

Anglais	Français
Server	Serveur
Discovery server	Serveur de découverte
Registration timer expires	La minuterie d'enregistrement expire
Online registrations expire, servers must register at least once every 10 minutes	L'enregistrement en ligne expire, les serveurs doivent s'enregistrer au moins toutes les 10 minutes
Remove expired registrations	Retirer les enregistrements expirés
Start	Démarrage
Stop	Arrêt
Nothing to find	Rien à trouver

Figure 11 – Processus d'enregistrement – Serveurs lancés manuellement

Le processus d'enregistrement pour les *Serveurs* lancés automatiquement est illustré à la Figure 12.



Anglais	Français
Administrator	Administrateur
Server	Serveur
File system	Système de fichiers
Install	Installer
Create	Créer
Semaphor file	Fichier sémaphore
Load registrations at startup	Charger les enregistrements au démarrage
Save registration	Sauvegarder les enregistrements
Delete	Supprimer
Uninstall	Désinstaller
Check semaphor	Vérifier le fichier sémaphore
Delete registration	Supprimer l'enregistrement
The offline registration is deleted when the semaphor file is no longer available	L'enregistrement hors ligne est supprimé lorsque le fichier sémaphore n'est plus disponible

Anglais	Français
A server may also mark itself off-line by renaming or moving the semaphore file	Un serveur peut aussi se marquer lui-même hors ligne en renommant ou en déplaçant le fichier sémaphore

Figure 12 – Processus d'enregistrement – Serveurs lancés automatiquement

Le processus d'enregistrement est conçu pour être indépendant de la plateforme, être robuste et capable de réduire au maximum les problèmes créés par des erreurs de configuration. Pour cette raison, les *Serveurs* doivent s'enregistrer plus d'une fois.

Dans des conditions normales, les *Serveurs* lancés manuellement doivent s'enregistrer régulièrement auprès du *Discovery Server* tant qu'ils peuvent recevoir des connexions des *Clients*. Si un *Serveur* passe hors ligne, il doit s'enregistrer une fois de plus et indiquer s'il passe hors ligne. Il convient que la fréquence d'enregistrement soit configurable; cependant, la valeur maximale est de 10 minutes. Si une erreur se produit au cours de l'enregistrement (par exemple, le *Discovery Server* n'est pas en marche), le *Serveur* doit de nouveau tenter périodiquement de s'enregistrer. Il convient que la fréquence de ces tentatives commence à 1 s mais augmente progressivement jusqu'à ce que la fréquence d'enregistrement atteigne la valeur qu'elle aurait eue si aucune erreur ne s'était produite. L'approche recommandée est de doubler la période de chaque tentative jusqu'à ce que le maximum soit atteint.

Lorsqu'un *Serveur* lancé automatiquement (ou son programme d'installation) s'enregistre auprès du *Discovery Server*, il doit fournir un chemin vers un fichier sémaphore que le *Discovery Server* peut utiliser pour déterminer si le *Serveur* a été désinstallé de la machine. Le *Discovery Server* doit avoir eu accès au système de fichiers qui contient le fichier.

5.4.4.2 Paramètres

Le Tableau 5 définit les paramètres pour le *Service*.

Tableau 5 – Paramètres du service RegisterServer

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. L'*authenticationToken* (jeton d'authentification) est toujours omis. Le type *RequestHeader* est défini en 7.26.
Server	RegisteredServer	Le serveur à enregistrer. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
serverUri	String	L'identificateur globalement unique pour l'instance de *Serveur*. Le *serverUri* correspond à l'*applicationUri* de l'*ApplicationDescription* défini en 7.1.
productUri	String	L'identificateur globalement unique pour le produit *Serveur*.
serverNames []	LocalizedText	Une liste des noms descriptifs localisés pour le *Serveur*. La liste doit avoir au moins une entrée valide.
serverType	Enum ApplicationType	Le type d'application. Les valeurs d'énumération sont définies au Tableau 103. La valeur "CLIENT_1" (L'application est un *Client*.) n'est pas autorisée. Le résultat du *Service* doit être Bad_InvalidArgument dans ce cas.
gatewayServerUri	String	L'URI du *Gateway Server* associé aux *discoveryUrls*. Cette valeur est uniquement spécifiée par les *Gateway Servers* qui souhaitent enregistrer les *Serveurs* auxquels ils donnent accès. Pour les *Serveurs* qui n'agissent pas comme un *Gateway Server*, ce paramètre doit être null.
discoveryUrls []	String	Une liste de *Discovery Endpoints* pour le *Serveur*. La liste doit avoir au moins une entrée valide.
semaphoreFilePath	String	Le chemin menant au fichier sémaphore utilisé pour identifier l'instance de serveur lancé automatiquement. Les serveurs lancés manuellement n'utiliseront pas ce paramètre. Si un fichier sémaphore est fourni, le fanion isOnline est ignoré. Si un fichier sémaphore est fourni et existe, le *LocalDiscoveryServer* doit sauvegarder les informations relatives à l'enregistrement dans une unité de stockage persistante de données qu'il lit lorsque le *LocalDiscoveryServer* démarre. Si un fichier sémaphore est spécifié, mais n'existe pas, le *Discovery Server* doit retirer l'enregistrement de l'unité de stockage persistante de données. Si le Serveur a enregistré avec un semaphoreFilePath, le *DiscoveryServer* doit vérifier que ce fichier existe avant de renvoyer l'*ApplicationDescription* au client. Si le Serveur n'a pas enregistré avec un semaphoreFilePath (s'il est null ou vide) le *DiscoveryServer* ne tente pas de vérifier l'existence du fichier avant de renvoyer l'ApplicationDescription au client.
isOnline	Boolean	True (vrai) si le *Serveur* est actuellement capable d'accepter des connexions de *Clients*. Le *DiscoveryServer* doit renvoyer l'*ApplicationDescriptions* au *Client*. Il est prévu que le Serveur se réenregistre régulièrement avec le *DiscoveryServer*. False (faux) si le *Serveur* est réellement incapable d'accepter des connexions de *Clients*. Le *DiscoveryServer* ne doit PAS renvoyer les *ApplicationDescriptions* au *Client*. Ce paramètre est ignoré si un semaphoreFilePath est fourni.
Response		
ResponseHeader	ResponseHeader	Paramètres communs aux réponses. Le type *ResponseHeader* est défini en 7.27.

5.4.4.3 Résultats des services

Le Tableau 6 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 6 – Codes des résultats de services RegisterServer

Id symbolique	Description
Bad_InvalidArgument	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ServerUriInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ServerNameMissing	Aucun *ServerName* n'a été spécifié.
Bad_DiscoveryUrlMissing	Aucun DiscoveryUrl n'a été spécifié.
Bad_SempahoreFileMissing	Le fichier sémaphore spécifié n'est pas valide.

5.5 Jeu de services de canal sécurisé (SecureChannel Service Set)

5.5.1 Vue d'ensemble

Ce *Service Set* (jeu de services) définit les *Services* utilisés pour ouvrir un canal de communication qui assure la *Confidentiality* et l'*Integrity* de tous les *Messages* échangés avec le *Serveur*. Les concepts de base pour la sécurité OPC UA sont définis dans l'IEC TR 62541-2.

Les *SecureChannel Services* (services de canal sécurisé) ne sont pas comme d'autres *Services,* car ils ne sont pas implémentés directement par l'*Application OPC UA*. En revanche, ils sont fournis par la *Pile de Communication* sur laquelle est bâtie l'*Application OPC UA*. Par exemple, un *Serveur* OPC UA peut être bâti sur une pile SOAP (Simple Object Access Protocol, Protocole d'accès à objet simple) qui permet à des applications d'établir un *Canal Sécurisé* en utilisant la spécification WS Secure Conversation. Dans ces cas, l'*Application OPC UA* doit vérifier que le *Message* qu'elle a reçu était dans le contexte d'une WS Secure Conversation. L'IEC 62541-6 décrit comment les *Services SecureChannel* sont mis en œuvre.

Un *Canal Sécurisé* est une connexion logique de longue durée établie entre un seul *Client* et un seul *Serveur*. Ce canal maintient un jeu de clés connues seulement du *Client* et du *Serveur*, qui sont utilisées pour authentifier et chiffrer des *Messages* à travers le réseau. Les *Services SecureChannel* permettent au *Client* et au *Serveur* de négocier en toute sécurité les clés à utiliser.

Une *EndpointDescription* indique au Client comment établir un *Canal Sécurisé* avec un *Endpoint* donné. Un *Client* peut obtenir l'*EndpointDescription* provenant d'un *Discovery Server*, via quelque serveur non défini par l'UA, ou provenant de sa propre configuration.

Les algorithmes exacts utilisés pour authentifier et chiffrer les *Messages* sont décrits dans le champ *SecurityPolicy* de l'*EndpointDescription*. Un *Client* doit utiliser ces algorithmes lorsqu'il crée un *Canal Sécurisé*.

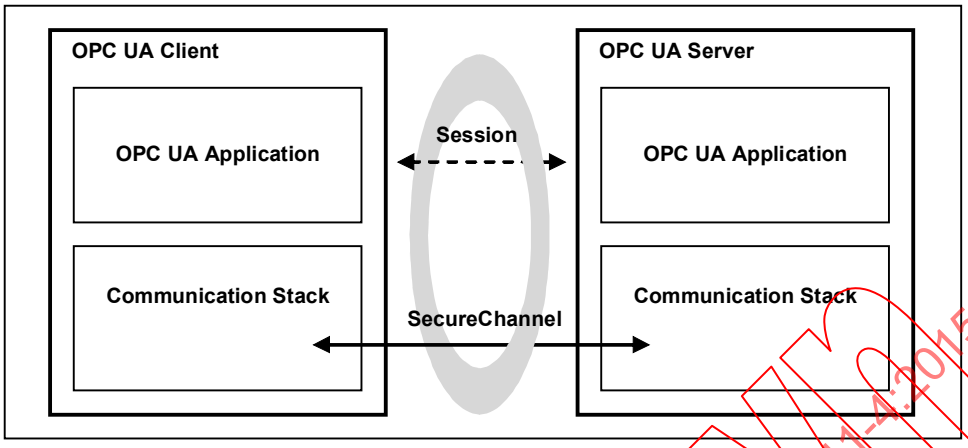
Il convient de noter que certaines *SecurityPolicies* définies dans l'IEC 62541-7 désactivent l'authentification et le chiffrement, conduisant à un *Canal Sécurisé* qui ne procure aucune sécurité.

Lorsqu'un *Client* et un *Serveur* sont en communication via un *Canal Sécurisé*, ils doivent vérifier que tous les *Messages* entrants ont été signés et chiffrés conformément aux exigences spécifiées dans l'*EndpointDescription*. Une *Application OPC UA* ne doit traiter aucun *Message* qui n'est pas conforme à ces exigences.

La relation entre le *Canal Sécurisé* et l'*Application OPC UA* dépend de la technologie d'implémentation. L'IEC 62541-6 définit toutes les exigences induites par la technologie utilisée.

La corrélation entre une *Session* d'*Application OPC UA* et le *Canal Sécurisé* est illustrée à la Figure 13. La *Pile de Communication* (*Communication Stack*) est utilisée par les *Applications OPC UA* pour échanger des *Messages*. Dans une première étape, les *Services SecureChannel* sont utilisés pour établir un *Canal Sécurisé* entre les deux *Piles de Communication* qui permet l'échange sécurisé de *Messages*. Dans une seconde étape, les

Applications OPC UA utilisent le *Session Service Set* (Jeu de services de session) pour établir une *Session* d'*Application OPC UA*.



Anglais	Français
OPC UA Client	Client OPC UA
OPC UA Application	Application OPC UA
Communication Stack	Pile de Communication
OPC UA Server	Serveur OPC UA

Figure 13 – Services SecureChannel et Session

Une fois qu'un *Client* a établi une *Session*, il peut souhaiter accéder à cette *Session* à partir d'un *Canal Sécurisé* différent. Le *Client* peut le faire en validant le nouveau *Canal Sécurisé* avec le *Service ActivateSession* décrit en 5.6.3.

Si un *Serveur* agit comme *Client* auprès d'autres *Serveurs*, ce qui est communément appelé chaînage de *Serveurs*, le *Serveur* doit être à même de maintenir le niveau de sécurité utilisateur. Cela signifie qu'il convient que l'identité de l'utilisateur soit transmise au serveur sous-jacent ou qu'elle soit mise en correspondance avec une identité d'utilisateur appropriée dans le serveur sous-jacent. Il est inacceptable d'ignorer la sécurité de niveau utilisateur. Elle est indispensable pour s'assurer que la sécurité est maintenue et qu'un utilisateur n'obtiendra pas d'informations auxquelles il convient qu'il n'ait pas accès. Autant que possible, il convient qu'un *Serveur* se substitue au *Client* d'origine en transmettant l'identité d'utilisateur du *Client* d'origine au *Serveur* sous-jacent lorsqu'il appelle le *Service ActivateSession*. Si la substitution d'identité n'est pas une option, le *Serveur* doit faire correspondre l'identité d'utilisateur du *Client* avec une nouvelle identité d'utilisateur que le *Serveur* sous-jacent reconnaît effectivement.

5.5.2 Ouvrir un canal sécurisé (OpenSecureChannel)

5.5.2.1 Description

Ce *Service* est utilisé pour ouvrir ou renouveler un *Canal Sécurisé* qui peut être utilisé pour assurer la *Confidentiality* et l'*Integrity* pour l'échange de *Messages* pendant cette *Session*. Ce *Service* exige de la *Pile de Communication* qu'elle applique les divers algorithmes de sécurité aux *Messages* à mesure qu'ils sont envoyés et reçus. Des mises en œuvre spécifiques de ce *Service* pour différentes *Piles de Communication* (*Communication Stacks*) sont décrites dans l'IEC 62541-6.

Chaque *Canal Sécurisé* a un identificateur globalement unique et est valide pour une combinaison spécifique d'instances d'applications *Client* et *Serveur*. Chaque canal contient un ou plusieurs *SecurityTokens* (jetons de sécurité) qui identifient un jeu de clés cryptographiques qui sont utilisées pour chiffrer et authentifier des *Messages*. Les

SecurityTokens ont aussi des identificateurs globalement uniques qui sont joints à chaque *Message* sécurisé avec le jeton. Cela permet à un récipiendaire autorisé de savoir comment déchiffrer et vérifier les *Messages*.

Les *SecurityTokens* ont une durée de vie finie négociée avec ce *Service*. Cependant, les différences entre les horloges système de plusieurs machines et les temps de latence réseau impliquent que des *Messages* valides peuvent arriver après l'expiration du jeton. Pour éviter que des *Messages* valides soient rejetés, il convient que les applications exécutent les actions suivantes:

a) il convient que les *Clients* demandent un nouveau *SecurityToken* après 75 % de leur durée de vie. Il convient que cela garantisse que les *Clients* reçoivent le nouveau *SecurityToken* avant que l'ancien n'expire effectivement.

b) les *Serveurs* doivent utiliser le *SecurityToken* existant pour sécuriser les *Messages* sortants jusqu'à ce que le *SecurityToken* expire ou que le *Serveur* reçoive un *Message* sécurisé avec un nouveau *SecurityToken*. Il convient que cela garantisse que les *Clients* ne rejettent pas des *Messages* sécurisés reçus avec le nouveau *SecurityToken* avant que le *Client* ne reçoive le nouveau *SecurityToken.*

c) il convient que les *Clients* acceptent des *Messages* sécurisés par un *SecurityToken* expiré pendant une durée pouvant atteindre 25 % de la durée de vie du jeton. Il convient que cela garantisse que les *Messages* envoyés par le *Serveur* avant l'expiration du jeton ne soient pas rejetés en raison des retards du réseau.

Chaque *Canal Sécurisé* existe jusqu'à ce qu'il soit explicitement fermé ou jusqu'à ce que le dernier jeton ait expiré et que la période de chevauchement ait expiré. Il convient qu'une application *Serveur* limite le nombre de *Canaux Sécurisés*. Afin d'assurer une protection contre des Clients mal intentionnés et des attaques de déni de service, le Serveur doit fermer le plus ancien *Canal Sécurisé* qui n'a pas de *Session* assignée avant d'atteindre le nombre maximal de *Canaux Sécurisés* pris en charge.

Les *Messages* de demande d'*OpenSecureChannel* et de réponse doivent être signés avec le *Certificat* de l'expéditeur. Ces *Messages* doivent toujours être chiffrés. Si la couche transport ne fournit pas de chiffrement, ces *Messages* doivent être chiffrés avec le *Certificat* du récipiendaire. Ces exigences pour *OpenSecureChannel* ne s'appliquent que si le *securityPolicyUri* n'est pas None.

Les *Certificats* utilisés dans le service *OpenSecureChannel* doivent être les *Certificats d'Instance d'Application (Application Instance Certificates).* Les *Clients* et *Serveurs* doivent vérifier que les mêmes *Certificats* ont été utilisés dans les services *CreateSession* et *ActivateSession*. Les *Certificats* ne sont pas fournis et ne doivent pas être vérifiés si le *securityPolicyUri* est None.

Si le *securityPolicyUri* n'est pas None, un *Client* doit vérifier que le *HostName* spécifié dans le *Certificat* du *Serveur* est le même que le *HostName* contenu dans l'*endpointUrl*. En cas de différence, le *Client* doit alors signaler la différence et peut choisir de ne pas ouvrir le *Canal Sécurisé*. Les *Serveurs* doivent ajouter tous les *HostNames* possibles tels que MyHost et MyHost.local dans le *Certificat* du *Serveur*. Ceci comprend les adresses IP de l'hôte ou l'*HostName* présenté par un router NAT utilisé pour établir la connexion avec le *Serveur*.

Il convient que les *Clients* soient prêts à remplacer le *HostName* retourné dans l'*EndPointDescription* par le *HostName* ou les adresses IP qu'ils utilisent pour appeler les *GetEndpoints*.

5.5.2.2 Paramètres

Le Tableau 7 définit les paramètres pour le *Service*.

Contrairement aux autres *Services*, les paramètres applicables à ce *Service* ne fournissent qu'une définition abstraite. La représentation concrète sur le réseau dépend des mises en correspondance définies dans l'IEC 62541-6.

Tableau 7 – Paramètres du service OpenSecureChannel

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. L'*authenticationToken* (jeton d'authentification) est toujours omis. Le type *RequestHeader* est défini en 7.26.
clientCertificate	ApplicationInstance Certificate	Un *Certificat* qui identifie le *Client*. La demande d'*OpenSecureChannel* doit être signée avec ce *Certificat*. Le type *ApplicationInstanceCertificate* est défini en 7.2. Si le *securityPolicyUri* est None, le *Serveur* doit ignorer l'*ApplicationInstanceCertificate*.
requestType	Enum SecurityToken RequestType	Le type de demande de *SecurityToken*. Une énumération qui doit être l'une des suivantes: ISSUE_0 crée un nouveau *SecurityToken* pour un nouveau *Canal Sécurisé*. RENEW_1 crée un nouveau *SecurityToken* pour un *Canal Sécurisé* existant.
secureChannelId	ByteString	L'identificateur du *Canal Sécurisé* auquel il convient que le nouveau jeton appartienne. Ce paramètre doit être null lors de la création d'un nouveau *Canal Sécurisé*.
securityMode	Enum MessageSecurityMode	Le type de sécurité à appliquer aux messages. Le type *MessageSecurityMode* est défini en 7.14. Il est possible qu'il faille créer un *Canal Sécurisé* même si le *securityMode* est NONE. Le comportement exact dépend de la mise en correspondance utilisée et est décrit dans l'IEC 62541-6.
securityPolicyUri	String	L'URI pour la *SecurityPolicy* à utiliser lors de la sécurisation des messages envoyés sur le *Canal Sécurisé*. Le jeu d'URI connus et les *SecurityPolicies* qui leur sont associées sont définis dans l'IEC 62541-7.
clientNonce	ByteString	Un nombre aléatoire qui ne doit être utilisé par aucune autre demande. Un nouveau *clientNonce* doit être généré chaque fois qu'un *Canal Sécurisé* est renouvelé. Ce paramètre doit avoir une longueur égale à la taille de clé utilisée pour l'algorithme de chiffrement symétrique qui est identifié par le *securityPolicyUri*.
requestedLifetime	Duration	La durée de vie demandée, en millisecondes, pour le nouveau *SecurityToken*. Il spécifie quand le *Client* pense renouveler le *Canal Sécurisé* en appelant de nouveau le *Service OpenSecureChannel*. Si un *Canal Sécurisé* n'est pas renouvelé, tous les *Messages* envoyés en utilisant les *SecurityTokens* courants doivent être rejetés par le récipiendaire.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition du type *ResponseHeader*).
securityToken	ChannelSecurityToken	Décrit le nouveau *SecurityToken* émis par le *Serveur*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
channelId	ByteString	Un identificateur unique pour le *Canal Sécurisé*. Il s'agit de l'identificateur qui doit être fourni chaque fois que le *Canal Sécurisé* est renouvelé.
tokenId	ByteString	Un identificateur unique pour un seul *SecurityToken* dans le canal. Il s'agit de l'identificateur qui doit être transmis avec chaque *Message* sécurisé avec le *SecurityToken*.
createdAt	UtcTime	Le moment où le *SecurityToken* a été créé.
revisedLifetime	Duration	La durée de vie du *SecurityToken* en millisecondes. L'heure d'expiration TUC pour le jeton peut être calculée en ajoutant la durée de vie au temps *createdAt*.
serverNonce	ByteString	Un nombre aléatoire qui ne doit être utilisé par aucune autre demande. Un nouveau *serverNonce* doit être généré chaque fois qu'un *Canal Sécurisé* est renouvelé. Ce paramètre doit avoir une longueur égale à la taille de clé utilisée pour l'algorithme de chiffrement symétrique qui est identifié par le *securityPolicyUri*.

5.5.2.3 Résultats des services

Le Tableau 8 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 8 – Codes des résultats de services OpenSecureChannel

Id symbolique	Description
Bad_SecurityChecksFailed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateTimeInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerTimeInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateHostNameInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUriInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUseNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerUseNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUntrusted	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateRevocationUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerRevocationUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateRevoked	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerRevoked	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_RequestTypeInvalid	Le type de la demande de jeton de sécurité n'est pas valide.
Bad_SecurityModeRejected	Le mode de sécurité ne satisfait pas aux exigences établies par le *Serveur*.
Bad_SecurityPolicyRejected	La politique de sécurité ne satisfait pas aux exigences établies par le *Serveur*.
Bad_SecureChannelIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NonceInvalid	Voir le Tableau 165 pour la description de ce code de résultat. Un serveur doit vérifier la longueur minimale du nonce du client et renvoyer ce statut si la longueur est inférieure à 32 octets. Une vérification des nonces dupliqués ne peut être faite que dans les appels OpenSecureChannel avec le type de demande RENEW_1.

5.5.3 Fermer un canal sécurisé (CloseSecureChannel)

5.5.3.1 Description

Ce *Service* est utilisé pour mettre fin à un *Canal Sécurisé*.

Les *Messages* de demande doivent être signés avec la clé appropriée associée au jeton courant pour le *Canal Sécurisé*.

5.5.3.2 Paramètres

Le Tableau 9 définit les paramètres pour le *Service*.

Tableau 9 – Paramètres du service CloseSecureChannel

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. Le *sessionId* (identificateur de session) est toujours omis. Le type *RequestHeader* est défini en 7.26.
secureChannelId	ByteString	L'identificateur pour le *Canal Sécurisé* à fermer.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).

5.5.3.3 Résultats des services

Le Tableau 10 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 10 – Codes des résultats de services CloseSecureChannel

Id symbolique	Description
Bad_SecureChannelIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.6 Jeu de services de session (Session Service Set)

5.6.1 Vue d'ensemble

Ce *Service Set* définit des *Services* pour l'établissement de connexions de couche application dans le contexte d'une *Session*.

5.6.2 Créer une session (CreateSession)

5.6.2.1 Description

Ce *Service* est utilisé par un *Client* OPC UA pour créer une *Session* et le *Serveur* retourne deux valeurs qui identifient de façon unique cette *Session*. La première valeur est le *sessionId* qui est utilisé pour identifier cette *Session* dans les journaux d'audit et dans l'espace d'adresses du *Serveur*. La seconde est l'*authenticationToken* qui est utilisé pour associer une demande entrante à une *Session*.

Avant d'appeler ce *Service*, le *Client* doit créer un *Canal Sécurisé* avec le *Service OpenSecureChannel* pour garantir l'*Integrity* de tous les *Messages* échangés au cours de cette *Session*. Ce *Canal Sécurisé* a un identificateur unique, que le *Serveur* doit associer à l'*authenticationToken*. Le *Serveur* peut accepter des demandes avec l'*authenticationToken* seulement si elles sont associées au *Canal Sécurisé* qui a été utilisé pour créer cette *Session*. Le *Client* peut associer un nouveau *Canal Sécurisé* à la *Session* en appelant la méthode *ActivateSession*.

Le *Canal Sécurisé* est toujours géré par la *Pile de Communication*, ce qui signifie qu'il doit fournir les API que le *Serveur* peut utiliser pour trouver des informations du *Canal Sécurisé* utilisé pour n'importe quelle demande donnée. La *Pile de Communication* doit, au minimum, fournir la *SecurityPolicy* et le *SecurityMode* utilisés par le *Canal Sécurisé*. Elle doit aussi fournir un *Canal Sécurisé Id* (identificateur de canal sécurisé) qui identifie de façon unique le *Canal Sécurisé* ou le *Client Certificate* (certificat de client) utilisé pour établir le *Canal Sécurisé*. Le *Serveur* utilise l'un de ceux-ci pour identifier le *Canal Sécurisé* utilisé pour envoyer une demande. Le paragraphe 7.29 décrit comment créer l'*authenticationToken* pour différents types de *Pile de Communication*.

En fonction de la *SecurityPolicy* et du *SecurityMode* du *Canal Sécurisé,* l'échange des *ApplicationInstanceCertificates* (Certificats d'instances d'application) et des *Nonces (*number used once, nombre utilisé une seule fois*)* peut être facultatif et les signatures peuvent être vides. Voir l'IEC 62541-7 pour la définition des *SecurityPolicies* et la gestion de ces paramètres.

Le *Serveur* retourne ses *EndpointDescriptions* dans la réponse. Les *Clients* utilisent cette information pour déterminer si, oui ou non, la liste des *EndpointDescriptions* retournées par le *Discovery Endpoint* correspond aux *Endpoints* que le *Serveur* possède. S'il y a une différence, le *Client* doit alors fermer cette *Session* et rapporter l'erreur. Le *Serveur* retourne toutes les *EndpointDescriptions* pour le *serverUri* spécifié par le *Client* dans la demande. Le *Client* vérifie uniquement les *EndpointDescriptions* ayant un *transportProfileUri* (URI de profil de transport) qui correspond au *profileUri* (URI de profil) spécifié dans la demande de *GetEndpoints* initiale. Un *Client* peut sauter cette vérification si les *EndpointDescriptions* ont été fournies par une source de confiance telle que l'*Administrateur*.

Une *Session* créée avec ce *Service* ne doit pas être utilisée jusqu'à ce que le *Client* appelle le *Service ActivateSession*, fournisse ses *SoftwareCertificates* et prouve la possession de son *Certificat d'Instance d'Application* et de tout jeton d'identité utilisateur qu'il fournit.

Le paramètre *SoftwareCertificates* dans la réponse du *Serveur* est dévalorisé de façon à réduire la taille du message pour les Applications OPC UA aux ressources limitées. Les *SoftwareCertificates* sont fournis dans l'*Espace d'Adresses* du *Serveur,* comme défini dans l'IEC 62541-5. Un *SoftwareCertificate* identifie les fonctions du *Serveur* et contient également la liste des *Profils* OPC UA pris en charge par le *Serveur*. Les *Profils* OPC UA sont définis dans l'IEC 62541-7.

Les *Certificats* supplémentaires émis par d'autres organisations peuvent être inclus pour identifier des fonctions supplémentaires du *Serveur*. Les exemples de ces *Profiles* incluent la prise en charge pour des modèles spécifiques d'informations et la prise en charge pour l'accès aux types spécifiques d'appareils.

Lorsqu'une *Session* est créée, le *Serveur* ajoute une entrée pour le *Client* dans sa *Variable SessionDiagnosticsArray*. Voir l'IEC 62541-5 pour la description de cette *Variable*.

Les *Sessions* sont créées de façon à être indépendantes de la connexion de communications sous-jacente. Par conséquent, une défaillance de la connexion de communications n'affecte pas immédiatement la *Session*. Le mécanisme exact de reprise sur une erreur de la connexion de communications sous-jacente dépend de la mise en correspondance de SecureChannel décrite dans l'IEC 62541-6.

Les *Sessions* sont automatiquement fermées par le *Serveur* si le *Client* oublie d'émettre une demande de *Service* sur la *Session* dans la période d'attente de temporisation négociée avec le *Serveur* dans la réponse à la demande de *Service CreateSession*. Le *Serveur* est ainsi protégé des défaillances du *Client* et des situations dans lesquelles la connexion sous-jacente défectueuse ne peut pas être rétablie. Les *Clients* doivent être prêts à transmettre des demandes dans le temps prévu afin d'empêcher la fermeture automatique de cette *Session*. Les *Clients* peuvent explicitement mettre fin à des *Sessions* au moyen du *Service CloseSession* (Service Fermer la Session).

Lorsqu'il est mis fin à une *Session*, toutes les demandes en cours sur cette *Session* sont abandonnées et des *StatusCodes Bad_SessionClosed* sont retournés au *Client*. De plus, le *Serveur* supprime de sa *Variable SessionDiagnosticsArray* (Matrice de diagnostic de session) l'entrée pour le *Client* et en notifie les autres éventuels *Clients* qui s'étaient abonnés à cette entrée.

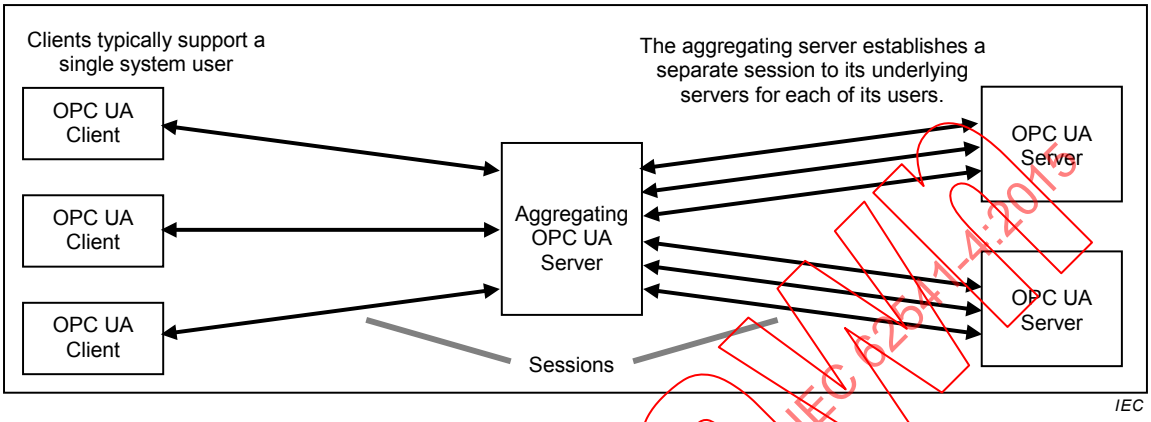
Si un *Client* invoque le *Service CloseSession*, tous les *Subscriptions* (Abonnements) associés à la *Session* sont également supprimés si le fanion *deleteSubscriptions* (Effacer les abonnements) est mis à TRUE. Si un *Serveur* met fin à une *Session* pour une autre raison quelconque, les *Subscriptions* associés à cette *Session* ne sont pas supprimés. Chaque *Subscription* a sa propre durée de vie pour se protéger des pertes de données en cas de fermeture d'une *Session*. Dans ces cas, le *Subscription* peut être réaffecté à un autre *Client* avant que sa durée de vie n'expire.

Certains *Serveurs*, tels que les *Serveurs* combinés, agissent aussi comme *Clients* pour d'autres *Serveurs*. Les *Serveurs* prennent typiquement en charge plus d'un utilisateur système, agissant comme leurs agents auprès des *Serveurs* qu'ils représentent. La sécurité pour ces *Serveurs* est prise en charge à deux niveaux.

Premièrement, chaque demande de *Service* OPC UA contient un paramètre chaîne qui est utilisé pour contenir un id d'enregistrement d'audit. Un *Client* ou tout *Serveur* opérant comme *Client*, tel qu'un *Serveur* combiné par exemple, peut créer une entrée locale de journal d'audits pour une demande qu'il présente. Ce paramètre permet au *Client* de transmettre l'identificateur pour cette entrée avec la demande. Si le *Serveur* maintient aussi un journal d'audits, il peut alors inclure cet id dans l'entrée de journal d'audits qu'il écrit. Lorsque le journal est examiné et l'entrée trouvée, l'examinateur peut la relier directement à l'entrée de journal d'audits créée par le *Client*. Cette fonction permet la traçabilité à travers les journaux d'audits dans un système. Voir l'IEC TR 62541-2 pour des informations complémentaires relatives aux audits. Un *Serveur* qui maintient un journal d'audits doit fournir les informations

dans les entrées du journal d'audits via des *Messages* d'événements définis dans la présente Norme. Le *Serveur* peut choisir de fournir uniquement les informations d'*Audit* via des *Messages* d'événements. L'*EventType Audit* est défini dans l'IEC 62541-3.

Deuxièmement, ces *Serveurs* combinés peuvent ouvrir des *Sessions* indépendantes sur les *Serveurs* sous-jacents pour chaque *Client* qui, à partir d'eux, accède aux données. La Figure 14 illustre ce concept.



Anglais	Français
Clients typically support a single system user	Les clients prennent typiquement en charge un seul utilisateur système
OPC UA Client	Client OPC UA
Aggregating OPC UA Server	Serveur combiné OPC UA
The aggregating server establishes a separate session to its underlying servers for each of its users	Le serveur combiné établit une session séparée dans ses serveurs sous-jacents pour chacun de ses utilisateurs
OPC UA Server	Serveur OPC UA

Figure 14 – Multiplexage d'utilisateurs sur une Session

5.6.2.2 Paramètres

Le Tableau 11 définit les paramètres pour le *Service*.

Tableau 11 – Paramètres du service CreateSession

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. L'*authenticationToken* (jeton d'authentification) est toujours omis. Le type *RequestHeader* est défini en 7.26.
clientDescription	Application Description	Informations qui décrivent l'application *Client*. Le type *ApplicationDescription* est défini en 7.1.
serverUri	String	Cette valeur est seulement spécifiée si l'*EndpointDescription* a un *gatewayServerUri*. Cette valeur est l'*applicationUri* issue de l'*EndpointDescription* qui est l'*applicationUri* pour le *Serveur* sous-jacent. Le type *EndpointDescription* est défini en 7.9.
endpointUrl	String	L'adresse réseau que le *Client* a utilisée pour accéder au *Session Endpoint*. Il convient que la partie *HostName* de l'URL soit l'un des *HostNames* pour l'application qui sont spécifiés dans l'*ApplicationInstanceCertificate* du *Serveur* (voir 7.2). Le *Serveur* doit soulever un événement *AuditUrlMismatchEventType* (un événement de type événement de discordance d'URL d'audit) si l'URL ne correspond pas aux *HostNames* du *Serveur*. Le type d'événement *AuditUrlMismatchEventType* est défini dans l'IEC 62541-5. Le *Serveur* utilise cette information pour des diagnostics et pour déterminer le jeu d'*EndpointDescriptions* à retourner dans la réponse.
sessionName	String	Chaîne interprétable par l'utilisateur qui identifie une *Session*. Le *Serveur* rend ce nom et le *sessionId* visibles dans son *Espace d'Adresses* à des fins de diagnostic. Il convient que le *Client* fournisse un nom qui est unique pour l'instance du *Client*. Si ce paramètre n'est pas spécifié, le *Serveur* doit attribuer une valeur.
clientNonce	ByteString	Un nombre aléatoire qu'il convient de ne jamais utiliser dans aucune autre demande. Ce nombre doit avoir une longueur minimale de 32 octets. Les profils peuvent augmenter la longueur requise. Le *Serveur* doit utiliser cette valeur pour prouver la possession de son *Certificat d'instance d'application* dans la réponse.
clientCertificate	ApplicationInstance Certificate	Le *Certificat d'instance d'application* émis au *Client*. Le type *ApplicationInstanceCertificate* est défini en 7.2. Si le *securityPolicyUri* est None, le *Serveur* doit ignorer l'*ApplicationInstanceCertificate*.
Requested SessionTimeout	Duration	Nombre maximal demandé de millisecondes pendant lesquelles il convient qu'une *Session* reste ouverte sans activité. Si le *Client* oublie d'émettre une demande de *Service* dans la limite de cet intervalle, le *Serveur* doit automatiquement mettre fin à la *Session* du *Client*.
maxResponse MessageSize	UInt32	La taille maximale, en octets, pour le corps de tout message de réponse. Il convient que le *Serveur* retourne un défaut de service *Bad_ResponseTooLarge* si un message de réponse dépasse cette limite. La valeur zéro indique que ce paramètre n'est pas utilisé. Le paragraphe 5.3 fournit plus d'informations sur l'utilisation de ce paramètre.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour le type *ResponseHeader*).
sessionId	NodeId	Un *NodeId* unique affecté par le *Serveur* à une *Session*. Cet identificateur est utilisé pour accéder aux informations de diagnostic pour une *Session* dans l'Espace d'Adresses du *Serveur*. Il est également utilisé dans les journaux d'audits et les éventuels événements qui rapportent des informations relatives à cette *Session*. Les informations de diagnostic d'une *Session* sont décrites dans l'IEC 62541-5. Les journaux d'audits et leurs événements connexes sont décrits en 6.2
authentication Token	Session AuthenticationToken	Un identificateur unique affecté par le *Serveur* à la *Session*. Cet identificateur doit être transmis dans le *RequestHeader* de chaque demande et il est utilisé avec le *SecureChannelId* pour déterminer si, oui ou non, un *Client* a accès à la *Session*. Cet identificateur ne doit pas être réutilisé d'une manière induisant le risque pour le *Client* ou le *Serveur* de le confondre avec une *Session* précédente ou existante. Le type *SessionAuthenticationToken* est décrit en 7.29.
revisedSession Timeout	Duration	Nombre maximal effectif de millisecondes pendant lesquelles une *Session* doit rester ouverte sans activité. Il convient que le *Serveur* tente d'honorer la demande du *Client* pour ce paramètre, mais il peut négocier cette valeur à la hausse ou à la baisse pour satisfaire à ses propres contraintes.
serverNonce	ByteString	Un nombre aléatoire qu'il convient de ne jamais utiliser dans aucune autre demande. Ce nombre doit avoir une longueur minimale de 32 octets. Le *Client* doit utiliser cette valeur pour prouver la possession de son *Certificat d'instance d'application* dans la demande *ActivateSession*. Cette valeur peut aussi être utilisée pour prouver la possession du *userIdentityToken* qu'il a spécifié dans la demande *ActivateSession*.
serverCertificate	ApplicationInstance Certificate	Le *Certificat d'instance d'application* émis au *Serveur*. Un *Serveur* doit prouver la possession en utilisant la clé privée pour signer le *Nonce* fourni par le *Client* dans la demande. Le *Client* doit vérifier que ce *Certificat* est le même que celui qu'il a utilisé pour créer le *Canal Sécurisé*. Le type *ApplicationInstanceCertificate* est défini en 7.2. Si le *securityPolicyUri* est NONE et qu'aucune *UserTokenPolicy* ne nécessite de

Nom	Type	Description
		chiffrement, le *Client* doit ignorer l'*ApplicationInstanceCertificate*.
serverEndpoints []	Endpoint Description	Liste des *Endpoints* que le serveur prend en charge. Le *Serveur* doit retourner un jeu d'*EndpointDescriptions* disponibles pour le *serverUri* spécifié dans la demande. Le type *EndpointDescription* est défini en 7.9. Le *Client* doit vérifier cette liste avec la liste issue d'un *Discovery Endpoint* s'il a utilisé un *Discovery Endpoint* pour rechercher des *EndpointDescriptions*. Il est recommandé que les *Serveurs* ne comprennent que l'*endpointUrl*, le *securityMode,* le *securityPolicyUri*, les *userIdentityTokens*, le *transportProfileUri* et le *securityLevel* avec tous les autres paramètres définis sur null. Seuls les paramètres recommandés doivent être vérifiés par le client.
serverSoftware Certificates []	SignedSoftware Certificate	Ce paramètre est dévalorisé et la matrice doit être vide. Les *SoftwareCertificates* sont fournis dans l'*Espace d'Adresses* du *Serveur* comme défini dans l'IEC 62541-5.
serverSignature	SignatureData	Il s'agit d'une signature générée par la clé privée associée au *serverCertificate*. Ce paramètre est calculé en accolant le *clientNonce* au *clientCertificate* et en signant la séquence d'octets obtenue. Le *SignatureAlgorithm* (algorithme de signature) doit être l'*AsymmetricSignatureAlgorithm* (de signature asymétrique) spécifié dans la *SecurityPolicy* pour l'*Endpoint*. Le type *SignatureData* est défini en 7.30.
maxRequest MessageSize	UInt32	La taille maximale, en octets, pour le corps de tout message de demande. Il convient que la *Pile de Communication* du *Client* retourne une erreur *Bad_RequestTooLarge* à l'application si un message de demande dépasse cette limite. La valeur zéro indique que ce paramètre n'est pas utilisé. Le paragraphe 5.3 fournit plus d'informations sur l'utilisation de ce paramètre.

5.6.2.3 Résultats des services

Le Tableau 12 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 12 – Codes des résultats de services CreateSession

Id symbolique	Description
Bad_SecureChannelIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NonceInvalid	Voir le Tableau 165 pour la description de ce code de résultat. Un serveur doit vérifier la longueur minimale du nonce du client et renvoyer ce statut si la longueur est inférieure à 32 octets. Une vérification des nonces dupliqués est facultative et nécessite un accès au nonce utilisé pour créer le canal sécurisé.
Bad_SecurityChecksFailed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateTimeInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerTimeInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateHostNameInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUriInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUseNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerUseNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateUntrusted	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateRevocationUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerRevocationUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateRevoked	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_CertificateIssuerRevoked	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManySessions	Le serveur a atteint le nombre maximal de sessions.
Bad_ServerUriInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.6.3 Activer une session (ActivateSession)

5.6.3.1 Description

Ce *Service* est utilisé par le *Client* pour présenter ses *SoftwareCertificates* au *Serveur* pour validation et pour spécifier l'identité de l'utilisateur associé à la *Session*. Cette demande de *Service* doit être émise par le *Client* avant qu'il n'émette une quelconque autre demande de *Service* après *CreateSession*. Si tel n'est pas le cas, le *Serveur* doit fermer la *Session*.

Chaque fois que le *Client* appelle ce *Service*, le *Client* doit prouver qu'il s'agit de la même application qui a appelé le *Service CreateSession*. Pour ce faire, le *Client* crée une signature

avec la clé privée associée au *clientCertificate* spécifié dans la demande *CreateSession*. Cette signature est créée en accolant le dernier *serverNonce* fourni par le *Serveur* au *serverCertificate* et en calculant la signature de la séquence d'octets obtenue.

Une fois utilisé, un *serverNonce* ne peut plus être réutilisé. C'est pourquoi le *Serveur* retourne un nouveau *serverNonce* chaque fois que le *Service ActivateSession* est appelé.

Lorsque le *Service ActivateSession* est appelé la première fois, le Serveur doit rejeter la demande si le *Canal Sécurisé* n'est pas le même que celui qui est associé à la demande *CreateSession*. Les appels consécutifs à *ActivateSession* peuvent être associés à des *Canaux Sécurisés* différents. Si tel est le cas, le *Serveur* doit vérifier que le *Certificat* utilisé par le *Client* pour créer le nouveau *Canal Sécurisé* est le même que le *Certificat* utilisé pour créer le *Canal Sécurisé* d'origine. En outre, le *Serveur* doit vérifier que le *Client* a fourni un *UserIdentityToken* identique au jeton réellement associé à la *Session*. Une fois que le *Serveur* a accepté le nouveau *Canal Sécurisé*, il doit rejeter les demandes envoyées par l'ancien *Canal Sécurisé*.

Le *Service ActivateSession* est utilisé pour associer une identité d'utilisateur à une *Session*. Lorsqu'un *Client* fournit une identité d'utilisateur, il doit fournir la preuve qu'il est habilité à utiliser l'identité d'utilisateur en question. Le mécanisme exact utilisé pour fournir cette preuve dépend du type d'*UserIdentityToken*. Si le jeton est un *UserNameIdentityToken*, la preuve est le *password* (Mot de passe) qui a été inclus dans le jeton. Si le jeton est un X509IdentityToken, la preuve est une signature générée avec la clé privée associée au *Certificate*. Les données à signer sont créées en accolant le dernier *serverNonce* au *serverCertificate* spécifié dans la réponse *CreateSession*. Si un jeton inclut un secret, il convient de le déchiffrer à l'aide de la clé publique contenue dans le *serverCertificate.*

Les *Clients* peuvent modifier l'identité d'un utilisateur associé à une *Session* en appelant le *Service ActivateSession*. Le *Serveur* valide les signatures fournies avec la demande puis valide la nouvelle identité d'utilisateur. En l'absence d'erreurs, le *Serveur* remplace l'identité d'utilisateur pour la *Session*. La modification de l'identité utilisateur pour une *Session* peut induire des discontinuités des *Subscriptions* actifs, car le *Serveur* peut devoir casser des connexions à un système sous-jacent et les rétablir en utilisant les nouveaux authentifiants.

Lorsqu'un *Client* fournit une liste d'identificateurs de paramètres de lieu dans la demande, il est requis de chaque identificateur de paramètre de lieu qu'il contienne la composante langue. Il peut facultativement contenir la composante <country/region> (pays/région). Lorsque le *Serveur* retourne un *LocalizedText* dans le contexte d'une *Session*, il peut aussi retourner les deux composantes langue et pays/région ou juste la composante langue comme la valeur par défaut de son identificateur de paramètre de lieu.

Lorsqu'un *Serveur* retourne une chaîne au *Client*, il détermine d'abord s'il existe des traductions pour celle-ci. S'il en existe, le *Serveur* retourne alors la chaîne dont l'identificateur de paramètre de lieu correspond exactement à l'identificateur de paramètre de lieu ayant la priorité la plus haute dans la liste fournie par le *Client*.

S'il n'existe pas de correspondances exactes, le *Serveur* ignore alors la composante <country/region> de l'identificateur de paramètre de lieu et retourne la chaîne dont la composante <language> correspond à la composante <language> de l'identificateur de paramètre de lieu ayant la priorité la plus haute dans la liste fournie par le *Client*.

S'il n'y a pas de correspondances, le *Serveur* retourne alors la chaîne qu'il a avec l'identificateur de paramètre de lieu.

Un *Gateway Server* est censé se substituer à l'utilisateur donné par le *Client* lorsqu'il se connecte au *Serveur* sous-jacent. Cela signifie qu'il doit recalculer les signatures sur l'*UserIdentityToken* en utilisant le nonce fourni par le *Serveur* sous-jacent. Le *Gateway Serveur* aura à utiliser ses propres authentifiants d'utilisateur si l'*UserIdentityToken* fourni par le *Client* ne prend pas en charge la substitution d'identité.

5.6.3.2 Paramètres

Le Tableau 13 définit les paramètres pour le *Service*.

Tableau 13 – Paramètres du service ActivateSession

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes. Le type *RequestHeader* est défini en 7.26.
clientSignature	SignatureData	Il s'agit d'une signature générée par la clé privée associée au *clientCertificate*. Le *SignatureAlgorithm* (algorithme de signature) doit être l'*AsymmetricSignatureAlgorithm* (de signature asymétrique) spécifié dans la *SecurityPolicy* pour l'*Endpoint*. Le type *SignatureData* est défini en 7.30.
clientSoftwareCertificates []	SignedSoftware Certificate	Ce sont les *SoftwareCertificates* qui ont été émis à l'application *Client*. Le *productUri* contenu dans les *SoftwareCertificates* doit correspondre au *productUri* dans l'Application*Description* utilisée par le *Client* dans les demandes *CreateSession*. Il convient d'ignorer les *Certificats* sans *productUri* correspondant. Les *Serveurs* peuvent rejeter des connexions issues des *Clients* s'ils ne sont pas satisfaits par les *SoftwareCertificates* fournis par le *Client*. Il n'est nécessaire de spécifier ce paramètre que dans la première demande *ActivateSession* après *CreateSession*. Il doit toujours être omis si la *maxRequestMessageSize* retournée par le *Serveur* dans la réponse *CreateSession* est inférieure à un mégaoctet. Le type *SignedSoftwareCertificate* est défini en 7.31
localeIds []	LocaleId	Liste des identificateurs de paramètres de lieu dans l'ordre des priorités pour les chaînes localisées. Le premier *LocaleId* dans la liste a la priorité la plus haute. Si le *Serveur* retourne une chaîne localisée au *Client*, le *Serveur* doit retourner la traduction avec la plus haute priorité qu'il peut. S'il n'a de traduction pour aucun des paramètres de lieu identifiés dans cette liste, il doit retourner la valeur chaîne qu'il a et inclure l'identificateur de paramètre de lieu avec la chaîne. Voir l'IEC 62541-3 pour plus de détails sur les identificateurs de paramètres de lieu. Si le *Client* ne réussit pas à spécifier au moins un identificateur de paramètre de lieu, le *Serveur* doit utiliser n'importe lequel en sa possession. Il n'est nécessaire de spécifier ce paramètre que pendant le premier appel au *Service ActivateSession* au cours d'une seule *Session* d'application. S'il n'est pas spécifié, le *Serveur* doit continuer d'utiliser les *localeIds* courants pour la *Session*.
userIdentityToken	Extensible Parameter UserIdentityToken	Les authentifiants de l'utilisateur associé à l'application *Client*. Le *Serveur* utilise ces authentifiants pour déterminer s'il convient ou non d'autoriser le *Client* à activer une *Session* et pour déterminer également les ressources auxquelles le *Client* a accès au cours de la *Session*. L'*UserIdentityToken* est un type de paramètre extensible défini en 7.35. L'EndpointDescription spécifie les *UserIdentityTokens* que le *Serveur* doit accepter. Les jetons d'utilisateur null ou vides doivent toujours être interprétés comme anonyme.
userTokenSignature	SignatureData	Si le *Client* a spécifié un jeton d'identité d'utilisateur qui prend en charge les signatures numériques, il doit créer une signature et la passer comme étant ce paramètre. Autrement le paramètre est omis. Le *SignatureAlgorithm* dépend du type de jeton d'identité. Le type *SignatureData* est défini en 7.30.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
serverNonce	ByteString	Un nombre aléatoire qu'il convient de ne jamais utiliser dans aucune autre demande. Ce nombre doit avoir une longueur minimale de 32 octets. Le *Client* doit utiliser cette valeur pour prouver la possession de son *Certificat d'Instance d'Application* lors de la prochaine demande d'appel à *ActivateSession*.
results []	StatusCode	Liste de résultats de validation pour les *SoftwareCertificates* (voir 7.33 pour la définition de *StatusCode*).
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic associées à des erreurs de validation de *SoftwareCertificate* (voir 7.8 pour la définition de *DiagnosticInfo*). Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.6.3.3 Résultats des services

Le Tableau 14 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 14 – Codes des résultats de services ActivateSession

Id Symbolique	Description
Bad_IdentityTokenInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_IdentityTokenRejected	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ApplicationSignatureInvalid	La signature fournie par l'application *Client* est absente ou non valide.
Bad_UserSignatureInvalid	La signature du jeton d'utilisateur est absente ou non valide.
Bad_NoValidCertificates	Le *Client* n'a pas fourni au moins un *Certificat de Logiciel* qui est valide et satisfait aux exigences relatives aux profils pour le *Serveur*.
Bad_IdentityChangeNotSupported	Le *Serveur* ne prend pas en charge la modification de l'identité d'utilisateur affectée à la session.

5.6.4 Fermer une session (CloseSession)

5.6.4.1 Description

Ce *Service* est utilisé pour mettre fin à une *Session*. Le *Serveur* entreprend les actions suivantes lorsqu'il reçoit une demande *CloseSession*:

a) Il arrête d'accepter des demandes pour cette *Session*. Toutes les demandes ultérieures reçues pour la *Session* sont rejetées.

b) Il retourne des réponses négatives avec le *StatusCode* Bad_SessionClosed à toutes les demandes qui sont actuellement en cours pour permettre le retour en temps utile de la réponse *CloseSession*. Les *Clients* sont vivement encouragés à attendre que toutes les demandes en cours se terminent avant de présenter une demande *CloseSession*.

c) Il retire l'entrée pour le *Client* dans sa *Variable SessionDiagnosticsArray*.

5.6.4.2 Paramètres

Le Tableau 15 définit les paramètres pour le *Service*.

Tableau 15 – Paramètres du service CloseSession

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
deleteSubscriptions	Boolean	Si la valeur est TRUE, le *Serveur* supprime tous les Subscriptions (Abonnements) associés à la Session. Si la valeur est FALSE, le *Serveur* conserve les *Abonnements* associés à la *Session* jusqu'à ce qu'ils expirent sur la base de leur propre durée de vie.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).

5.6.4.3 Résultats des services

Le Tableau 16 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 16 – Codes des résultats de services CloseSession

Id symbolique	Description
Bad_SessionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.6.5 Annuler (Cancel)

5.6.5.1 Description

Ce *Service* est utilisé pour annuler des demandes de Services en cours. Les demandes de services qui ont été annulées avec succès doivent répondre avec Bad_RequestCancelledByClient.

5.6.5.2 Paramètres

Le Tableau 17 définit les paramètres pour le *Service*.

Tableau 17 – Paramètres du service Cancel

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
requestHandle	IntegerId	Le *requestHandle* (descripteur de demande) affecté à une ou plusieurs demandes qu'il convient d'annuler. Toutes les demandes en cours avec le *requestHandle* correspondant doivent être annulées.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
cancelCount	UInt32	Nombre de demandes annulées.

5.6.5.3 Résultats des services

Les *StatusCodes* communs sont définis au Tableau 165.

5.7 Jeu de services de gestion de nœuds (NodeManagement Service Set)

5.7.1 Vue d'ensemble

Ce *Service Set* (Jeu de services) définit les *Services* pour ajouter et supprimer des *Nœuds de l'Espace d'Adresses* et les *Références* entre eux. Tous les *Nœuds* ajoutés continuent d'exister dans l'*Espace d'Adresses* même si le *Client* qui les a créés se déconnecte du *Serveur*.

5.7.2 Ajouter des nœuds (AddNodes)

5.7.2.1 Description

Ce *Service* est utilisé pour ajouter un ou plusieurs *Nœuds* dans la hiérarchie de l'*Espace d'Adresses*. En utilisant ce *Service*, chaque *Nœud* est ajouté comme étant le *TargetNode* d'une *HierarchicalReference* pour assurer que *l'Espace d'Adresses* est pleinement connecté et que le *Nœud* est ajouté comme enfant dans la hiérarchie de *l'Espace d'Adresses* (voir l'IEC 62541-3).

5.7.2.2 Paramètres

Le Tableau 18 définit les paramètres pour le *Service*.

Tableau 18 – Paramètres du service AddNodes

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
nodesToAdd []	AddNodesItem	Liste des *Nœuds* à ajouter. Tous les *Nœuds* sont ajoutés comme une *Référence* à un *Nœud* existant en utilisant un *ReferenceType* hiérarchique. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
parentNodeId	Expanded NodeId	*ExpandedNodeId* (identificateur de nœud étendu) du *Nœud* parent pour la *Référence*. Le type *ExpandedNodeId* est défini en 7.10.
referenceTypeId	NodeId	*NodeId* du *ReferenceType* hiérarchique à utiliser pour la *Référence* allant du *Nœud* parent au nouveau *Nœud*.
requestedNewNodeId	Expanded NodeId	*NodeId* étendu demandé par le *Client* pour le *Nœud* à ajouter. Le *serverIndex* dans le *NodeId* étendu doit être 0. Si le *Serveur* ne peut pas utiliser ce *NodeId*, il rejette ce *Nœud* et renvoie un code d'erreur approprié. Si le *Client* ne souhaite pas demander un *NodeId*, il met la valeur de ce paramètre au *NodeId* étendu null. Si le *Nœud* à ajouter est un *Nœud* de *ReferenceType*, il convient que son *NodeId* soit un identificateur numérique. Voir l'IEC 62541-3 pour une description des *NodeIds* de *ReferenceType*.
browseName	QualifiedName	Le nom de navigation du *Nœud* à ajouter.
nodeClass	NodeClass	*NodeClass* (classe de nœuds) du *Nœud* à ajouter.
nodeAttributes	Extensible Parameter NodeAttributes	Les *Attributs* qui sont spécifiques à la *NodeClass*. Le type de paramètre *NodeAttributes* est un type de paramètre extensible spécifié en 7.18. Un *Client* est autorisé à omettre les valeurs pour certains ou tous les *Attributs*. Si une valeur d'*Attribut* est omise, le *Serveur* doit utiliser les valeurs par défaut provenant du *TypeDefinitionNode*. Si un *TypeDefinitionNode* n'a pas été fourni, le *Serveur* doit choisir une valeur par défaut appropriée. Le *Serveur* peut encore ajouter un *Attribut* facultatif au *Nœud* avec une valeur par défaut appropriée même si le *Client* ne spécifie pas une valeur.
typeDefinition	Expanded NodeId	*NodeId* du *TypeDefinitionNode* pour le *Nœud* à ajouter. Ce paramètre doit être null pour toutes les *NodeClasses* autres qu'*Object* et *Variable* et, dans ce cas, il doit être fourni.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	AddNodesResult	Liste de résultats pour les *Nœuds* à ajouter. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToAdd*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour le *Nœud* à ajouter (voir 7.33 pour la définition de *StatusCode*).
addedNodeId	NodeId	*NodeId* affecté par le *Serveur* au *Nœud* ajouté. *NodeId* null si l'opération a échoué.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Nœuds* à ajouter (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToAdd*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.7.2.3 Résultats des services

Le Tableau 19 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 19 – Codes des résultats de services AddNodes

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.7.2.4 StatusCodes

Le Tableau 20 définit les valeurs du paramètre *statusCode* de niveau opération qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 20 – Codes de résultats de niveau opération des AddNodes

Id symbolique	Description
Bad_ParentNodeIdInvalid	L'identificateur de nœud parent ne renvoie pas à un nœud valide.
Bad_ReferenceTypeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceNotAllowed	La référence n'a pas pu être créée, car elle viole des contraintes imposées par le modèle de données.
Bad_NodeIdRejected	L'identificateur de nœud demandé a été rejeté, car il était invalide ou parce que le serveur n'autorise pas que des identificateurs de nœuds soient spécifiés par le client.
Bad_NodeIdExists	L'identificateur de nœud demandé est déjà utilisé par un autre nœud.
Bad_NodeClassInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_BrowseNameInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_BrowseNameDuplicated	Le nom de navigation n'est pas unique parmi les nœuds qui partagent la même relation avec le parent.
Bad_NodeAttributesInvalid	Les *Attributs* de nœud ne sont pas valides pour la classe de nœuds.
Bad_TypeDefinitionInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.

5.7.3 Ajouter des références (AddReferences)

5.7.3.1 Description

Ce *Service* est utilisé pour ajouter une ou plusieurs *Références* à un ou plusieurs *Nœuds*. La *NodeClass* est un paramètre d'entrée qui est utilisé pour valider le fait que la *Référence* à ajouter corresponde à la *NodeClass* du *TargetNode*. Ce paramètre n'est pas validé si la *Référence* renvoie à un *TargetNode* dans un *Serveur* distant.

Dans certains cas, l'ajout de nouvelles *Références* à *l'Espace d'Adresses* doit exiger que le *Serveur* ajoute de nouveaux identificateurs de *Serveur* à la *Variable ServerArray* (matrice de serveurs) du *Serveur*. C'est pourquoi les *Serveurs* distants sont identifiés par leur URI et non par leur indice de *ServerArray*. Le *Serveur* peut ainsi ajouter les URI de *Serveurs* distants à son *ServerArray*.

5.7.3.2 Paramètres

Le Tableau 21 définit les paramètres pour le *Service*.

Tableau 21 – Paramètres du service AddReferences

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
referencesToAdd []	AddReferences Item	Liste d'instances *Référence* à ajouter au *SourceNode*. La *targetNodeClass* de chaque *Référence* dans la liste doit correspondre à la *NodeClass* du *TargetNode*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
sourceNodeId	NodeId	*NodeId* au *Nœud* auquel la *Référence* est à ajouter. Le *Nœud* source doit toujours exister dans le *Serveur* pour ajouter la *Référence*. Le paramètre isForward («est direct») peut être mis à FALSE si le *Nœud* cible est sur le *Serveur* local et le *Nœud* source sur le *Serveur* distant.
referenceTypeId	NodeId	*NodeId* du *ReferenceType* qui définit la *Référence*.
isForward	Boolean	Si la valeur est TRUE, le *Serveur* crée une *Référence* directe (forward Reference). Si la valeur est FALSE, le *Serveur* crée une *Référence* inverse (inverse Reference).
targetServerUri	String	URI du *Serveur* distant. Si ce paramètre n'est pas null, il se substitue au *serverIndex* dans le *targetNodeId*.
targetNodeId	Expanded NodeId	*NodeId* étendu du *TargetNode*. Le type *ExpandedNodeId* est défini en 7.10.
targetNodeClass	NodeClass	*NodeClass* du *TargetNode*. Le *Client* doit spécifier cela, car le *TargetNode* peut ne pas être directement accessible au *Serveur*.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *Références* à ajouter (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *referencesToAdd*.
diagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les *Références* à ajouter (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *referencesToAdd*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.7.3.3 Résultats des services

Le Tableau 22 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 22 – Codes des résultats de services AddReferences

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.7.3.4 StatusCodes

Le Tableau 23 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 23 – Codes de résultats de niveau opération des AddReferences

Id symbolique	Description
Bad_SourceNodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceTypeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ServerUriInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TargetNodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeClassInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceNotAllowed	La référence n'a pas pu être créée, car elle viole des contraintes imposées par le modèle de données à ce serveur.
Bad_ReferenceLocalOnly	Le type de référence n'est pas valide pour une référence à un *Serveur* distant.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_DuplicateReferenceNotAllowed	Le type de référence entre les nœuds est déjà défini.
Bad_InvalidSelfReference	Le serveur n'autorise pas ce type d'autoréférence sur ce nœud.

5.7.4 Supprimer des nœuds (DeleteNodes)

5.7.4.1 Description

Ce *Service* est utilisé pour supprimer un ou plusieurs *Nœuds* de l'*Espace d'Adresses*.

Lorsque l'un quelconque des *Nœuds* supprimés par une invocation de ce *Service* est le *TargetNode* d'une *Référence*, ces *Références* sont laissées non résolues sur la base du paramètre *deleteTargetReferences*.

Lorsque l'un quelconque des *Nœuds* supprimés par une invocation de ce *Service* est sous surveillance, une *Notification* contenant le code de statut Bad_NodeIdUnknown est envoyée au *Client* sous surveillance indiquant que le *Nœud* a été supprimé.

5.7.4.2 Paramètres

Le Tableau 24 définit les paramètres pour le *Service*.

Tableau 24 – Paramètres du service DeleteNodes

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
nodesToDelete []	DeleteNodes Item	Liste des *Nœuds* à supprimer. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
nodeId	NodeId	*NodeId* du *Nœud* à supprimer.
deleteTargetReferences	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE supprimer les *Références* dans les *TargetNodes* qui *référencent* le *Nœud* à supprimer. FALSE supprimer seulement les *Références* dont le *Nœud* à supprimer est la source. Le *Serveur* ne peut pas garantir qu'il est capable de supprimer toutes les *Références* des *TargetNodes* si ce paramètre est TRUE.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *Nœuds* à supprimer (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToDelete.*
diagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les *Nœuds* à supprimer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToDelete.* Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.7.4.3 Résultats des services

Le Tableau 25 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 25 – Codes de résultats des services DeleteNodes

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.7.4.4 StatusCodes

Le Tableau 26 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 26 – Codes de résultats de niveau opération des DeleteNodes

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NoDeleteRights	Voir le Tableau 166 pour la description de ce code de résultat.
Uncertain_ReferenceNotDeleted	Le serveur n'était pas capable de supprimer toutes les références cibles.

5.7.5 Supprimer des références (DeleteReferences)

5.7.5.1 Description

Ce *Service* est utilisé pour supprimer une ou plusieurs *Références* d'un *Nœud*.

Lorsqu'un nombre quelconque de *Références* supprimées par une invocation de ce *Service* est contenu dans une *Vue*, la *Property ViewVersion* est mise à jour si cette *Property* est prise en charge.

Le Tableau 27 définit les paramètres du *Service*.

Tableau 27 – Paramètres du service DeleteReferences

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
referencesToDelete []	DeleteReferences Item	Liste des *Références* à supprimer. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
sourceNodeId	NodeId	*NodeId* du *Nœud* qui contient la *Référence* à supprimer.
referenceTypeId	NodeId	*NodeId* du *ReferenceType* qui définit la *Référence* à supprimer.
isForward	Boolean	Si la valeur est TRUE, le *Serveur* supprime une *Référence* directe (forward Reference). Si la valeur est FALSE, le *Serveur* supprime une *Référence* inverse (inverse Reference).
targetNodeId	ExpandedNodeId	*NodeId* du *TargetNode* de la *Référence*. Si l'indice du *Serveur* indique que le *TargetNode* est un *Nœud* distant, le *nodeId* doit contenir l'URI d'espace de noms absolu. Si le *TargetNode* est un *Nœud* local, le *nodeId* doit contenir l'indice d'espace de noms.
deleteBidirectional	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE supprimer la *Référence* spécifiée et la *Référence* opposée partant du *TargetNode*. Si le *TargetNode* est situé sur un *Serveur* distant, le *Serveur* est autorisé à supprimer la *Référence* spécifiée seulement. FALSE supprimer la *Référence* spécifiée seulement.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *Références* à supprimer (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *referencesToDelete.*
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Références* à supprimer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *referencesToDelete.* Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été trouvée dans le traitement de la demande.

5.7.5.2 Résultats des services

Le Tableau 28 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 28 – Codes de résultats des services DeleteReferences

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.7.5.3 StatusCodes

Le Tableau 29 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 29 – Codes de résultats de niveau opération des DeleteReferences

Id symbolique	Description
Bad_SourceNodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceTypeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ServerIndexInvalid	L'indice de serveur n'est pas valide.
Bad_TargetNodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NoDeleteRights	Voir le Tableau 166 pour la description de ce code de résultat.

5.8 Jeu de services pour les vues (View Service Set)

5.8.1 Vue d'ensemble

Les *Clients* utilisent les *Services* navigation du *View Service Set* (Jeu de services pour les vues) pour naviguer à travers l'*Espace d'Adresses* ou à travers une *Vue* qui est un sous-ensemble de l'*Espace d'Adresses*.

Une *Vue* est un sous-ensemble de l'*Espace d'Adresses* créé par le *Serveur*. Les futures versions de la présente Norme peuvent aussi définir des services pour créer des *Vues* définies par le *Client*. Voir l'IEC 62541-5 pour une description de l'organisation des vues dans l'*Espace d'Adresses*.

5.8.2 Parcourir (Browse)

5.8.2.1 Description

Ce *Service* est utilisé pour découvrir les *Références* d'un *Nœud* spécifié. La navigation peut en outre être limitée par l'utilisation d'une *Vue*. Ce *Service* Browse prend aussi en charge une fonction de filtrage primitive.

5.8.2.2 Paramètres

Le Tableau 30 définit les paramètres du *Service*.

Tableau 30 – Paramètres du service Browse

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
view	ViewDescription	Description de la *Vue* à parcourir (voir 7.37 pour la définition de *ViewDescription*). Une valeur vide de *ViewDescription* indique l'*Espace d'Adresses* tout entier. L'utilisation de la valeur vide de *ViewDescription* entraîne que toutes les *Références* du *nodesToBrowse* sont retournées. L'utilisation d'une autre *Vue* entraîne que seules les *Références* du *nodesToBrowse* qui sont définies pour la Vue en question seront retournées.
requestedMax ReferencesPerNode	Counter	Indique le nombre maximal de références à retourner pour chaque Nœud de départ spécifié dans la demande. La valeur 0 indique que le *Client* n'impose aucune limitation (voir 7.5 pour la définition de *Counter*).
nodesToBrowse []	BrowseDescription	Une liste de nœuds à parcourir (Browse). Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
nodeId	NodeId	*NodeId* du Nœud à parcourir. Si une *Vue* est fournie, elle doit inclure ce *Nœud*.
browseDirection	Enum BrowseDirection	Une énumération qui spécifie le sens des *Références* à suivre. Cette énumération a les valeurs suivantes: FORWARD_0 sélectionner des *Références* directes seulement. INVERSE_1 sélectionner des *Références* inverses seulement. BOTH_2 sélectionner des *Références* directes et inverses. Les *Références* retournées indiquent effectivement le sens que le *Serveur* a suivi dans le paramètre *isForward* de la *ReferenceDescription*. Les *Références* symétriques sont toujours considérées être dans le sens direct; par conséquent, le fanion isForward est toujours mis à TRUE et les *Références* symétriques ne sont pas retournées si *browseDirection* (sens de navigation) est mis à *INVERSE_1*.
referenceTypeId	NodeId	Spécifie le *NodeId* du *ReferenceType* à suivre. Seules des instances de ce *ReferenceType* ou de ses sous-types sont retournées. S'il n'est pas spécifié, toutes les *Références* sont retournées et includeSubtypes est ignoré.
includeSubtypes	Boolean	Indique s'il convient d'inclure dans la navigation les sous-types du *ReferenceType*. Si TRUE, les instances de *referenceTypeId* et de tous ses sous-types sont retournées.
nodeClassMask	UInt32	Spécifie les *NodeClasses* des *TargetNodes*. Seuls les *TargetNodes* avec les *NodeClasses* sélectionnées sont retournés. Il est affecté aux *NodeClasses* les bits suivants: **Bit** / **NodeClass** 0 Object 1 Variable 2 Method 3 ObjectType 4 VariableType 5 ReferenceType 6 DataType 7 View S'il est mis à zéro, toutes les *NodeClasses* sont retournées.
resultMask	UInt32	Spécifie les champs dans la structure *ReferenceDescription* qu'il convient de retourner. Il est affecté aux champs les bits suivants: **Bit** / **Result** 0 ReferenceType 1 IsForward 2 NodeClass 3 BrowseName 4 DisplayName 5 TypeDefinition Le type *ReferenceDescription* est défini en 7.24.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	BrowseResult	Une liste de *BrowseResults* (résultats de navigation). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du *nodesToBrowse* spécifié dans la demande. Le type *BrowseResult* est défini en 7.3.
diagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les *results* (voir 7.8 pour la définition de

Nom	Type	Description
		DiagnosticInfo). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de réponse *results*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.8.2.3 Résultats des services

Le Tableau 31 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 31 – Codes de résultats des services Browse

Id symbolique	Description
Bad_ViewIdUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewTimestampInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewParameterMismatchInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewVersionInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.8.2.4 StatusCodes

Le Tableau 32 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 32 – Codes de résultats de niveau opération de Browse

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceTypeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_BrowseDirectionInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeNotInView	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NoContinuationPoints	Voir le Tableau 166 pour la description de ce code de résultat.
Uncertain_NotAllNodesAvailable	Les résultats de la navigation (Browse) peuvent être incomplets en raison de l'indisponibilité d'un sous-système.

5.8.3 Parcourir le prochain (BrowseNext)

5.8.3.1 Description

Ce *Service* est utilisé pour demander le prochain jeu d'informations de réponse *Browse* ou *BrowseNext* qui sont trop volumineuses pour être envoyées dans une seule réponse. «Trop volumineuses» signifie dans ce contexte que le *Serveur* n'est pas capable de retourner une réponse plus volumineuse ou que le nombre de résultats à retourner dépasse le nombre maximal de résultats à retourner qui a été spécifié par le *Client* dans la demande *Browse* d'origine. Le *BrowseNext* doit être présenté dans la même *Session* qui a été utilisée pour présenter le Browse ou *BrowseNext* qui se poursuit.

5.8.3.2 Paramètres

Le Tableau 33 définit les paramètres du *Service*.

Tableau 33 – Paramètres du service BrowseNext

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
releaseContinuationPoints	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE les *continuationPoints* passés doivent être réinitialisés pour libérer des ressources dans le *Serveur*. Les points de continuation sont libérés et les résultats et les matrices de diagnosticInfos sont vides. FALSE les *continuationPoints* passés doivent être utilisés pour récupérer le prochain jeu d'informations de navigation. Un *Client* doit toujours utiliser le point de continuation retourné par une réponse *Browse* ou *BrowseNext* pour libérer les ressources pour le point de continuation dans le *Serveur*. Si le *Client* ne souhaite pas récupérer le prochain jeu d'informations de navigation, *BrowseNext* doit être appelé avec ce paramètre mis à TRUE.
continuationPoints []	Continuation Point	Une liste de valeurs opaques définies par le *Serveur* qui représentent des points de continuation. La valeur pour un point de continuation a été retournée au *Client* dans une réponse *Browse* ou *BrowseNext* précédente. Ces valeurs sont utilisées pour identifier la demande *Browse* ou *BrowseNext* précédemment traitée qui se poursuit et le point dans le jeu de résultats à partir duquel il faut que la réponse de navigation continue. Les Clients peuvent mélanger les points de continuation provenant de réponses Browse ou BrowseNext différentes. Le type *ContinuationPoint* est décrit en 7.6.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	BrowseResult	Une liste de références qui satisfont aux critères spécifiés dans la demande *Browse* d'origine. La taille et l'ordre de cette liste correspondent à la taille et à l'ordre du paramètre de demande *continuationPoints*. Le type *BrowseResult* est défini en 7.3.
diagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les *results* (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de réponse *results*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.8.3.3 Résultats des services

Le Tableau 34 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 34 – Codes de résultats des services BrowseNext

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.8.3.4 StatusCodes

Le Tableau 35 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 35 – Codes de résultats de niveau opération de BrowseNext

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ReferenceTypeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_BrowseDirectionInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeNotInView	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ContinuationPointInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.8.4 Traduire les chemins de navigation en identifiants de nœuds (TranslateBrowsePathsToNodeIds)

5.8.4.1 Description

Ce *Service* est utilisé pour demander au *Serveur* de traduire un ou plusieurs chemins de navigation en des *NodeIds*. Chaque chemin de navigation est composé d'un *Nœud* de départ et d'un *RelativePath* (chemin relatif). Le *Nœud* de départ spécifié identifie le *Nœud* sur lequel le *RelativePath* est basé. Le *RelativePath* contient une séquence de *ReferenceTypes* et de *BrowseNames*.

Un objectif de ce *Service* est de permettre la programmation en fonction des définitions de type. Sachant que les *BrowseNames* doivent être uniques dans le contexte des définitions de type, un *Client* peut créer un chemin de navigation qui est valide pour une définition de type et utiliser ce chemin sur des instances du type. Par exemple, un *ObjectType* "Boiler" (chaudière) peut avoir une *Variable* "HeatSensor" (capteur de chaleur) comme *InstanceDeclaration*. Un élément graphique programmé en fonction du "Boiler" peut avoir besoin d'afficher la *Value* du "HeatSensor". Si l'élément graphique était appelé sur "Boiler1", une instance de "Boiler", il aurait besoin d'appeler ce *Service* en spécifiant le *NodeId* de "Boiler1" comme *Nœud* de départ et le *BrowseName* du "HeatSensor" comme chemin de navigation. Le *Service* retournerait le *NodeId* du "HeatSensor" de "Boiler1" et l'élément graphique pourrait s'abonner à son *Attribut Value*.

Si un *Nœud* a plusieurs cibles ayant le même *BrowseName*, le *Serveur* doit retourner une liste de *NodeIds*. Cependant, comme l'un des principaux objectifs de ce *Service* est de prendre en charge la programmation en fonction des définitions de type, le *NodeId* du *Nœud* basé sur la définition de type du *Nœud* de départ est retourné comme le premier *NodeId* dans la liste.

5.8.4.2 Paramètres

Le Tableau 36 définit les paramètres du *Service*.

Tableau 36 – Paramètres du service TranslateBrowsePathsToNodeIds

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
browsePaths []	BrowsePath	Liste des chemins de navigation pour lesquels les *NodeIds* sont demandés. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
startingNode	NodeId	*NodeId* du *Nœud* de départ pour le chemin de navigation.
relativePath	RelativePath	Le chemin à suivre à partir du *startingNode*. Le dernier élément dans le *relativePath* doit toujours avoir un *targetName* spécifié. Cela restreint encore plus la définition du type RelativePath. Le *Serveur* doit retourner *Bad_BrowseNameInvalid* si le *targetName* est absent. La structure *RelativePath* est définie en 7.25.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	BrowsePathResult	Liste de résultats pour la liste de chemins de navigation. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *browsePaths*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour le chemin de navigation (voir 7.33 pour la définition de *StatusCode*).
targets []	BrowsePathTarget	Liste de cibles pour le *relativePath* à partir du *startingNode*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants. Un *Serveur* peut rencontrer une *Référence* à un *Nœud* dans un autre *Serveur* qu'il ne peut pas suivre pendant qu'il traite le *RelativePath*. Si cela se produit, le *Serveur* retourne le *NodeId* du *Nœud* externe et positionne le paramètre *remainingPathIndex* pour indiquer les éléments *RelativePath* qu'il est encore nécessaire de traiter. Pour parachever l'opération, le *Client* doit se connecter à l'autre *Serveur* et appeler de nouveau ce service en utilisant la cible comme le *startingNode* et les éléments non traités comme le *relativePath*.
targetId	ExpandedNodeId	L'identificateur pour une cible du *RelativePath*.
remainingPathIndex	Index	L'indice du premier élément non traité dans le *RelativePath*. Cette valeur doit être égale à la valeur maximale du type de données *Index* si tous les éléments ont été traités (voir 7.12 pour la définition d'*Index*).
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour la liste des chemins de navigation (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *browsePaths*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.8.4.3 Résultats des services

Le Tableau 37 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis en 7.33.

Tableau 37 – Codes de résultats des services TranslateBrowsePathsToNodeIds

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.8.4.4 StatusCodes

Le Tableau 38 définit les valeurs pour les paramètres *statusCode* de niveau opération qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 38 – Codes de résultats de niveau opération de TranslateBrowsePathsToNodeIds

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat. Ce code indique que le relativePath contenait une liste vide.
Bad_BrowseNameInvalid	Voir le Tableau 166 pour la description de ce code de résultat. Ce code indique qu'un TargetName était absent dans un RelativePath.
Uncertain_ReferenceOutOfServer	L'élément de chemin a des cibles qui sont dans un autre serveur.
Bad_TooManyMatches	L'opération demandée a de trop nombreuses correspondances à retourner. Il convient que les utilisateurs emploient des interrogations pour les volumineux jeux de résultats. Il convient que les *Serveurs* autorisent au moins 10 correspondances avant de retourner ce code d'erreur.
Bad_QueryTooComplex	L'opération demandée exige un trop grand nombre de ressources dans le serveur.
Bad_NoMatch	Le relativePath demandé ne peut être résolu en une cible à retourner.

5.8.5 Enregistrer les nœuds (RegisterNodes)

5.8.5.1 Description

Un *Serveur* n'a souvent pas d'accès direct aux informations qu'il gère. Les variables ou les services peuvent se trouver dans des systèmes sous-jacents quand un effort supplémentaire est requis pour établir une connexion avec ces systèmes. Le *Service RegisterNodes* peut être utilisé par des *Clients* pour enregistrer les *Nœuds* auxquels ils savent qu'ils vont accéder de façon répétée (par exemple Write, Call). Il permet aux *Serveurs* d'installer tout ce qui est nécessaire pour rendre les opérations d'accès plus efficaces. Les *Clients* peuvent espérer des améliorations de performance lorsqu'ils utilisent des *NodeIds* enregistrés, mais les mesures d'optimisation sont spécifiques au fournisseur. Pour les *Nodes Variable*, les *Serveurs* doivent concentrer leurs efforts d'optimisation sur l'*Attribut Value*.

La validité des *NodeIds* enregistrés ne peut être garantie que dans la *Session* courante. Les *Clients* doivent immédiatement supprimer les enregistrements des identificateurs inutiles pour libérer des ressources.

RegisterNodes ne valide pas les *NodeIds* issus de la demande. Les *Serveurs* copient simplement les *NodeIds* inconnus dans la réponse. Les erreurs de structure des *NodeIds* (violations de taille, types d'identificateur non valides) entraînent la défaillance du *Service* complet.

Pour les besoins de l'*Auditing* (Audits), les *Serveurs* ne doivent pas utiliser les *NodeIds* enregistrés, mais ils doivent utiliser seulement les *NodeId*s canoniques qui sont la valeur de l'*Attribut NodeId*.

5.8.5.2 Paramètres

Le Tableau 39 définit les paramètres du *Service*.

Tableau 39 – Paramètres du service RegisterNodes

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
nodesToRegister []	NodeId	Liste des *NodeIds* à enregistrer que le client a récupérés par navigation, interrogation ou de quelque autre manière.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
registeredNodeIds []	NodeId	Une liste de *NodeIds* que le *Client* doit utiliser pour des opérations d'accès ultérieures. La taille et l'ordre de cette liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToRegister*. Le *Serveur* peut retourner le *NodeId* de la demande ou un nouveau *NodeId* (un pseudonyme). Il est recommandé que le *Serveur* retourne un *NodeIds* numérique pour la dénomination. Si aucune optimisation n'est prise en charge pour un *Nœud*, le *Serveur* doit retourner le *NodeId* issu de la demande.

5.8.5.3 Résultats des services

Le Tableau 40 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 40 – Codes de résultats des services RegisterNodes

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat. Les *Serveurs* doivent rejeter complètement la demande *RegisterNodes* si l'un quelconque des *NodeIds* dans le paramètre *nodesToRegister* est structurellement non valide.

5.8.6 Supprimer l'enregistrement de nœuds (UnregisterNodes)

5.8.6.1 Description

Ce *Service* est utilisé pour supprimer l'enregistrement des *NodeIds* qui ont été obtenus via le service *RegisterNodes*.

UnregisterNodes ne valide pas les *NodeIds* issus de la demande. Les *Serveurs* doivent simplement supprimer l'enregistrement des *NodeIds* qui sont connus comme des *NodeIds* enregistrés. Tous les *NodeIds* qui sont dans la liste, mais ne sont pas des *NodeIds* enregistrés, sont simplement ignorés.

5.8.6.2 Paramètres

Le Tableau 41 définit les paramètres du *Service*.

Tableau 41 – Paramètres du service UnregisterNodes

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
nodesToUnregister []	NodeId	Une liste des *NodeIds* qui ont été obtenus via le service *RegisterNodes*.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).

5.8.6.3 Résultats des services

Le Tableau 42 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 42 – Codes de résultats des services UnregisterNodes

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.9 Jeu de services d'interrogation (Query Service Set)

5.9.1 Vue d'ensemble

Ce *Service Set* est utilisé pour émettre une *Query* (interrogation) vers un *Serveur*. La *Query* OPC UA est générique en ce qu'elle fournit un mécanisme de stockage sous-jacent indépendant de la fonction *Query* qui peut être utilisée pour accéder à une grande variété d'unités de stockage de données et de systèmes de gestion d'informations d'OPC UA. La *Query* OPC UA permet à un *Client* d'accéder à des données maintenues par un *Serveur* sans connaître le plan logique utilisé pour le stockage interne des données. La connaissance de l'*Espace d'Adresses* est suffisante.

Une *Application OPC UA* est censée utiliser les *Services Query* d'OPC UA comme une partie d'un processus d'initialisation ou une étape occasionnelle de synchronisation d'informations. Par exemple, *Query* OPC UA serait utilisée pour un accès à des données en vrac d'une unité de stockage persistante pour initialiser une application d'analyse avec l'état courant d'une configuration système. Une *Query* peut aussi être utilisée pour initialiser ou alimenter des données en vue d'un rapport.

Une *Query* définit les instances d'un ou plusieurs *TypeDefinitionNodes* dans l'*Espace d'Adresses* qu'il convient d'appliquer à un jeu d'*Attributs*. Les résultats retournés par un *Serveur* sont sous la forme d'une matrice de *QueryDataSets* (jeu de données d'interrogation). Les valeurs d'*Attributs* sélectionnées dans chaque *QueryDataSet* proviennent de la définition des *TypeDefinitionNodes ou TypeDefinitionNodes connexes* sélectionnés et les résultats apparaissent dans le même ordre que les *Attributs* qui ont été passés dans la *Query*. La *Query* prend également en charge le filtrage des *Nœuds* en fonction des valeurs d'*Attributs*, ainsi que les relations entre *TypeDefinitionNodes*.

Voir Annexe B pour des exemples d'interrogations.

5.9.2 Interroger des vues (Querying Views)

Une *Vue* est un sous-ensemble de l'*Espace d'Adresses* disponible dans le *Serveur*. Voir l'IEC 62541-5 pour une description de l'organisation des *Vues* dans l'*Espace d'Adresses*.

Pour toute *Vue* existante, une *Query* peut être utilisée pour retourner un sous-ensemble de données de la *Vue*. Lorsqu'une application émet une *Query* en fonction d'une *Vue*, seules les données définies par la *Vue* sont retournées. Les données non incluses dans la *Vue,* mais incluses dans l'*Espace d'Adresses* d'origine ne sont pas retournées.

Les *Services Query* prennent en charge l'accès à des données courantes et historiques. Le *Service* prend en charge un *Client* qui interroge une version antérieure de l'*Espace d'Adresses.* Les *Clients* peuvent spécifier une *ViewVersion* (Version de vue) ou un *Timestamp* (Horodatage) dans une *Query* pour accéder à des versions antérieures de l'*Espace d'Adresses*. Le service *Query* OPC UA est complémentaire au service Historical Access (Accès à historique) en ce que le premier est utilisé pour *Query* (interroger) un *Espace d'Adresses* qui existait à un moment donné et que le second est utilisé pour *Query* afin d'obtenir la valeur des *Attributs* au fil du temps. Ainsi, une *Query* peut être utilisée pour récupérer une partie d'un *Espace d'Adresses* antérieur et, de ce fait, un historique des

valeurs de l'*Attribut* peut être accessible à l'aide de Historical Access même si le *Nœud* ne se trouve plus dans *l'Espace d'Adresses* courant.

Les *Serveurs* qui prennent en charge *Query* sont censés être capables d'accéder à *l'Espace d'Adresses* qui est associé au *Serveur* local et à n'importe quelles *Vues* qui sont disponibles sur le *Serveur* local. Si une *Vue* ou l'espace d'adresses référence également un *Serveur* distant, l'interrogation peut être capable d'accéder à l'espace d'adresses du *Serveur* distant, mais cela n'est pas obligatoire. Si un *Serveur* accède effectivement à un *Serveur* distant, l'accès doit être accompli en utilisant l'identité d'utilisateur du *Client* tel que décrit en 5.5.1.

5.9.3 Émettre une demande (QueryFirst)

5.9.3.1 Description

Ce *Service* est utilisé pour émettre une demande *Query* vers le *Serveur*. La complexité de la *Query* peut être très simple ou hautement sophistiquée. La *Query* peut simplement demander des données issues d'instances d'un *TypeDefinitionNode* ou *TypeDefinitionNode* soumis à des restrictions spécifiées par le filtre. Par ailleurs, la *Query* peut demander des données issues d'instances de type *Nœuds* connexes en spécifiant un *RelativePath* partant d'un *TypeDefinitionNode* d'origine. Dans le filtre, un jeu séparé de chemins peut être construit pour limiter les instances qui fournissent les données. Un chemin filtrant peut inclure plusieurs opérateurs *RelatedTo* (Relié à) pour définir un chemin à sauts multiples entre des instances sources et des instances cibles. Par exemple, on pourrait passer au filtre des étudiants d'une école particulière, mais retourner des informations relatives aux étudiants et à leurs familles. Dans ce cas, la relation étudiant-école est parcourue pour le filtrage, mais la relation étudiant-famille est parcourue pour sélectionner des données. Pour une description complète de *ContentFilter,* voir 7.4; voir aussi B.1 pour des exemples simples et B.2 pour de plus complexes exemples de filtre de contenu et d'interrogations.

Le *Client* fournit une matrice de *NodeTypeDescription* qui spécifie le *NodeId* d'un *TypeDefinitionNode* et sélectionne les *Attributs* à retourner dans la réponse. Un client peut aussi fournir un jeu de *RelativePaths* à travers le système de type commençant à partir d'un *TypeDefinitionNode* de départ. En utilisant ces chemins, le client sélectionne un jeu d'*Attributs* à partir des *Nœuds* qui sont reliés à des instances du *TypeDefinitionNode* de départ. En outre, le *Client* peut demander au *Serveur* de retourner des instances des sous-types de *TypeDefinitionNodes*. Si un *Attribut* sélectionné n'existe pas dans un *TypeDefinitionNode,* mais existe dans un sous-type, il est admis avoir une valeur null dans le *TypeDefinitionNode* en question. Par conséquent, ceci ne constitue pas une condition d'erreur et une valeur null est retournée pour l'*Attribut*.

Le *Client* peut utiliser le paramètre filtre pour limiter le jeu de résultats en limitant les *Attributs* et les *Properties* à certaines valeurs. Une autre manière dont le *Client* peut utiliser un filtre pour limiter le jeu de résultats consiste à spécifier comment il convient de relier les instances, en utilisant des opérateurs *RelatedTo*. Dans ce cas, si une instance au sommet du chemin *RelatedTo* ne peut pas être suivie jusqu'au bas du chemin via des sauts spécifiés, aucun *QueryDataSet* n'est retourné pour l'instance de départ ou l'une quelconque des instances intermédiaires.

Lorsqu'il effectue une interrogation relative à des instances connexes dans le *RelativePath*, le *Client* peut facultativement demander des *Références*. Une *Référence* est demandée via un *RelativePath* qui inclut seulement un *ReferenceType*. Si toutes les *Références* sont désirées, le *ReferenceType* racine est alors énuméré. Ces *Références* sont retournées comme partie des *QueryDataSets*.

Les Services Query permettent une gestion particulière du champ targetName dans le RelativePath. Dans plusieurs cas d'utilisation des Services Query, un NodeId de type est nécessaire dans le chemin, à la place d'un QualifiedName. Le Client peut donc spécifier un NodeId dans le QualifiedName. Cela est réalisé en mettant le namespaceIndex du targetName à zéro et la partie nom du targetName à la représentation en XML du NodeId. La représentation en XML est définie dans l'IEC 62541-6. Quand les instances de

correspondance sont retournées comme le Nœud cible, le Nœud cible doit être une instance du type spécifié ou de son sous-type.

Le Tableau 43 définit les paramètres de demande et le Tableau 44 les paramètres de réponse pour le *Service QueryFirst*.

Tableau 43 – Paramètres de demande QueryFirst

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
view	ViewDescription	Spécifie une *Vue* et un contexte temporel au *Serveur* (voir 7.37 pour la définition de *ViewDescription*).
nodeTypes []	NodeTypeDescription	Il s'agit de la description du type de *Nœud*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
typeDefinitionNode	ExpandedNodeId	*NodeId* du *TypeDefinitionNode* de départ des instances pour lesquelles les données sont à retourner.
includeSubtypes	Boolean	Un fanion qui indique si, oui ou non, il convient que le *Serveur* inclue des instances des sous-types de TypeDefinitionNode dans la liste des instances du type *Node*.
dataToReturn []	QueryDataDescription	Spécifie un *Attribut* ou une *Référence* partant du typeDefinitionNode de départ le long d'un relativePath donné pour lequel retourner des données. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
relativePath	RelativePath	Chemin de navigation relatif au *Nœud* de départ qui identifie le *Nœud* contenant les données demandées, le *Nœud* de départ étant un *Nœud* d'instance du type défini par le *Nœud* de définition de type. Les *Nœuds* d'instance font l'objet d'une limitation supplémentaire par le filtre fourni comme partie de cet appel. Pour une définition de relativePath, voir 7.25. Ce chemin relatif peut se terminer sur une *Référence* et, dans ce cas, la *ReferenceDescription* de la *Référence* est retournée comme étant sa valeur. Le champ targetName du relativePath peut contenir un NodeId de type. Cela est réalisé en mettant le *namespaceIndex* du *targetName* à zéro et la partie nom du *targetName* de la représentation en XML du *NodeId*. La représentation en XML est définie dans l'IEC 62541-6. Quand les instances de correspondance sont retournées comme le *Nœud* cible, le *Nœud* cible doit être une instance du type spécifié ou de son sous-type.
attributeID	IntegerId	Identificateur de l'*Attribut*. Il doit être un identificateur d'*Attribut* valide. L'*IntegerId* est défini en 7.13. *L'IntegerId* pour les *Attributs* est défini dans l'IEC 62541-6. Si le *RelativePath* s'est terminé à une *Référence*, ce paramètre est 0 et ignoré par le serveur.
indexRange	NumericRange	Ce paramètre est utilisé pour identifier un élément simple d'une structure ou d'une matrice ou bien une plage simple d'indices pour les matrices. Si une plage d'éléments est spécifiée, les valeurs sont retournées sous forme de composé. Le premier élément est identifié par l'indice 0 (zéro). Le type *NumericRange* est défini en 7.21. Ce paramètre est null si l'*Attribut* spécifié n'est pas une matrice ou une structure. Cependant, si l'*Attribut* spécifié est une matrice ou une structure, et ce paramètre est donc null, tous les éléments sont à inclure dans la plage.
filter	ContentFilter	Les *Nœuds* obtenus doivent être limités aux *Nœuds* qui correspondent aux critères définis par le filtre. ContentFilter est l'objet d'un débat en 7.4. Si un filtre vide est fourni, l'*espace d'adresses* tout entier doit être examiné et tous les *Nœuds* contenant un *Attribut* ou une *Référence* demandé(e) correspondant aux critères sont retournés.
maxDataSetsToReturn	Counter	Le nombre de *QueryDataSets* que le *Client* souhaite que le *Serveur* retourne dans la réponse et sur chaque réponse ultérieure d'appel de continuation. Le *Serveur* est autorisé à limiter encore plus la réponse, mais ne doit pas dépasser cette limite. Une valeur 0 indique que le *Client* n'impose aucune limitation.
maxReferencesToReturn	Counter	Le nombre de *Références* que le *Client* souhaite que le *Serveur* retourne dans la réponse pour chaque *QueryDataSet* et sur chaque réponse ultérieure d'appel de continuation. Le *Serveur* est autorisé à limiter encore plus la réponse, mais ne doit pas dépasser cette limite. Une valeur 0 indique que le *Client* n'impose aucune limitation. Par exemple, dans le cas d'un résultat où quatre *Nœuds* sont retournés, mais chacun a 100 *Références* et cette limite est fixée à 50, seules les 50 premières *Références* pour chaque *Nœud* seront retournées sur l'appel initial et un point de continuation sera établi pour indiquer l'existence de données supplémentaires.

Tableau 44 – Paramètres de réponse QueryFirst

Nom	Type	Description
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
queryDataSets []	QueryDataSet	La matrice de *QueryDataSets*. Cette matrice est vide s'il n'y a pas de *Nœuds* ou de *Références* ayant satisfait aux critères de *nodeTypes*. Dans ce cas, le paramètre continuationPoint doit être vide. Le type *QueryDataSet* est défini en 7.22.
continuationPoint	ContinuationPoint	Valeur opaque définie par le serveur qui identifie le point de continuation. Le point de continuation est utilisé seulement lorsque les résultats de *Query* sont trop volumineux pour être retournés dans une seule réponse. «Trop volumineux» signifie dans ce contexte que le *Serveur* n'est pas capable de retourner une réponse plus volumineuse ou que le nombre de *QueryDataSets* à retourner dépasse le nombre maximal de *QueryDataSets* à retourner qui a été spécifié par le *Client* dans la demande. Le point de continuation est utilisé dans le *Service QueryNext*. Lorsqu'il n'est pas utilisé, la valeur de ce paramètre est null. Si un point de continuation est retourné, le *Client* doit appeler *QueryNext* pour récupérer le prochain jeu de *QueryDataSets* ou pour libérer des ressources pour le point de continuation dans le *Serveur*. Un point de continuation doit rester actif jusqu'à ce que le *Client* passe le point de continuation à *QueryNext* ou jusqu'à ce que la session soit fermée. Si le nombre maximal de points de continuation a été atteint, le point de continuation le plus ancien doit être réinitialisé. Le type *ContinuationPoint* est décrit en 7.6.
parsingResults []	ParsingResult	Liste de résultats d'analyse pour *QueryFirst*. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *NodeTypes*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants. Cette liste est alimentée par tous les codes de statut qui se rapportent au traitement des types de nœuds qui font partie de l'interrogation. La matrice peut être vide s'il ne s'est pas produit d'erreurs. Si un quelconque type de nœud a rencontré une erreur, tous les types de nœuds doivent avoir un code de statut associé.
statusCode	StatusCode	Résultat d'analyse pour la *NodeTypeDescription* demandée.
dataStatusCodes []	StatusCode	Liste de résultats pour *dataToReturn*. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *dataToReturn*. La matrice peut être vide s'il ne s'est pas produit d'erreurs.
dataDiagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *dataToReturn* (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *dataToReturn*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande d'interrogation.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour la *NodeTypeDescription* demandée. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande d'interrogation.
filterResult	ContentFilter Résultat	Une structure qui contient toutes les éventuelles erreurs associées au filtre. Cette structure doit être vide s'il ne s'est pas produit d'erreurs. Le type *ContentFilterResult* est défini en 7.4.2.

5.9.3.2 Résultats des services

Si la *Query* est invalide ou ne peut pas être traitée, les *QueryDataSets* ne sont pas retournés et seul un résultat de *Service* est retourné, filterResult, parsingResults, ainsi qu'éventuellement les *DiagnosticInfos* facultatives. Le Tableau 45 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 45 – Codes de résultats des services QueryFirst

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ContentFilterInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_ViewIdUnknown	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewTimestampInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewParameterMismatchInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ViewVersionInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_InvalidFilter	Le filtre fourni n'est pas valide, voir le filterResult pour les erreurs spécifiques.
Bad_NodelistError	Les NodeTypes fournis contiennent une erreur, voir les parsingResults pour les erreurs spécifiques.
Bad_InvalidView	La ViewDescription fournie n'est pas une ViewDescription valide.
Good_ResultsMayBeIncomplete	Il convient que le serveur ait suivi une référence à un nœud dans un serveur distant, mais il ne l'a pas fait. Le jeu de résultats peut être incomplet.

5.9.3.3 StatusCodes

Le Tableau 46 définit les valeurs pour le paramètre *statusCode* de parsingResults qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 46 – Codes de résultats de niveau opération de QueryFirst

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NotTypeDefinition	Le NodeId fourni n'était pas un NodeId de définition de type.
Bad_AttributeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.9.4 Poursuivre la demande (QueryNext)

5.9.4.1 Descriptions

Ce *Service* est utilisé pour demander le prochain jeu d'informations de réponse *QueryFirst* ou *QueryNext* qui sont trop volumineuses pour être envoyées dans une seule réponse. «Trop volumineuses» signifie dans ce contexte que le *Serveur* n'est pas capable de retourner une réponse plus volumineuse ou que le nombre de *QueryDataSets* à retourner dépasse le nombre maximal de *QueryDataSets* à retourner qui a été spécifié par le *Client* dans la demande d'origine. Le *QueryNext* doit être présenté dans la même session qui a été utilisée pour présenter le *QueryFirst* ou *QueryNext* qui se poursuit.

5.9.4.2 Paramètres

Le Tableau 47 définit les paramètres du *Service*.

Tableau 47 – Paramètres du service QueryNext

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
releaseContinuationPoint	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE le *continuationPoint* passé doit être réinitialisé pour libérer des ressources pour le point de continuation dans le *Serveur*. FALSE le *continuationPoint* passé doit être utilisé pour récupérer le prochain jeu de *QueryDatSets*. Un *Client* doit toujours utiliser le point de continuation retourné par une réponse *QueryFirst* ou *QueryNext* pour libérer les ressources pour le point de continuation dans le *Serveur*. Si le *Client* ne souhaite pas récupérer le prochain jeu d'informations de *Query*, *QueryNext* doit être appelé avec ce paramètre mis à TRUE. Si le paramètre est mis à TRUE, tous les paramètres de matrices dans la réponse doivent contenir des matrices vides.
continuationPoint	ContinuationPoint	Valeur opaque définie par le *Serveur* qui représente le point de continuation. La valeur pour le point de continuation a été retournée au *Client* dans une réponse *QueryFirst* ou *QueryNext* précédente. Cette valeur est utilisée pour identifier la demande *QueryFirst* ou *QueryNext* précédemment traitée qui se poursuit et le point dans le jeu de résultats à partir duquel il faut que la réponse de navigation continue. Le type *ContinuationPoint* est décrit en 7.6.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
queryDataSets []	QueryDataSet	La matrice de *QueryDataSets*. Le type *QueryDataSet* est défini en 7.22.
revisedContinuationPoint	ContinuationPoint	Valeur opaque définie par le *Serveur* qui représente le point de continuation. Elle est utilisée seulement si les informations à retourner sont trop volumineuses pour être contenues dans une seule réponse. Lorsqu'elle n'est pas utilisée ou lorsque le *releaseContinuationPoint* est mis, la valeur de ce paramètre est null. Le type *ContinuationPoint* est décrit en 7.6.

5.9.4.3 Résultats des services

Le Tableau 48 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 48 – Codes de résultats des services QueryNext

Id symbolique	Description
Bad_ContinuationPointInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.10 Jeu de services pour les attributs (Attribute Service Set)

5.10.1 Vue d'ensemble

Ce *Service Set* fournit les *Services* pour accéder à des *Attributs* qui sont partie intégrante des *Nœuds*.

5.10.2 Lecture (Read)

5.10.2.1 Description

Ce *Service* est utilisé pour lire un ou plusieurs *Attributs* d'un ou plusieurs *Nœuds*. Pour les valeurs d'*Attributs* construits dont les éléments ont des indices, comme une matrice, ce *Service* permet aux *Clients* de lire la totalité du jeu de valeurs à indices comme un composé, de lire des éléments individuels ou de lire des plages d'éléments du composé.

Le paramètre maxAge est utilisé pour ordonner au *Serveur* d'accéder à la valeur à partir de la source de données sous-jacente (telle qu'un appareil par exemple) si sa copie des données

est plus ancienne que celle que le maxAge spécifie. Si le *Serveur* ne peut pas satisfaire à l'âge maximal demandé, il retourne la valeur «de meilleur effort» au lieu de rejeter la demande.

5.10.2.2 Paramètres

Le Tableau 49 définit les paramètres du *Service*.

Tableau 49 – Paramètres du service Read

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
maxAge	Duration	Âge maximal de la valeur à lire en millisecondes. L'âge de la valeur est basé sur la différence entre le *ServeurTimestamp* (horodatage du serveur) et l'heure à laquelle le *Serveur* commence à traiter la demande. Par exemple, si le *Client* spécifie un *maxAge* de 500 millisecondes et la durée d'attente est de 100 millisecondes avant que le *Serveur* ne commence à traiter la demande, l'âge de la valeur retournée peut être 600 millisecondes avant l'heure à laquelle elle a été demandée. Si le *Serveur* a une ou plusieurs valeurs d'un *Attribut* qui sont dans la limite de l'âge maximal, il peut retourner n'importe laquelle de ces valeurs ou il peut lire une nouvelle valeur dans la source de données. Le nombre de valeurs d'un *Attribut* que possède un *Serveur* dépend du nombre de *MonitoredItems* qui sont définis pour cet *Attribut*. Dans tous les cas, le *Client* ne peut présumer de la copie des données qui sera retournée. Si le *Serveur* n'a pas de valeur s'inscrivant dans la limite de l'âge maximal, il doit tenter de lire une nouvelle valeur dans la source de données. Si le *Serveur* ne peut pas satisfaire au *maxAge* demandé, il retourne sa valeur de "meilleur effort" au lieu de rejeter la demande. Ce cas de figure peut se produire lorsque le temps mis par le *Serveur* pour traiter et retourner la nouvelle valeur de donnée après accès est supérieur à l'âge maximal spécifié. Si le *maxAge* est mis à 0, le *Serveur* doit tenter de lire une nouvelle valeur dans la source de données. Si *maxAge* est mis à une valeur supérieure ou égale à la valeur Int32 maximale, le *Serveur* doit tenter de récupérer une valeur mise en cache. Les valeurs négatives ne sont pas valides pour *maxAge*.
timestampsTo Return	Enum TimestampsTo Return	Une énumération qui spécifie les *Timestamps* (Horodatages) à retourner pour chaque *Variable Value Attribute*. L'énumération *TimestampsToReturn* est définie en 7.34.
nodesToRead []	ReadValueId	Liste des *Nœuds* et de leurs *Attributs* à lire. Pour chaque entrée dans la liste, un *StatusCode* est retourné, et s'il indique un succès, l'*Attribut Value* est également retourné. Le type de paramètre ReadValueId est défini en 7.23.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	DataValue	Liste de valeurs d'*Attribut* (voir 7.7 pour la définition *DataValue*). La taille et l'ordre de cette liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToRead*. Il y a une entrée dans cette liste pour chaque *Nœud* contenu dans le paramètre *nodesToRead*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de cette liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToRead*. Il y a une entrée dans cette liste pour chaque *Nœud* contenu dans le paramètre *nodesToRead*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.10.2.3 Résultats des services

Le Tableau 50 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 50 – Codes de résultats des services Read

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_MaxAgeInvalid	Le paramètre âge maximal n'est pas valide.
Bad_TimestampsToReturnInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.10.2.4 StatusCodes

Le Tableau 51 définit les valeurs pour le *statusCode* de niveau opération contenu dans la structure *DataValue* de chaque élément *values*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 51 – Codes de résultats de niveau opération de Read

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_AttributeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeNoData	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NotReadable	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.

5.10.3 Lecture de l'historique (HistoryRead)

5.10.3.1 Description

Ce *Service* est utilisé pour lire des valeurs ou *Événements* historiques d'un ou plusieurs *Nœuds*. Pour les valeurs d'*Attributs* construits dont les éléments ont des indices, comme une matrice, ce *Service* permet aux *Clients* de lire la totalité du jeu de valeurs à indices comme un composé, de lire des éléments individuels ou de lire des plages d'éléments du composé. Les *Serveurs* peuvent rendre les valeurs historiques disponibles aux *Clients* en utilisant ce *Service*, bien que les valeurs historiques elles-mêmes ne soient pas visibles dans *l'Espace d'Adresses*.

L'*Attribute AccessLevel* (Attribut Niveau d'accès) défini dans l'IEC 62541-3 indique la prise en charge des valeurs historiques par des *Nœuds*. Plusieurs paramètres de demande indiquent comment il faut que le *Serveur* accède à des valeurs de la source sous-jacente des données de l'historique. L'*Attribute EventNotifier* (Attribut Notificateur d'événement) défini dans l'IEC 62541-3 indique la prise en charge d'*Événements* historiques par des Nœuds.

Le paramètre *continuationPoint* dans l'*HistoryRead* est utilisé pour marquer le point à partir duquel poursuivre la lecture si toutes les valeurs n'ont pas pu être retournées dans une seule réponse. La valeur n'est pas connue du *Client* et elle est utilisée seulement pour maintenir les informations d'état pour le point de continuation pour le *Serveur*. Un *Serveur* peut utiliser l'horodatage du dernier élément de données retourné si l'horodatage est unique. Cela réduit la nécessité de stocker des informations d'état dans le *Serveur* pour le point de continuation.

Pour des détails supplémentaires relatifs à la lecture de données historiques et d'*Événements* historiques, voir l'IEC 62541-11.

5.10.3.2 Paramètres

Le Tableau 52 définit les paramètres du *Service*.

Tableau 52 – Paramètres du service HistoryRead

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
historyReadDetails	Extensible Parameter HistoryReadDetails	Les détails définissent les types de lectures d'historiques qui peuvent être effectuées. Le type de paramètre *HistoryReadDetails* est un type de paramètre extensible qui est défini formellement dans l'IEC 62541-11. Le type *ExtensibleParameter* est défini en 7.11.
timestampsToReturn	Enum TimestampsTo Return	Une énumération qui spécifie les horodatages à retourner pour chaque *Variable Value Attribute*. L'énumération *TimestampsToReturn* est définie en 7.34. Il n'est pas valide de spécifier un *TimestampsToReturn* de NEITHER. Dans ce cas, le *Serveur* doit retourner un *StatusCode Bad_InvalidTimestampArgument*. L'IEC 62541-11 définit les exceptions pour lesquelles ce paramètre doit être ignoré.
releaseContinuation Points	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE les *continuationPoints* passés doivent être réinitialisés pour libérer des ressources dans le *Serveur*. FALSE les *continuationPoints* passés doivent être utilisés pour récupérer le prochain jeu d'informations historiques. Un *Client* doit toujours utiliser le point de continuation retourné par une réponse *HistoryRead* pour libérer les ressources pour le point de continuation dans le *Serveur*. Si le *Client* ne souhaite pas récupérer le prochain jeu d'informations historiques, *HistoryRead* doit être appelé avec ce paramètre mis à TRUE.
nodesToRead []	HistoryReadValueId	Ce paramètre contient la liste d'éléments sur lesquels la récupération d'historique est à accomplir. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
nodeId	NodeId	Si l'*HistoryReadDetails* est RAW, PROCESSED, MODIFIED ou ATTIME: Le *nodeId* des *Nœuds* dont les valeurs historiques sont à lire. La valeur retournée doit toujours comprendre un horodatage. Si l'*HistoryReadDetails* est EVENTS: Le *NodeId* du *Nœud* dont l'historique des *Événements* est à lire. Si le *Nœud* ne prend pas en charge l'accès demandé pour des valeurs historiques ou des *Événements* historiques, une réponse d'erreur appropriée pour le *Nœud* donné doit être générée.
indexRange	NumericRange	Ce paramètre est utilisé pour identifier un élément simple d'une matrice ou une plage simple d'indices pour les matrices. Si une plage d'éléments est spécifiée, les valeurs sont retournées sous forme de composé. Le premier élément est identifié par l'indice 0 (zéro). Le type *NumericRange* est défini en 7.21. Ce paramètre est null si la valeur n'est pas une matrice. Cependant, si la valeur est une matrice, et ce paramètre est donc null, tous les éléments sont à inclure dans la plage.
dataEncoding	QualifiedName	Un *QualifiedName* qui spécifie le codage de données à retourner pour la *Value* à lire (voir 7.23 pour la définition de la façon de spécifier le codage de données). Le paramètre est ignoré lors de la lecture de l'historique d'*Événements*.
continuationPoint	ByteString	Pour chaque *NodesToRead*, ce paramètre spécifie un point de continuation retourné à partir d'un précédent appel *HistoryRead*, permettant au *Client* de poursuivre cette lecture à partir de la dernière valeur reçue. L'*HistoryRead* est utilisé pour sélectionner une séquence ordonnée de valeurs ou événements historiques. Un point de continuation marque un point dans cette séquence ordonnée et, ainsi, le *Serveur* retourne le sous-ensemble de la séquence qui suit le point en question. Une valeur null indique que ce paramètre n'est pas utilisé. Ce point de continuation est décrit plus en détail dans l'IEC 62541-11.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour le type *ResponseHeader*).
results []	HistoryReadResult	Liste de résultats lus. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToRead*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour l'élément *NodesToRead* (voir 7.33 pour la définition de *StatusCode*).
continuationPoint	ByteString	Ce paramètre est utilisé seulement si le nombre des valeurs à retourner est trop grand pour que celles-ci soient retournées dans une seule réponse. Lorsque ce paramètre n'est pas utilisé, sa valeur est null. Les *Serveurs* doivent prendre en charge au moins un point de continuation par *Session*. Les *Serveurs* spécifient un nombre maximal de points de continuation d'historiques par *Session* dans l'*Object* fonctions du *Serveur* défini dans l'IEC 62541-5. Un point de continuation doit rester actif jusqu'à ce que le *Client* passe le point de continuation à *HistoryRead* ou jusqu'à ce que la *Session* soit fermée. Si le nombre maximal de points de continuation a été atteint, le point

Nom	Type	Description
Request		
		de continuation le plus ancien doit être réinitialisé.
historyData	Extensible Parameter HistoryData	Les données d'historique retournées par le *Nœud*. Le type de paramètre *HistoryData* est un type de paramètre extensible qui est défini formellement dans l'IEC 62541-11. Il spécifie les types des données d'historique qui peuvent être retournées. Le type de base *ExtensibleParameter* est défini en 7.11.
diagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToRead*. Il y a une entrée dans cette liste pour chaque *Nœud* contenu dans le paramètre *nodesToRead*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.10.3.3 Résultats des services

Le Tableau 53 définit les résultats de *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 53 – Codes de résultats des services HistoryRead

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TimestampsToReturnInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_HistoryOperationInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_HistoryOperationUnsupported	Voir le Tableau 166 pour la description de ce code de résultat. L'opération demandée sur l'historique n'est pas prise en charge par le *Serveur*.

5.10.3.4 StatusCodes

Le Tableau 54 définit les valeurs pour le paramètre *statusCode* de niveau opération qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166. Les *StatusCodes* spécifiques à l'accès aux historiques sont définis dans l'IEC 62541-11.

Tableau 54 – Codes de résultats de niveau opération de HistoryRead

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_ContinuationPointInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_InvalidTimestampArgument	L'horodatage défini à retourner n'était pas valide.
Bad_HistoryOperationUnsupported	Voir le Tableau 166 pour la description de ce code de résultat. L'opération demandée sur l'historique n'est pas prise en charge pour le *Nœud* demandé.

5.10.4 Écrire (Write)

5.10.4.1 Description

Ce *Service* est utilisé pour écrire des valeurs dans un ou plusieurs *Attributs* d'un ou plusieurs *Nœuds*. Pour les valeurs d'*Attributs* construits dont les éléments ont des indices, comme une matrice, ce *Service* permet aux *Clients* d'écrire la totalité du jeu de valeurs à indices comme un composé, d'écrire des éléments individuels ou d'écrire des plages d'éléments du composé.

Les valeurs sont écrites dans la source de données (telle qu'un appareil par exemple) et le *Service* ne retourne rien jusqu'à ce qu'il écrive les valeurs ou détermine que la valeur ne peut pas être écrite. Dans certains cas, le *Serveur* écrit avec succès dans un système ou *Serveur* intermédiaire et ne sait pas si la source de données a été correctement mise à jour. Dans ces cas, il convient que le *Serveur* rapporte un code de succès qui indique que l'écriture n'a pas

été vérifiée. Dans les cas où le *Serveur* est à même de vérifier qu'il a réussi à écrire dans la source de données, il rapporte un succès inconditionnel.

L'ordre dans lequel les opérations sont traitées dans le *Serveur* n'est pas défini et dépend des différentes sources de données et de la logique interne du *Serveur*. Si une combinaison *Attribut* et *Nœud* est contenue dans plus d'une opération, l'ordre de traitement n'est pas défini. Si un *Client* exige un traitement séquentiel, le *Client* a besoin d'appels de *Services* séparés.

Il est possible que le *Serveur* puisse réussir à écrire certains *Attributs* et pas d'autres. Le retour en arrière est du ressort du *Client*.

Si un *Serveur* autorise l'écriture d'*Attributs* ayant le *DataType* LocalizedText, le Client peut ajouter ou écraser le texte pour un paramètre de lieu en écrivant le texte avec le *LocaleId* associé. Le fait d'écrire une *String* null pour le texte pour un paramètre de lieu doit supprimer la *String* pour le paramètre de lieu en question. Le fait d'écrire une *String* null pour le *paramètre de lieu* et une *String* non null pour le *texte* met le texte à un paramètre de lieu invariable. Le fait d'écrire une *String* null pour le *texte* et une *String* non null pour le *paramètre de lieu* doit supprimer les entrées pour tous les paramètres de lieu. Si un *Client* tente d'écrire un *paramètre de lieu* qui est soit non valide syntaxiquement, soit non pris en charge, le *Serveur* renvoie Bad_LocaleNotSupported.

5.10.4.2 Paramètres

Le Tableau 55 définit les paramètres du *Service*.

Tableau 55 – Paramètres du service Write

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
nodesToWrite []	WriteValue	Liste des *Nœuds* et de leurs *Attributs* à écrire. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
nodeId	NodeId	*NodeId* du *Nœud* qui contient les *Attributs*.
AttributeID	IntegerId	Identificateur de l'*Attribut*. Il doit s'agir d'un identificateur d'*Attribut* valide. L'*IntegerId* est défini en 7.13. Les IntegerIds pour les Attributs sont définis dans l'IEC 62541-6.
indexRange	NumericRange	Ce paramètre est utilisé pour identifier un élément simple d'une matrice ou une plage simple d'indices pour les matrices. Le premier élément est identifié par l'indice 0 (zéro). Le type *NumericRange* est défini en 7.21. Ce paramètre n'est pas utilisé si l'*Attribut* spécifié n'est pas une matrice. Cependant, si l'*Attribut* spécifié est une matrice et que ce paramètre n'est pas utilisé, tous les éléments sont à inclure dans la plage. Ce paramètre est null s'il n'est pas utilisé. Un *Serveur* doit retourner une erreur Bad_WriteNotSupported si une *indexRange* est fournie et si l'écriture d'*indexRange* n'est pas possible pour le *Nœud*.
value	DataValue	La valeur d'*Attribut* du Nœud (voir 7.7 pour la définition de *DataValue*). Si le paramètre *indexRange* est spécifié, la *Value* doit être une matrice même si un seul élément est en cours d'écriture. Si le *SourceTimestamp* ou le *ServerTimestamp* est spécifié, le *Serveur* doit utiliser ces valeurs. Le *Serveur* retourne une erreur Bad_WriteNotSupported s'il ne prend pas en charge l'écriture d'horodatages. Un *Serveur* doit retourner une erreur Bad_TypeMismatch si le type de données de la valeur écrite n'est pas le même que le type ou le sous-type du *DataType* de l'*Attribut*. Sur la base de la hiérarchie du *DataType*, les sous-types du *DataType* de l'*Attribut* doivent être acceptés par le *Serveur*. Pour le *Value Attribute*, le *DataType* est défini au moyen de l'*Attribut* du *DataType*. La structure d'une *ByteString* est la même que celle d'une matrice à une dimension de *Byte*. Un *Serveur* doit accepter une *ByteString* si une matrice de *Byte* est prévue. Le *Serveur* retourne une erreur Bad_DataEncodingUnsupported s'il ne prend pas en charge le codage de données fourni. Les *Simple DataTypes* (voir l'IEC 62541-3) utilisent la même représentation à la volée que leurs supertypes et écrire une valeur d'un *DataType* simple ne peut être distingué du fait d'écrire une valeur de son supertype. Le *Serveur* doit supposer qu'en recevant la représentation correcte à la volée pour un *DataType* simple, le type correct a été choisi. Les *Serveurs* peuvent imposer des validations supplémentaires de données sur la valeur indépendante du codage (par exemple, une image au format GIF dans une *ByteString*). Dans ce cas, le *Serveur* doit retourner une erreur Bad_TypeMismatch si la validation échoue.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des résultats pour les *Nœuds* à écrire (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToWrite*. Il y a une entrée dans cette liste pour chaque *Nœud* contenu dans le paramètre *nodesToWrite*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Nœuds* à écrire (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *nodesToWrite*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.10.4.3 Résultats des services

Le Tableau 56 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 56 – Codes de résultats des services Write

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.10.4.4 StatusCodes

Le Tableau 57 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 57 – Codes de résultats de niveau opération de Write

Id symbolique	Description
Good_CompletesAsynchronously	Voir le Tableau 165 pour la description de ce code de résultat. La valeur a été écrite avec succès dans un système intermédiaire, mais le *Serveur* ne sait pas si la source de données a été correctement mise à jour.
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_AttributeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeNoData	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_WriteNotSupported	L'opération d'écriture demandée n'est pas prise en charge. Si un *Client* tente d'écrire n'importe quelle combinaison de "value", "quality", "timestamp" alors que le *Serveur* ne prend pas en charge la combinaison demandée (qui pourrait être une seule grandeur telle que juste un horodatage), le *Serveur* ne doit pas effectuer d'écriture sur ce *Nœud* et doit retourner ce *StatusCode* pour ce *Nœud*. Il est aussi utilisé si l'écriture d'un IndexRange n'est pas prise en charge pour un *Nœud*.
Bad_NotWritable	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat. L'utilisateur courant n'a pas la permission d'écrire l'attribut.
Bad_OutOfRange	Voir le Tableau 166 pour la description de ce code de résultat. Si un *Client* tente d'écrire une valeur hors de la plage valide telle qu'une valeur non contenue dans le type de données d'énumération du *Nœud*, le *Serveur* doit retourner ce *StatusCode* pour ce *Nœud*.
Bad_TypeMismatch	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NoCommunication	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_LocaleNotSupported	Le paramètre de lieu dans l'opération d'écriture demandée n'est pas pris en charge.

5.10.5 Mise à jour de l'historique (HistoryUpdate)

5.10.5.1 Description

Ce *Service* est utilisé pour mettre à jour des valeurs ou *Événements* historiques d'un ou plusieurs *Nœuds*. Plusieurs paramètres de demande indiquent comment il faut que le *Serveur* mette à jour une valeur ou un *Événement* historique. Les actions valides sont Insert (Insérer), Replace (Remplacer) ou Delete (Supprimer).

5.10.5.2 Paramètres

Le Tableau 58 définit les paramètres du *Service*.

Tableau 58 – Paramètres du service HistoryUpdate

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
historyUpdateDetails []	Extensible Parameter HistoryUpdate Details	Les détails définis pour la mise à jour. Le type de paramètre *HistoryUpdateDetails* est un type de paramètre extensible qui est défini formellement dans l'IEC 62541-11. Il spécifie les types des mises à jour d'historique qui peuvent être accomplies. Le type *ExtensibleParameter* est défini en 7.11.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	HistoryUpdate Result	Liste des résultats de mise à jour pour les détails de mise à jour d'historiques. La taille et l'ordre de la liste correspondent à la taille et à l'ordre des éléments de détails du paramètre *historyUpdateDetails* spécifié dans la demande. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour la mise à jour du *Nœud* (voir 7.33 pour la définition de *StatusCode*).
operationResults []	StatusCode	Liste des *StatusCodes* pour les opérations à exécuter sur un *Nœud*. La taille et l'ordre de la liste correspondent à la taille et à l'ordre de toute liste définie par l'élément "details" communiqué dans un rapport par l'entrée de résultat.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les opérations à exécuter sur un *Nœud* (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre de toute liste définie par l'élément "details" communiqué dans un rapport par cette entrée de *updateResults*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les détails de mise à jour d'historiques. La taille et l'ordre de la liste correspondent à la taille et à l'ordre des éléments de détails du paramètre *historyUpdateDetails* spécifié dans la demande. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.10.5.3 Résultats des services

Le Tableau 59 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 59 – Codes de résultats des services HistoryUpdate

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.10.5.4 StatusCodes

Le Tableau 60 définit les valeurs pour les paramètres *statusCode* et *operationResults* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166. Les *StatusCodes* spécifiques à l'accès aux historiques sont définis dans l'IEC 62541-11.

Tableau 60 – Codes de résultats de niveau opération de HistoryUpdate

Id symbolique	Description
Bad_NotWritable	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_HistoryOperationInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_HistoryOperationUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat. L'utilisateur courant n'a pas la permission de mettre à jour l'historique.

5.11 Jeu de services de méthodes (Method Service Set)

5.11.1 Vue d'ensemble

Les *Méthodes* représentent les appels de fonction des *Objects*. Elles sont définies dans l'IEC 62541-3. Les *Méthodes* sont invoquées et retournent un résultat (couronné de succès ou non) seulement à la fin. Les temps d'exécution pour les méthodes peuvent varier, selon la fonction qu'elles accomplissent.

Le *Method Service Set* (Jeu de services de méthodes) définit les moyens pour invoquer des méthodes. Une *méthode* doit être un *composant* d'un *Object*. La découverte est fournie par le biais des *Services Browse* (Navigation) et *Query* (Interrogation). Les *Clients* découvrent les *méthodes* prises en charge par un *Serveur* en naviguant à la recherche des *Références* d'*Objects* propriétaires qui identifient leurs *méthodes* prises en charge.

Parce que des *Méthodes* peuvent commander un certain aspect des opérations d'une installation, l'invocation de méthodes peut dépendre des conditions d'environnement ou autres. Cela peut être particulièrement vrai lors d'une tentative d'invocation d'une méthode immédiatement après que celle-ci a parachevé son exécution. Les conditions indispensables pour invoquer la *Méthode* pourraient ne pas être encore retournées à l'état qui permet à la *Méthode* de redémarrer.

5.11.2 Appel (Call)

5.11.2.1 Description

Ce *Service* est utilisé pour appeler (invoquer) une liste de *Méthodes*. Chaque appel de *Méthode* est invoqué dans le contexte d'une *Session* existante. S'il est mis fin à cette *Session*, les résultats de l'exécution de la *Méthode* ne peuvent pas être retournés au *Client* et sont éliminés. Cela est indépendant de la tâche réellement exécutée au niveau du *Serveur*.

L'ordre dans lequel les opérations sont traitées dans le *Serveur* n'est pas défini et dépend des différentes tâches et de la logique interne du *Serveur*. Si une *Méthode* est contenue dans plusieurs opérations, l'ordre de traitement n'est pas défini. Si un *Client* exige un traitement séquentiel, le *Client* a besoin d'appels de *Services* séparés.

Ce *Service* permet le passage d'arguments d'entrée et de sortie en direction/provenance de la méthode. Ces arguments sont définis par les *Properties* (Propriétés) de la *Méthode*.

5.11.2.2 Paramètres

Le Tableau 61 définit les paramètres du *Service*.

Tableau 61 – Paramètres du service Call

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
methodsToCall []	CallMethodRequest	Liste des *Méthodes* à appeler. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
objectId	NodeId	Le *NodeId* doit être celui de l'*Object* ou *ObjectType* qui est la source d'une *Référence HasComponent* (ou sous-type de *Référence HasComponent*) à la *Methode* spécifiée en *methodId*. Voir l'IEC 62541-3 pour une description des *Objects* et de leurs *Méthodes*.
methodId	NodeId	*NodeId* de la *Méthode* à invoquer. Si l'*objectId* est le *NodeId* d'un *Object*, il peut utiliser le *NodeId* d'une *Methode* qui est la cible d'une *Référence HasComponent* de l'*ObjectType* de l'*Object*.
inputArguments []	BaseDataType	Liste de valeurs des arguments d'entrée. Une liste vide indique qu'il n'y a pas d'arguments d'entrée. La taille et l'ordre de cette liste correspondent à la taille et à l'ordre des arguments d'entrée définis par la *Property InputArguments* de la *Méthode*. Le nom, une description et le type de données de chaque argument sont définis par la structure *Argument* dans chaque élément de la *Property InputArguments* de la méthode.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	CallMethodResult	Résultat pour les appels de la *Méthode*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* de la *Méthode* exécutée dans le serveur. Ce *StatusCode* est mis au *StatusCode* Bad_InvalidArgument si au moins un argument d'entrée a enfreint une contrainte (par exemple type de données erroné, valeur hors plage). Ce *StatusCode* est mis à bad *StatusCode* si l'exécution de la *Méthode* a échoué dans le *Serveur*, par exemple sur la base d'une exception.
inputArgumentResults []	StatusCode	Liste de *StatusCodes* correspondant aux *inputArguments*. Cette liste est vide sauf si le résultat de niveau opération est Bad_InvalidArgument Si cette liste est remplie, elle a la même longueur que la liste *inputArgument*
inputArgumentDiagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic correspondant aux *inputArguments*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.
outputArguments []	BaseDataType	Liste de valeurs des arguments de sortie. Une liste vide indique qu'il n'y a pas d'arguments de sortie. La taille et l'ordre de cette liste correspondent à la taille et à l'ordre des arguments de sortie définis par la *Property OutputArguments* de la *Méthode*. Le nom, une description et le type de données de chaque argument sont définis par la structure *Argument* dans chaque élément de la *Property OutputArguments* de la méthode
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour le *statusCode* des *results*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.11.2.3 Résultats des services

Le Tableau 62 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 62 – Codes de résultats des services Call

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.11.2.4 StatusCodes

Le Tableau 63 définit les valeurs pour les paramètres *statusCode* et *inputArgumentResults* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 63 – Codes de résultats de niveau opération de Call

Id symbolique	Description
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat. Sert à indiquer que l'objet spécifié n'est pas valide.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat. Sert à indiquer que l'objet spécifié n'est pas valide.
Bad_ArgumentsMissing	Le client n'a pas spécifié la totalité des arguments d'entrée pour la méthode.
Bad_TooManyArguments	Le client a spécifié plus d'arguments d'entrée que le nombre défini pour la méthode.
Bad_InvalidArgument	Voir le Tableau 166 pour la description de ce code de résultat. Sert à indiquer dans les résultats de niveau opération qu'un ou plusieurs des arguments d'entrée ne sont pas valides. Les inputArgumentResults contiennent le code de statut spécifique pour chaque argument non valide.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_MethodInvalid	L'identificateur de la méthode ne se réfère pas à une méthode pour l'objet spécifié.
Bad_OutOfRange	Voir le Tableau 166 pour la description de ce code de résultat. Sert à indiquer qu'un argument d'entrée se trouve à l'extérieur de la plage acceptable.
Bad_TypeMismatch	Voir le Tableau 166 pour la description de ce code de résultat. Sert à indiquer qu'un argument d'entrée n'a pas le type de données correct. La structure d'une ByteString est la même que celle d'une matrice à une dimension de *Byte*. Un *Serveur* doit accepter une ByteString si une matrice de Byte est prévue.
Bad_NoCommunication	Voir le Tableau 166 pour la description de ce code de résultat.

5.12 Jeu de services des éléments surveillés (MonitoredItem Service Set)

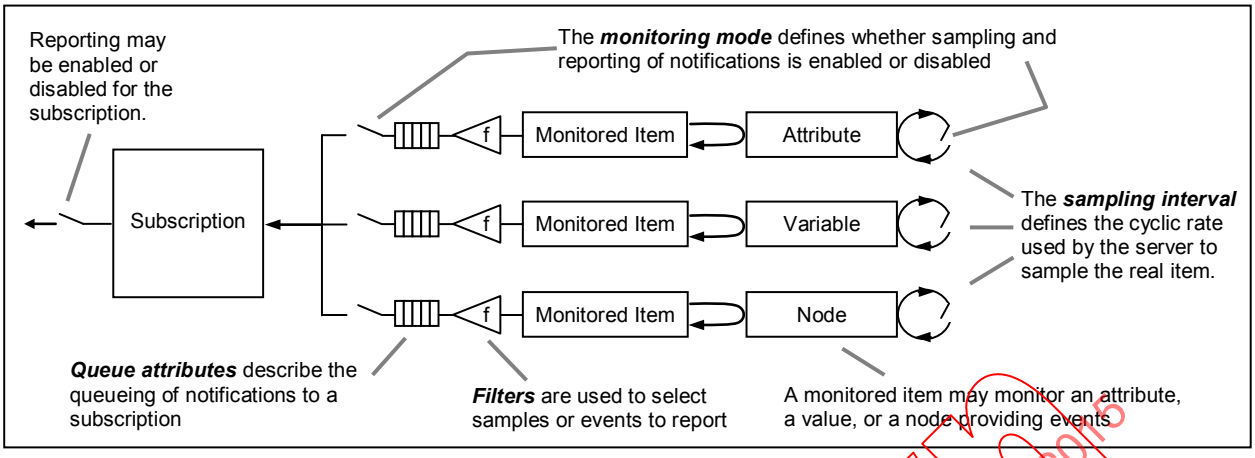
5.12.1 Modèle pour les éléments surveillés (MonitoredItem Model)

5.12.1.1 Vue d'ensemble

Les *Clients* définissent des *MonitoredItems* pour s'abonner à des données et à des *Événements*. Chaque *MonitoredItem* identifie l'élément à surveiller et le *Subscription* à utiliser pour envoyer des *Notifications*. L'élément à surveiller peut être n'importe quel *Node Attribute*.

Les *Notifications* sont des structures de données qui décrivent l'apparition des modifications de données et d'*Événements*. Elles sont empaquetées dans des *NotificationMessages* pour le transfert au *Client*. Le *Subscription* (abonnement) envoie périodiquement des *NotificationMessages* selon un intervalle d'édition spécifié par l'utilisateur, et le cycle durant lequel ces messages sont envoyés est appelé cycle d'édition.

Il a été défini quatre paramètres primaires pour les *MonitoredItems* qui indiquent au *Serveur* comment l'élément est à échantillonner, évaluer et rapporter. Ces paramètres sont l'intervalle d'échantillonnage, le mode de surveillance, le filtre et le paramètre de mise en file d'attente. La Figure 15 illustre ces concepts.



Anglais	Français
Reporting may be enabled or disabled for the subscription.	La production de rapports peut être activée ou désactivée pour l'abonnement.
Monitored Item	Élément surveillé
Attribute	Attribut
Variable	Variable
Node	Nœud
The *sampling interval* defines the cyclic rate used by the server to sample the real item.	L'*intervalle d'échantillonnage* définit la fréquence cyclique utilisée par le serveur pour échantillonner l'élément réel.
Subscription	Abonnement
The *monitoring mode* defines whether sampling and reporting of notifications are enabled or disabled	Le *mode de surveillance* définit si l'échantillonnage et les rapports de notifications sont activés ou désactivés
Filters are used to select samples or events to report	Les *filtres* sont utilisés pour sélectionner des échantillons ou des événements à consigner dans des rapports
A monitored item may monitor an attribute, a value, or a node providing events	Un élément surveillé peut surveiller un attribut, une valeur ou un nœud fournissant des événements
Queue attributes describe the queueing of notifications to a subscription	Les *attributs de file d'attente* décrivent la mise en file d'attente de notifications destinées à un abonnement

Figure 15 – Modèle MonitoredItem

Les *Attributs* autres que l'*Attribute Value* sont seulement surveillés pour détecter un changement de valeur. Le filtre n'est pas utilisé pour ces *Attributs*. Toute modification de valeur pour ces *Attributs* entraîne la création d'une *Notification*.

L'*Attribute Value* (Valeur Attribut) est utilisé pour surveiller des *Variables*. Les valeurs des *Variables* sont surveillées pour détecter une modification de valeur ou une modification de leur statut. Les filtres définis dans la présente Norme (voir 7.16.2) et dans l'IEC 62541-8 sont utilisés pour déterminer si la modification de la valeur est suffisamment forte pour entraîner la création d'une *Notification* pour cette *Variable*.

Les *Objects* et les vues peuvent être utilisés pour surveiller les *Événements*. Les *Événements* sont disponibles uniquement à partir *des Nœuds* pour lesquels le bit *SubscribeToEvents* de l'*Attribute EventNotifier* est positionné. Le filtre défini dans la présente Norme (voir 7.16.3) est utilisé pour déterminer si un *Événement* reçu *du Nœud* est envoyé au *Client*. Le filtre permet aussi de sélectionner des champs de l'*EventType* qui seront contenus dans l'*Événement* tels que *EventId*, *EventType*, *SourceNode*, *Time* et *Description*.

L'IEC 62541-3 décrit le modèle *Evénement* et les *EventTypes* de base.

Les *Properties* des *EventTypes* de base et la représentation des *EventTypes* de base dans *l'Espace d'Adresses* sont spécifiées dans l'IEC 62541-5.

5.12.1.2 Intervalle d'échantillonnage

Chaque *MonitoredItem* créé par le *Client* reçoit un intervalle d'échantillonnage qui lui est affecté et qui est soit hérité de l'intervalle d'édition de *Subscription*, soit défini spécifiquement pour se substituer à cette fréquence. Un nombre négatif indique que l'intervalle d'échantillonnage par défaut défini par l'intervalle d'édition de *Subscription* est demandé. L'intervalle d'échantillonnage indique la fréquence la plus rapide à laquelle il convient que le *Serveur* échantillonne sa source sous-jacente pour les modifications de données.

Un *Client* doit définir un intervalle d'échantillonnage de 0 s'il s'abonne à des *Événements*.

L'intervalle d'échantillonnage affecté définit une fréquence cyclique de «meilleur effort» que le *Serveur* utilise pour échantillonner l'élément à partir de sa source. «Meilleur effort» signifie dans ce contexte que le *Serveur* fait de son mieux pour échantillonner à cette fréquence. L'échantillonnage à des fréquences plus rapides que celle-ci est acceptable, mais n'est pas indispensable pour répondre aux besoins du *Client*. La façon dont le *Serveur* traite de la fréquence d'échantillonnage et la fréquence à laquelle il sonde effectivement ses sources de données en interne sont un détail d'implémentation du *Serveur*. Cependant, le temps entre les valeurs retournées au *Client* doit être supérieur ou égal à l'intervalle d'échantillonnage.

Le *Client* peut aussi spécifier 0 pour l'intervalle d'échantillonnage, qui indique qu'il convient pour le *Serveur* d'utiliser la fréquence pratique la plus rapide. Il est escompté que les *Serveurs* prendront en charge uniquement un jeu limité d'intervalles d'échantillonnage pour optimiser leur fonctionnement. Si l'intervalle exact demandé par le *Client* n'est pas pris en charge par le *Serveur*, le *Serveur* affecte au *MonitoredItem* l'intervalle le plus approprié tel que déterminé par le *Serveur*. Il retourne au *Client* cet intervalle affecté. L'*Object Server* Capabilities (l'objet Fonctions du serveur) défini dans l'IEC 62541-5 identifie les intervalles d'échantillonnage pris en charge par le *Serveur*.

Le *Serveur* peut prendre en charge les données qui sont recueillies sur la base d'un modèle d'échantillonnage ou générées sur la base d'un modèle piloté par des exceptions. L'intervalle d'échantillonnage pris en charge le plus rapide peut être égal à 0, ceci indique que l'élément de donnée est piloté par des exceptions au lieu d'être échantillonné périodiquement. Un modèle piloté par des exceptions signifie que le système sous-jacent ne nécessite pas d'échantillonnage et rapporte les changements apportés aux données.

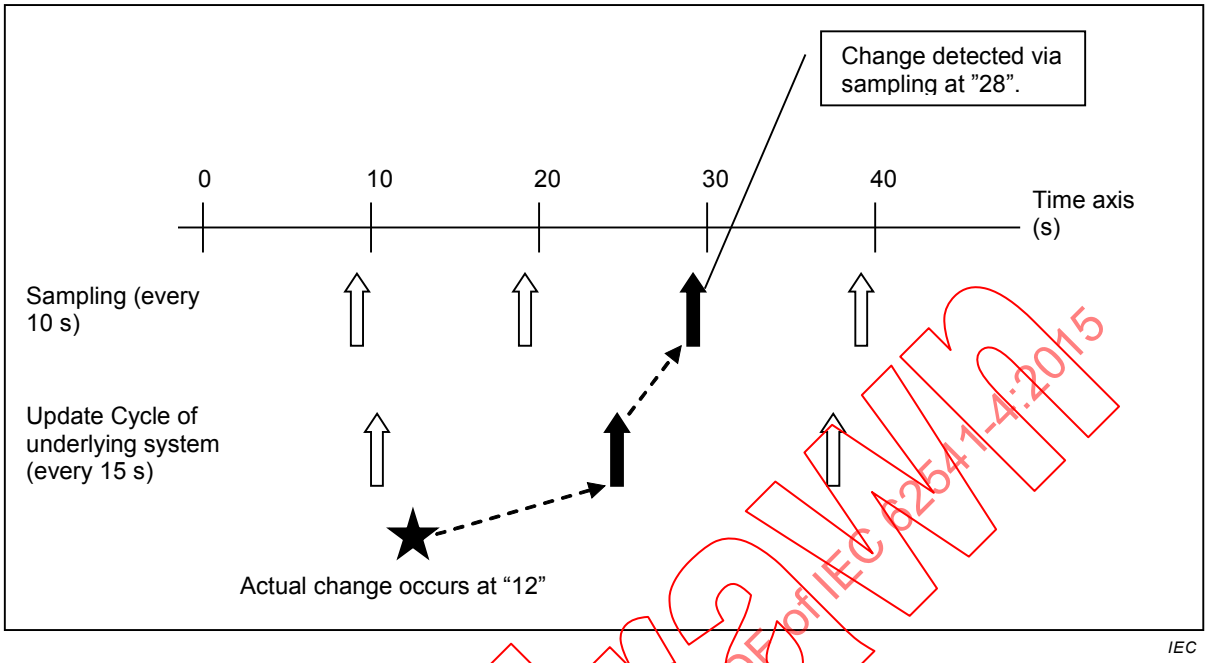
Le *Client* peut utiliser des valeurs révisées de l'intervalle d'échantillonnage comme suggestion pour l'établissement de l'intervalle d'édition ainsi que le compte d'entretien d'un *Subscription*. Si, par exemple, le plus petit intervalle d'échantillonnage révisé pour les *MonitoredItems* est 5 secondes, il convient que le temps avant qu'un entretien ne soit envoyé soit supérieur à 5 secondes.

A noter que, dans de nombreux cas, le *Serveur* OPC UA fournit l'accès à un système découplé et n'a donc aucune connaissance de la logique de mise à jour de données. Dans ce cas, même si le *Serveur* OPC UA échantillonne à la fréquence négociée, les données peuvent être mises à jour par le système sous-jacent à une fréquence beaucoup plus lente. Dans ce cas, les modifications ne peuvent être détectées qu'à cette fréquence plus lente.

Si le comportement par lequel le système sous-jacent met à jour l'élément est connu, il sera disponible via l'*Attribut MinimumSamplingInterval* défini dans l'IEC 62541-3. Si le *Serveur* spécifie une valeur pour l'*Attribut MinimumSamplingInterval*, il doit toujours retourner un *revisedSamplingInterval* qui est supérieur ou égal au *MinimumSamplingInterval* si le *Client* s'abonne à l'*Attribute Value*.

Il convient que les *Clients* sachent que l'échantillonnage par le *Serveur* OPC UA et le cycle de mise à jour du système sous-jacent ne sont pas, en général, synchronisés. Cela peut

introduire des retards supplémentaires dans la détection des modifications, comme illustré à la Figure 16.



Anglais	Français
Sampling (every 10 s)	Échantillonnage (toutes les 10 s)
Update Cycle of underlying system (every 15 s)	Cycle de mise à jour du système sous-jacent (toutes les 15 s)
Change detected via sampling at "28"	Changement détecté via l’échantillonnage à "28"
Actual change occurs at "12"	Un changement réel se produit à "12"
Time axis (s)	Axe des temps (s)

Figure 16 – Retard type pour détecter des modifications

5.12.1.3 Mode de surveillance

Le paramètre mode de surveillance est utilisé pour activer et désactiver l'échantillonnage d'un *MonitoredItem* et aussi pour permettre d'activer et de désactiver de façon indépendante la production de rapports de *Notifications*. Cette fonction permet de configurer un *MonitoredItem* pour échantillonner, échantillonner et rapporter ou ni l'un ni l'autre. La désactivation de l'échantillonnage ne change les valeurs d'aucun des autres paramètres de *MonitoredItem*, tels que son intervalle d'échantillonnage.

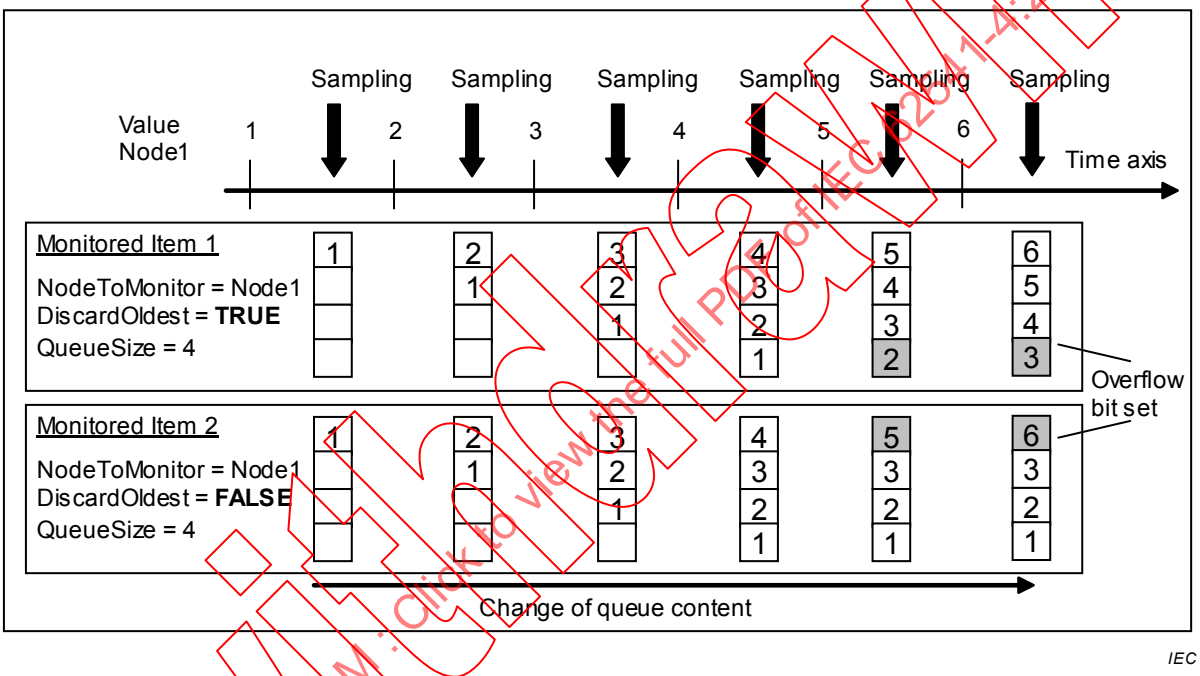
Lorsqu'un *MonitoredItem* est activé (c'est-à-dire lorsque le *MonitoringMode* est passé de *DISABLED* à *SAMPLING* ou à *REPORTING*) ou qu'il est créé dans l'état activé, le *Serveur* doit signaler le premier échantillon dès que possible dans le rapport et l'heure de cet échantillon devient le point de départ pour le prochain intervalle d'échantillonnage.

5.12.1.4 Filtre

Chaque fois qu'un *MonitoredItem* est échantillonné, le *Serveur* évalue l'échantillon en utilisant le filtre défini pour le *MonitoredItem*. Le paramètre filter définit les critères que le *Serveur* utilise pour déterminer s'il convient de générer une *Notification* pour l'échantillon. Le type de filtre dépend du type d'élément qui est surveillé. Par exemple, le *DataChangeFilter* et l'*AggregateFilter* sont utilisés pour surveiller les *Variable Values* (Valeurs de variables) et l'*EventFilter* est utilisé pour surveiller des *Événements*. L'échantillonnage et l'évaluation, y compris l'utilisation de filtres, sont décrits dans la présente Norme. Des filtres supplémentaires peuvent être définis dans d'autres parties de la présente série de normes.

5.12.1.5 Paramètres de mise en file d'attente

Si l'échantillon satisfait aux critères de filtrage, une *Notification* est générée et mise en file d'attente pour le transfert par *Subscription*. La taille de la file d'attente est définie lorsque *MonitoredItem* est créé. Lorsque la file d'attente est pleine et qu'une nouvelle *Notification* est reçue, le *Serveur* soit élimine la plus ancienne *Notification* et place la nouvelle dans la file d'attente, soit remplace la dernière valeur ajoutée à la file d'attente par la nouvelle valeur. À sa création, le *MonitoredItem* est configuré pour l'une de ces politiques de rejet. Si une Notification est rejetée pour une *DataValue* et que la taille de la file d'attente est supérieure à un, le bit *Overflow* (fanion) dans la partie *InfoBits* du *statusCode* de la *DataValue* est établi. Si *discardOldest* est TRUE, la valeur la plus ancienne est supprimée de la file d'attente et la valeur suivante dans la file d'attente établit le fanion. Si *discardOldest* est FALSE, la dernière valeur ajoutée à la file d'attente est remplacée par la nouvelle valeur. La nouvelle valeur établit le fanion pour indiquer les valeurs perdues dans le *NotificationMessage* suivant. La Figure 17 illustre la gestion du dépassement de capacité de la file d'attente.



Anglais	Français
Sampling	Échantillonnage
Time axis	Axe des temps
Monitored Item	Élément surveillé
Change of queue content	Modification du contenu de la file d'attente
Overflow bit set	Établissement du bit de dépassement de capacité

Figure 17 – Gestion du dépassement de capacité de la file d'attente

Si la taille de la file d'attente est 1, la file d'attente devient un tampon qui contient en permanence la *Notification* la plus récente. Dans ce cas, si l'intervalle d'échantillonnage du *MonitoredItem* est plus court que l'intervalle d'édition de *Subscription*, le *MonitoredItem* sera en suréchantillonnage et le *Client* recevra toujours la valeur la plus récente. La politique de rejet est ignorée si la taille de la file d'attente est un.

Par ailleurs, le *Client* peut souhaiter s'abonner à un flux continu de *Notifications* sans le moindre trou, mais ne souhaite pas que celles-ci soient rapportées à l'intervalle d'échantillonnage. Dans ce cas, le *MonitoredItem* met en place une file d'attente suffisamment longue pour contenir toutes les *Notifications* générées entre deux cycles d'édition consécutifs. Puis, à chaque cycle d'édition, *Subscription* envoie au *Client* toutes les *Notifications* mises en

file d'attente pour le *MonitoredItem*. Le *Serveur* doit retourner les *Notifications* pour tout élément particulier dans l'ordre dans lequel elles sont stockées dans la file d'attente.

Le *Serveur* peut échantillonner à une fréquence plus rapide que l'intervalle d'échantillonnage pour prendre en charge d'autres *Clients*; il convient que le *Client* attende seulement les valeurs à l'intervalle d'échantillonnage négocié. Le *Serveur* peut délivrer moins de valeurs que ne le dicte l'intervalle d'échantillonnage, sur la base des contraintes du filtre et de l'implémentation. Si un *DataChangeFilter* est configuré pour un *MonitoredItem*, il est toujours appliqué à la valeur la plus récente dans la file d'attente en comparaison à l'échantillon courant.

Si, par exemple, l'*AbsoluteDeadband* dans le *DataChangeFilter* est «10», la file d'attente peut être constituée de valeurs dans l'ordre suivant:

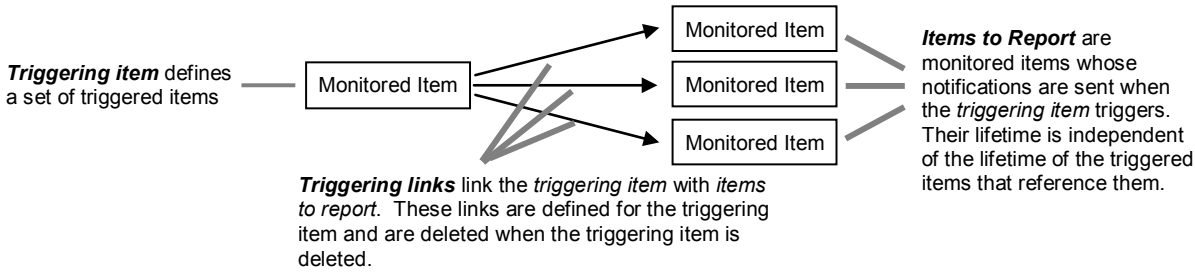
- 100
- 111
- 100
- 89
- 100

La mise des données en file d'attente peut conduire à un comportement inattendu lorsqu'un filtre *Deadband* est utilisé et que le nombre de modifications rencontrées est supérieur au nombre de valeurs qui peuvent être maintenues. La première nouvelle valeur dans la file d'attente peut ne pas dépasser la limite de *Deadband* de la valeur précédente envoyée au *Client*.

La taille de la file d'attente est la valeur maximale prise en charge par le *Serveur* lors de la surveillance d'*Événements*. Dans ce cas, le *Serveur* a la responsabilité du tampon d'*Événements*. Si des *Événements* sont perdus, un *Événement* du type *EventQueueOverflowEventType* est placé dans la file d'attente. Cet Événement est généré lorsque le premier Événement est à éliminer sur un MonitoredItem s'abonnant à des Événements. Il est mis dans la File d'attente du MonitoredItem en plus de la taille de la File d'attente définie pour ce MonitoredItem sans rejeter aucun autre Événement. Si *discardOldest* est mis à TRUE, il est placé au début de la file d'attente et n'est jamais rejeté, autrement à la fin. Un *Serveur* combiné ne doit pas transmettre un tel *Événement*. Il doit être géré comme les autres scénarios d'erreur de connexion.

5.12.1.6 Modèle de déclenchement

Le *Service MonitoredItems* permet d'ajouter des éléments qui sont signalés dans un rapport seulement lorsqu'un autre élément est déclenché (l'élément de déclenchement). Cela est réalisé en créant des liens entre les éléments déclenchés et les éléments à rapporter. Le mode de surveillance des éléments à rapporter est mis en échantillonnage seul et, de ce fait, il échantillonnera et mettra les *Notifications* en file d'attente sans les signaler dans un rapport. La Figure 18 illustre ce concept.



IEC

Anglais	Français
Triggering item defines a set of triggered items	Le ***Triggering item*** (élément déclencheur) définit un jeu d'éléments déclenchés
Monitored item	Élément surveillé
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.	Les ***Triggering links*** (liaisons de déclenchement) relient le *triggering item* à des *items to report* (*éléments à rapporter*). Ces liaisons sont définies pour l'élément déclencheur et sont supprimées lorsque l'élément déclencheur est supprimé
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	Les ***items to report*** (éléments à rapporter) sont des éléments surveillés dont les notifications sont envoyées lorsque l'*élément déclencheur* déclenche. Leur durée de vie est indépendante de la durée de vie des éléments déclenchés qui les référencent

Figure 18 – Modèle de déclenchement

Le mécanisme de déclenchement est une caractéristique utile qui permet aux *Clients* de réduire le volume de données à la volée en configurant certains éléments pour qu'ils échantillonnent fréquemment, mais se limitent seulement à produire un rapport lorsqu'un autre *Événement* se produit.

Les comportements de déclenchement suivants sont spécifiés.

a) Si le mode monitoring (de surveillance) de l'élément déclencheur est *SAMPLING, alors*, lorsque l'élément déclencheur déclenche les éléments à rapporter, cela ne fait pas l'objet d'un rapport.

b) Si le mode monitoring (de surveillance) de l'élément déclencheur est *REPORTING, alors,* lorsque l'élément déclencheur déclenche les éléments à rapporter, cela fait l'objet d'un rapport.

c) Si le mode monitoring (de surveillance) de l'élément déclencheur est *DISABLED*, alors l'élément déclencheur ne déclenche pas les éléments à rapporter.

d) Si le mode monitoring (de surveillance) de l'élément à rapporter est *SAMPLING, alors,* lorsque l'élément déclencheur déclenche les éléments à rapporter, cela fait l'objet d'un rapport.

e) Si le mode monitoring (de surveillance) de l'élément à rapporter est *REPORTING*, de ce fait, l'élément déclencheur est ignoré. Toutes les notifications des éléments à rapporter sont envoyées à l'expiration de l'intervalle d'édition.

f) Si le mode monitoring (de surveillance) de l'élément à rapporter est *DISABLED*, alors il n'y a pas d'échantillonnage de l'élément à rapporter et donc aucune notification à rapporter.

g) Le premier déclenchement doit se produire lorsque la première notification est mise en file d'attente pour l'élément déclencheur après la création de la liaison.

Les *Clients* créent et suppriment des liaisons de déclenchement entre un élément déclencheur et un ensemble d'éléments à rapporter. Si le *MonitoredItem* qui représente un

élément à rapporter est supprimé avant que sa liaison associée ne soit supprimée, la liaison de déclenchement est également supprimée, mais l'élément déclencheur n'est pas altéré.

Il convient de ne pas confondre la suppression d'un *MonitoredItem* avec le retrait de l'*Attribut* qu'il surveille. Si le Nœud qui contient l'*Attribut* surveillé est supprimé, le *MonitoredItem* génère une *Notification* avec un *StatusCode Bad_NodeIdUnknown* qui indique la suppression, mais le *MonitoredItem* n'est pas supprimé.

5.12.2 Créer des éléments surveillés (CreateMonitoredItems)

5.12.2.1 Description

Ce *Service* est utilisé pour créer et ajouter un ou plusieurs *MonitoredItems* à *Subscription*. Un *MonitoredItem* est supprimé automatiquement par le *Serveur* lorsque *Subscription* est supprimé. La suppression d'un *MonitoredItem* entraîne celle de l'ensemble complet des liaisons d'éléments déclenchés, mais n'a pas d'effet sur les *MonitoredItems* référencés par les éléments déclenchés.

L'appel à répétitions du *Service CreateMonitoredItems* pour ajouter chaque fois un petit nombre de *MonitoredItems* peut affecter défavorablement la performance du *Serveur*. Il convient que les *Clients* ajoutent plutôt un jeu complet de *MonitoredItems* à *Subscription* chaque fois que possible.

Lorsqu'un *MonitoredItem* est ajouté, le *Serveur* exécute un traitement d'initialisation pour celui-ci. Le traitement d'initialisation est défini par le type de *Notification* de l'élément surveillé. Les types de *Notification* sont spécifiés dans la présente Norme et dans les parties de la série de normes qui spécifient les types d'accès, telles que l'IEC 62541-8. Voir l'IEC TR 62541-1 pour la description des Parties Types d'accès

Lorsqu'un utilisateur ajoute un élément surveillé dont l'accès en lecture est refusé à l'utilisateur, l'opération d'ajout de l'élément doit réussir et le mauvais statut Bad_NotReadable ou Bad_UserAccessDenied doit être retourné dans la réponse Publish (éditer). Il s'agit du même comportement que dans le cas où les droits d'accès sont modifiés après l'appel à *CreateMonitoredItems*. Si les droits d'accès passent aux droits de lecture, le *Serveur* doit commencer à envoyer des données pour le *MonitoredItem*. La même procédure doit être appliquée pour un *IndexRange* qui ne fournit pas de données pour la valeur courante, mais qui peut fournir des données plus tard.

Les *Nœuds* surveillés peuvent être retirés de l'*Espace d'Adresses* après la création d'un *MonitoredItem*. Cela n'affecte pas la validité du *MonitoredItem,* mais un *Bad_NodeIdUnknown* doit être retourné dans la réponse Publish.

Le retour des informations de réglage de diagnostic dans l'en-tête de demande des *CreateMonitoredItems* ou le dernier *Service ModifyMonitoredItems* est appliqué aux *Monitored Items* et il sert de réglages d'informations de diagnostic lors de l'envoi des Notifications dans la réponse *Publish*.

Les valeurs de demande illégales pour les paramètres pouvant être révisés ne génèrent pas d'erreur. En revanche le *Serveur* choisit des valeurs par défaut et les indique dans le paramètre révisé correspondant.

Il est fortement recommandé par l'OPC UA qu'un *Client* réutilise un *Subscription* après une courte interruption du réseau en activant la *Session* existante sur un nouveau *SecureChannel* comme décrit en 6.5. Si un *Client* a appelé *CreateMonitoredItems* pendant l'interruption du réseau et si l'appel a réussi dans le *Serveur,* mais n'est pas retourné au *Client*, alors le *Client* ne sait pas si l'appel a réussi. Le *Client* peut recevoir des changements de données pour ces éléments surveillés, mais n'est pas capable de les retirer, car il ne connaît pas le *handle* du *Serveur* pour chaque élément surveillé. Il n'est pas non plus possible pour le *Client* de détecter si la création a réussi. Pour supprimer et recréer, utiliser *Subscription* n'est pas non

plus possible, car il peut y avoir plusieurs éléments surveillés fonctionnant normalement qu'il convient de ne pas interrompre. Pour résoudre cette situation, le *Server Object* (objet serveur) fournit une *Method GetMonitoredItems* qui retourne la liste des *handles* du *Client* et du *Serveur* pour les éléments surveillés dans *Subscription*. Cette *Method* est définie dans l'IEC 62541-5.

5.12.2.2 Paramètres

Le Tableau 64 définit les paramètres du *Service*.

Tableau 64 – Paramètres du service CreateMonitoredItems

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* qui signalera dans un rapport les *Notifications* pour ce *MonitoredItem* (voir 7.13 pour la définition d'*IntegerId*).
timestampsToReturn	Enum TimestampsToReturn	Une énumération qui spécifie les *Attributs* d'horodatage à transmettre pour chaque *MonitoredItem*. L'énumération *TimestampsToReturn* est définie en 7.34. Lors de la surveillance d'*Événements*, cela s'applique seulement aux champs *Evénement* qui sont du type *DataValue*.
itemsToCreate []	MonitoredItemCreateRequest	Une liste des *MonitoredItems* à créer et à affecter au *Subscription* spécifié. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
itemToMonitor	ReadValueId	Identifie un élément dans l'*Espace d'adresses* à surveiller. Pour surveiller afin de détecter des *Événements*, l'élément *attributeId* de la structure *ReadValueId* est l'identificateur de l'*Attribute EventNotifier*. Le type *ReadValueId* est défini en 7.23.
monitoringMode	Enum MonitoringMode	Le mode de surveillance à établir pour le *MonitoredItem*. L'énumération *MonitoringMode* est définie en 7.17.
requestedParameters	MonitoringParameters	Les paramètres de surveillance demandés. Les *Serveurs* négocient les valeurs de ces paramètres sur la base du *Subscription* et des fonctions du *Serveur*. Le type *MonitoringParameters* est défini en 7.15.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	MonitoredItemCreateResult	Liste de résultats pour les *MonitoredItems* à créer. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *itemsToCreate*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour le *MonitoredItem* à créer (voir 7.33 pour la définition de *StatusCode*).
monitoredItemId	IntegerId	Identificateur affecté par le *Serveur* pour le *MonitoredItem* (voir 7.13 pour la définition d'*IntegerId*). Cet identificateur est unique dans *Subscription*, mais pourrait ne pas être unique dans le *Serveur* ou dans une *Session*. Ce paramètre est présent seulement si le *statusCode* indique que le *MonitoredItem* a été créé avec succès.
revisedSampling Interval	Duration	L'intervalle d'échantillonnage réel que le *Serveur* utilisera. Cette valeur est fondée sur plusieurs facteurs, y compris les fonctions du système sous-jacent. Le *Serveur* doit toujours retourner un *revisedSamplingInterval* qui est supérieur ou égal au *samplinginterval* demandé. Si le *samplinginterval* demandé est supérieur à l'intervalle d'échantillonnage maximal pris en charge par le *Serveur*, c'est l'intervalle d'échantillonnage maximal qui est retourné.
revisedQueueSize	Counter	La taille réelle de file d'attente que le *Serveur* utilisera.
filterResult	Extensible Parameter MonitoringFilter Result	Contient toutes les éventuelles valeurs de paramètre révisées ou tous les éventuels résultats d'erreur associés au *MonitoringFilter* spécifié dans *requestedParameters*. Ce paramètre peut être omis s'il ne s'est pas produit d'erreurs. Le type de paramètre *MonitoringFilterResult* est un type de paramètre extensible spécifié en 7.16.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *MonitoredItems* à créer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *itemsToCreate*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.12.2.3 Résultats des services

Le Tableau 65 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 65 – Codes de résultats des services CreateMonitoredItems

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TimestampsToReturnInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.12.2.4 StatusCodes

Le Tableau 66 définit les valeurs de niveau opération pour le paramètre *statusCode* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 66 – Codes de résultats de niveau opération de CreateMonitoredItems

Id symbolique	Description
Bad_MonitoringModeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_AttributeIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_IndexRangeNoData	Voir le Tableau 166 pour la description de ce code de résultat. Si les *ArrayDimensions* ont une longueur fixe qui ne peut être modifiée et qu'aucune donnée n'existe dans la plage d'indices spécifiée, Bad_IndexRangeNoData est retourné dans *CreateMonitoredItems*. Sinon, si la longueur de la matrice est dynamique, le *Serveur* doit retourner ce statut dans une réponse *Publish* pour le *MonitoredItem* si aucune donnée n'existe dans la plage.
Bad_DataEncodingInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_DataEncodingUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_MonitoredItemFilterInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_MonitoredItemFilterUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_FilterNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyMonitoredItems	Le *Serveur* a atteint son nombre maximal d'éléments surveillés.

5.12.3 Modifier des éléments surveillés (ModifyMonitoredItems)

5.12.3.1 Description

Ce *Service* est utilisé pour modifier les *MonitoredItems* d'un *Subscription*. Les modifications apportées aux réglages de *MonitoredItem* doivent être appliquées immédiatement par le *Serveur*. Elles prennent effet le plus tôt possible sans toutefois dépasser une durée égale au double du nouveau *revisedSamplingInterval*.

Le retour des informations de réglage des diagnostics dans l'en-tête de demande de *CreateMonitoredItems* ou le dernier *Service ModifyMonitoredItems* est appliqué aux *Monitored Items* (éléments surveillés) et sert de réglages d'informations de diagnostic lors de l'envoi de Notifications dans la réponse *Publish*.

Les valeurs de demande illégales pour les paramètres pouvant être révisés ne génèrent pas d'erreur. En revanche le Serveur choisit des valeurs par défaut et les indique dans le paramètre révisé correspondant.

5.12.3.2 Paramètres

Le Tableau 67 définit les paramètres du *Service*.

Tableau 67 – Paramètres du service ModifyMonitoredItems

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* utilisé pour qualifier le *monitoredItemId* (voir 7.13 pour la définition d'IntegerId).
timestampsToReturn	Enum TimestampsToReturn	Une énumération qui spécifie les *Attributs* d'horodatage à transmettre pour chaque *MonitoredItem* à modifier. L'énumération *TimestampsToReturn* est définie en 7.34. Lors de la surveillance d'*Événements*, cela s'applique seulement aux champs *Evénement* qui sont du type *DataValue*.
itemsToModify []	MonitoredItemModifyRequest	La liste des *MonitoredItems* à modifier. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
monitoredItemId	IntegerId	Identificateur affecté par le *Serveur* pour le *MonitoredItem*.
requestedParameters	Monitoring Parameters	Les valeurs demandées pour les paramètres de surveillance. Le type *MonitoringParameters* est défini en 7.15. Si le nombre de notifications dans la file d'attente dépasse la taille de la nouvelle file d'attente, les notifications en excédent doivent être rejetées selon la politique de rejet configurée.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	MonitoredItemModifyResult	Liste de résultats pour les *MonitoredItems* à modifier. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *itemsToModify*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour le *MonitoredItem* à modifier (voir 7.33 pour la définition de *StatusCode*).
revisedSamplingInterval	Duration	L'intervalle d'échantillonnage réel que le *Serveur* utilisera. Le *Serveur* retourne la valeur qu'il utilisera réellement pour l'intervalle d'échantillonnage. Cette valeur est fondée sur plusieurs facteurs, y compris les fonctions du système sous-jacent. Le *Serveur* doit toujours retourner un *revisedSamplingInterval* qui est supérieur ou égal au *SamplingInterval* demandé. Si le *SamplingInterval* demandé est supérieur à l'intervalle d'échantillonnage maximal pris en charge par le *Serveur*, c'est l'intervalle d'échantillonnage maximal qui est retourné.
revisedQueueSize	Counter	La taille réelle de file d'attente que le *Serveur* utilisera.
filterResult	Extensible Parameter MonitoringFilterResult	Contient toutes les éventuelles valeurs de paramètre révisées ou tous les éventuels résultats d'erreurs associés au *MonitoringFilter* spécifié dans la demande. Ce paramètre peut être omis s'il ne s'est pas produit d'erreurs. Le type de paramètre *MonitoringFilterResult* est un type de paramètre extensible spécifié en 7.16.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *MonitoredItems* à modifier (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *itemsToModify*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.12.3.3 Résultats des services

Le Tableau 68 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 68 – Codes de résultats des services ModifyMonitoredItems

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TimestampsToReturnInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.12.3.4 StatusCodes

Le Tableau 69 définit les valeurs pour le paramètre *statusCode* de niveau opération qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 69 – Codes de résultats de niveau opération de ModifyMonitoredItems

Id symbolique	Description
Bad_MonitoredItemIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_MonitoredItemFilterInvalid	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_MonitoredItemFilterUnsupported	Voir le Tableau 166 pour la description de ce code de résultat.
Bad_FilterNotAllowed	Voir le Tableau 165 pour la description de ce code de résultat.

5.12.4 Gérer des modes de surveillance (SetMonitoringMode)

5.12.4.1 Description

Ce *Service* est utilisé pour établir le mode de surveillance pour un ou plusieurs *MonitoredItems* d'un *Subscription*. Le fait de mettre le mode à DISABLED entraîne la suppression de toutes les *Notifications* placées en file d'attente.

5.12.4.2 Paramètres

Le Tableau 70 définit les paramètres du *Service*.

Tableau 70 – Paramètres du service SetMonitoringMode

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* utilisé pour qualifier les *monitoredItemIds* (voir 7.13 pour la définition d'*IntegerId*).
monitoringMode	Enum MonitoringMode	Le mode de surveillance à établir pour les *MonitoredItems*. L'énumération *MonitoringMode* est définie en 7.17.
monitoredItemIds []	IntegerId	Liste des identificateurs affectés par le *Serveur* pour les *MonitoredItems*.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *MonitoredItems* à activer/désactiver (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *monitoredItemIds*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *MonitoredItems* à activer/désactiver (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *monitoredItemIds*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.12.4.3 Résultats des services

Le Tableau 71 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 71 – Codes de résultats des services SetMonitoringMode

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_MonitoringModeInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.12.4.4 StatusCodes

Le Tableau 72 définit les valeurs de niveau opération pour le paramètre *statusCode* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 72 – Codes de résultats de niveau opération de SetMonitoringMode

Valeur	Description
Bad_MonitoredItemIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.12.5 Gérer des déclenchements (SetTriggering)

5.12.5.1 Description

Ce *Service* est utilisé pour créer et supprimer des liaisons de déclenchement pour un élément déclencheur. L'élément déclencheur et les éléments à rapporter doivent appartenir au même *Subscription*.

Chaque liaison de déclenchement relie un élément déclencheur à un élément à rapporter. Chaque liaison est représentée par l'identificateur de *MonitoredItem* pour l'élément à rapporter. Un code d'erreur est retourné si cet identificateur n'est pas valide.

Voir 5.12.1.6 pour une description du modèle de déclenchement.

5.12.5.2 Paramètres

Le Tableau 73 définit les paramètres du *Service*.

Tableau 73 – Paramètres du service SetTriggering

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour le *Subscription* qui contient l'élément déclencheur et les éléments à rapporter (voir 7.13 pour la définition d'*IntegerId*).
triggeringItemId	IntegerId	Identificateur affecté par le *Serveur* pour le *MonitoredItem* utilisé comme l'élément déclencheur.
linksToAdd []	IntegerId	La liste des identificateurs affectés par le *Serveur* des éléments à rapporter qui sont à ajouter comme liaisons de déclenchement. La liste de *linksToRemove* est traitée avant le *linksToAdd*.
linksToRemove []	IntegerId	La liste des identificateurs affectés par le *Serveur* des éléments à rapporter pour les liaisons de déclenchement à supprimer. La liste de *linksToRemove* est traitée avant le *linksToAdd*.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
addResults []	StatusCode	Liste des *StatusCodes* pour les éléments à ajouter (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre *linksToAdd* spécifié dans la demande.
addDiagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les liaisons à ajouter (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *linksToAdd*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.
removeResults []	StatusCode	Liste des *StatusCodes* pour les éléments à supprimer. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre *linksToRemove* spécifié dans la demande.
removeDiagnosticInfos []	Diagnostic Info	Liste d'informations de diagnostic pour les liaisons à supprimer. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *linksToRemove*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.12.5.3 Résultats des services

Le Tableau 74 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis en 7.33.

Tableau 74 – Codes de résultats des services SetTriggering

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_MonitoredItemIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.12.5.4 StatusCodes

Le Tableau 75 définit les valeurs pour les paramètres results qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 75 – Codes de résultats de niveau opération de SetTriggering

Id symbolique	Description
Bad_MonitoredItemIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.12.6 Supprimer des éléments surveillés (DeleteMonitoredItems)

5.12.6.1 Description

Ce *Service* est utilisé pour retirer un ou plusieurs *MonitoredItems* d'un *Subscription*. Lorsqu'un *MonitoredItem* est supprimé, ses liaisons d'éléments déclenchés sont supprimées aussi.

Le retrait réussi d'un *MonitoredItem* peut toutefois ne pas retirer les *Notifications* pour le *MonitoredItem* qui sont en cours d'envoi par *Subscription*. Par conséquent, les *Clients* peuvent recevoir des *Notifications* pour le *MonitoredItem* après qu'ils aient reçu une réponse positive indiquant que le *MonitoredItem* a été supprimé.

5.12.6.2 Paramètres

Le Tableau 76 définit les paramètres du *Service*.

Tableau 76 – Paramètres du service DeleteMonitoredItems

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* qui contient les *MonitoredItems* à supprimer (voir 7.13 pour la définition d'IntegerId).
monitoredItemIds []	IntegerId	Liste des identificateurs affectés par le *Serveur* pour les *MonitoredItems* à supprimer.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *MonitoredItems* à supprimer (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *monitoredItemIds*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *MonitoredItems* à supprimer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *monitoredItemIds*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.12.6.3 Résultats des services

Le Tableau 77 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 77 – Codes de résultats des services DeleteMonitoredItems

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.12.6.4 StatusCodes

Le Tableau 78 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 78 – Codes de résultats de niveau opération de DeleteMonitoredItems

Id symbolique	Description
Bad_MonitoredItemIdInvalid	Voir le Tableau 166 pour la description de ce code de résultat.

5.13 Jeu de services d'abonnements (Subscription Service Set)

5.13.1 Modèle d'abonnement (SubscriptionModel)

5.13.1.1 Description

Les *Subscriptions* sont utilisés pour rapporter des *Notifications* au *Client*. Leur comportement général est récapitulé ci-après. Leur comportement précis est décrit en 5.13.1.2.

a) Les *Subscriptions* ont un jeu de *MonitoredItems* qui leur sont affectés par le *Client*. Les *MonitoredItems* génèrent des *Notifications* que *Subscription* est tenu de signaler dans un rapport au *Client* (voir 5.12.1 pour une description des *MonitoredItems*).

b) Les *Subscriptions* ont un intervalle d'édition. L'intervalle d'édition d'un *Subscription* définit la fréquence cyclique à laquelle le *Subscription* s'exécute. Chaque fois qu'il s'exécute, il tente d'envoyer un *NotificationMessage* au *Client*. Les *NotificationMessages* contiennent des *Notifications* qui n'ont pas encore été rapportées au *Client*.

c) Les *NotificationMessages* sont envoyés au *Client* en réponse à des demandes *Publish*. Les demandes *Publish* sont normalement placées en file d'attente dans la *Session* à mesure qu'elles sont reçues, et une est sortie de la file d'attente et traitée par un abonnement relatif à cette *Session* à chaque cycle d'édition, s'il y a des *Notifications* à rapporter. Lorsqu'il n'y en a pas, la demande *Publish* est sortie de la file d'attente de la *Session* et le *Serveur* attend le prochain cycle et vérifie de nouveau la présence de *Notifications*.

d) Au début d'un cycle, s'il y a des *Notifications* à envoyer, mais s'il n'y a pas de demandes *Publish* placées en file d'attente, le *Serveur* entre dans l'état d'attente (wait) pour qu'une demande *Publish* soit reçue. Lorsqu'une demande est reçue, elle est traitée immédiatement sans attendre le prochain cycle d'édition.

e) Les *NotificationMessages* sont identifiés de manière unique par des numéros de séquence qui permettent à des *Clients* de détecter des *Messages* manqués. L'intervalle d'édition définit aussi l'intervalle d'échantillonnage par défaut pour ses *MonitoredItems*.

f) Les *Subscriptions* ont un compteur d'entretien qui compte le nombre de cycles d'édition consécutifs dans lesquels il n'y a pas de *Notifications* à rapporter au *Client*. Lorsque le compte d'entretien maximal est atteint, une demande *Publish* est sortie de la file d'attente et utilisée pour retourner le *Message* d'entretien. Ce *Message* d'entretien informe le *Client* que *Subscription* est encore actif. Chaque *Message* d'entretien est une réponse à une demande *Publish* dans laquelle le paramètre *notificationMessage* ne contient pas de *Notifications* et qui contient le numéro de séquence du prochain *NotificationMessage* à envoyer. Dans les articles qui suivent, le terme *NotificationMessage* se réfère à une réponse à une demande *Publish* dans laquelle le paramètre *notificationMessage* contient réellement une ou plusieurs *Notifications*, par opposition à un *keep-alive Message* (Message d'entretien) dans lequel ce paramètre ne contient pas de *Notifications*. Le compte d'entretien maximal est établi par le *Client* pendant la création d'un *Subscription* et peut être modifié ultérieurement en utilisant le *Service ModifySubscription*. D'une manière

similaire au traitement des *Notifications* décrit en c) ci-dessus, s'il n'y a pas de demandes *Publish* placées en file d'attente, le *Serveur* attend la réception de la prochaine demande et envoie immédiatement le message d'entretien sans attendre le prochain cycle d'édition.

g) L'édition par un *Subscription* peut être activée ou désactivée par le *Client* à sa création ou ultérieurement en utilisant le *Service SetPublishingMode*. La désactivation force *Subscription* à cesser d'envoyer des *NotificationMessages* au *Client*. Cependant, *Subscription* continue à s'exécuter de façon cyclique et continue d'envoyer des *Messages* d'entretien au *Client*.

h) Les *Subscriptions* ont un compteur de durée de vie qui compte le nombre de cycles d'édition consécutifs dans lesquels il n'y a pas de demandes *Publish* disponibles pour envoyer une réponse *Publish* pour *Subscription*. Tout appel de *Service* qui utilise le *SubscriptionId* ou le traitement d'une réponse *Publish* réinitialise le compteur de durée de vie de *Subscription*. Lorsque ce compteur atteint la valeur calculée pour la durée de vie d'un *Subscription* en fonction du paramètre MaxKeepAliveCount du *Service CreateSubscription* (5.13.2), *Subscription* est fermé. La fermeture d'un *Subscription* entraîne la suppression de ses *MonitoredItems*. De plus, le *Serveur* doit émettre un *notificationMessage* de *StatusChangeNotification* avec le code de statut Bad_Timeout. Le type du *notificationMessage* de *StatusChangeNotification* est défini en 7.19.4.

i) Les *Sessions* maintiennent une file d'attente de transmission des *NotificationMessages* envoyés. Les *NotificationMessages* sont retenus dans cette file d'attente jusqu'à ce qu'ils soient acquittés. La *Session* doit maintenir une file d'attente de transmission d'une taille au moins égale à deux fois le nombre de demandes *Publish* par *Session* prises en charge par le *Serveur*. La taille minimale de la file d'attente de transmission peut être modifiée par un *Profil* dans l'IEC 62541-7. Le nombre minimal de demandes *Publish* par *Session* prises en charge par le *Serveur* est défini dans l'IEC 62541-7. Il est nécessaire que les *Clients* acquittent les *NotificationMessages* à leur réception. Dans le cas d'un dépassement de capacité de la file d'attente de transmission, le *NotificationMessage* le plus anciennement envoyé est supprimé. Si un *Subscription* est transféré vers une autre *Session*, les *NotificationMessages* placés dans la file d'attente pour ce *Subscription* sont déplacés de l'ancienne *Session* vers la nouvelle.

Le numéro de séquence est un entier non signé de 32 bits qui est incrémenté de 1 pour chaque *NotificationMessage* envoyé. La valeur 0 n'est jamais utilisée comme numéro de séquence. Le premier *NotificationMessage* envoyé sur un *Subscription* a le numéro de séquence 1. Si le numéro de séquence atteint son maximum, il repasse à 1.

Lorsqu'un *Subscription* est créé, le premier *Message* est envoyé à la fin du premier cycle d'édition pour signaler au *Client* que *Subscription* est opérationnel. Un *NotificationMessage* est envoyé s'il y a des *Notifications* prêtes à être rapportées. S'il n'y en a pas, il est envoyé à la place un *Message* d'entretien qui contient un numéro de séquence de 1, indiquant que le premier *NotificationMessage* n'a pas encore été envoyé. C'est la seule fois qu'un *Message* d'entretien est envoyé sans attendre d'avoir atteint le compte d'entretien maximal, comme spécifié en f) ci-dessus.

La valeur du numéro de séquence n'est jamais réinitialisée au cours de la durée de vie d'un *Subscription*. Par conséquent, ce même numéro de séquence ne doit jamais être réutilisé par un *Subscription* avant que plus de quatre milliards de *NotificationMessages* aient été envoyés. À une fréquence continue de mille *NotificationMessages* par seconde sur un *Subscription* donné, il faudrait environ cinquante jours pour que le même numéro de séquence soit réutilisé. Cela permet aux *Clients* de traiter en toute sécurité les numéros de séquence comme étant uniques.

Les numéros de séquence sont également utilisés par les *Clients* pour acquitter réception des *NotificationMessages*. Les demandes *Publish* permettent au *Client* d'acquitter toutes les *Notifications* jusqu'à un numéro de séquence spécifique et permettent d'acquitter le numéro de séquence du dernier *NotificationMessage* reçu. Un ou plusieurs trous peuvent exister entre elles. Les acquittements permettent au *Serveur* d'effacer des *NotificationMessages* de sa file d'attente de retransmission.

Les *Clients* peuvent demander la retransmission de *NotificationMessages* sélectionnés en utilisant le *Service* Republish. Ce *Service* retourne le *Message* demandé.

5.13.1.2 Table d'états

La table d'états décrit le fonctionnement du *Subscription*. Le modèle de fonctionnement des opérations est décrit par cette table d'états. Cette description s'applique lorsque l'édition est activée ou désactivée pour *Subscription*.

Après la création d'un *Subscription*, le *Serveur* lance la minuterie d'édition et la redémarre chaque fois qu'elle expire. Si la minuterie expire le nombre de fois défini pour la durée de vie d'un *Subscription* sans avoir reçu de demande du *Service* de *Subscription* de la part du *Client*, *Subscription* suppose que le *Client* n'est plus présent et se termine.

Les *Clients* envoient des demandes *Publish* aux *Serveurs* pour recevoir des *Notifications*. Les demandes *Publish* ne sont dirigées à aucun *Subscription* spécifique et, par conséquent, peuvent être utilisées par n'importe quel *Subscription*. Chacune contient des acquittements pour un ou plusieurs *Subscriptions*. Ces acquittements sont traités lorsque la demande *Publish* est reçue. Le *Serveur* met alors la demande dans une file d'attente partagée par tous les *Subscriptions*, sauf dans les cas suivants.

a) La réponse *Publish* précédente indiquait qu'il y avait encore d'autres *Notifications* prêtes à être transférées et il n'y avait plus de demandes *Publish* en file d'attente pour les transférer.

b) La minuterie d'édition d'un *Subscription* a expiré et il y avait soit des *Notifications* à envoyer, soit un *Message* d'entretien à envoyer.

Dans ces cas, la demande *Publish* nouvellement reçue est immédiatement traitée par le premier *Subscription* qui rencontrera le cas a) ou le cas b).

Chaque fois que la minuterie d'édition expire, elle est immédiatement réinitialisée. S'il y a des *Notifications* ou un *Message* d'entretien à envoyer, il sort de la file d'attente et traite une demande *Publish*. Lorsqu'un *Subscription* traite une demande *Publish*, il accède aux files d'attente de ses *MonitoredItems* et sort ses éventuelles *Notifications* de la file d'attente. Il retourne ces *Notifications* dans la réponse, positionnant le fanion *moreNotifications* s'il n'a pas été capable de retourner toutes les *Notifications* disponibles dans la réponse.

S'il y avait des *Notifications* ou un *Message* d'entretien à envoyer, mais s'il n'y avait pas de demandes *Publish* en file d'attente, *Subscription* suppose que la demande *Publish* est tardive et attend que la prochaine demande *Publish* soit reçue, comme décrit dans le cas b).

Si *Subscription* est désactivé lorsque la minuterie d'édition expire ou s'il n'y a pas de *Notifications* disponibles, il entre dans l'état d'entretien et met le compteur d'entretien à sa valeur maximale comme défini pour *Subscription*.

Pendant qu'il est dans l'état d'entretien, il recherche des *Notifications* chaque fois que la minuterie d'édition expire. Si une ou plusieurs *Notifications* ont été générées, une demande *Publish* est sortie de la file d'attente et un *NotificationMessage* est retourné dans la réponse. En revanche, si la minuterie d'édition expire sans qu'une *Notification* ne devienne disponible, une demande *Publish* est sortie de la file d'attente et un *Message* d'entretien est retourné dans la réponse. Le *Subscription* revient alors dans son état normal qui est d'attendre que la minuterie d'édition expire de nouveau. Si, dans l'un ou l'autre de ces cas, il n'y a pas de demande *Publish* dans la file d'attente, *Subscription* attend que la prochaine demande *Publish* soit reçue, comme décrit dans le cas b).

Les états de *Subscription* sont définis au Tableau 79.

Tableau 79 – États d'un abonnement

État	Description
CLOSED	Le *Subscription* n'a pas encore été créé ou s'est terminé.
CREATING	Le *Subscription* est en cours de création.
NORMAL	Le *Subscription* vérifie de façon cyclique s'il y a des *Notifications* issues de ses *MonitoredItems*. Le compteur d'entretien n'est pas utilisé dans cet état.
LATE	La minuterie d'édition a expiré et il y a des *Notifications* disponibles ou un *Message* d'entretien prêt à être envoyé, mais il n'y a pas de demandes *Publish* en file d'attente. Dans cet état, la prochaine demande *Publish* est traitée lorsqu'elle est reçue. Le compteur d'entretien n'est pas utilisé dans cet état.
KEEPALIVE	Le *Subscription* vérifie de façon cyclique s'il y a des *Notifications* issues de ses *MonitoredItems* ou si le compteur d'entretien compte à rebours jusqu'à 0 à partir de son maximum.

La table d'états est décrite au Tableau 80. Les règles et conventions suivantes s'appliquent.

a) Les *Événements* représentent la réception de demandes de *Service* et l'occurrence d'*Événements* internes, tels que les expirations de temporisateurs.

b) Les *Événements* de demandes de *Services* peuvent être accompagnés de conditions qui soumettent à essai les valeurs des paramètres de *Service*. Les noms de paramètres commencent par une lettre minuscule.

c) Des *Événements* internes peuvent être accompagnés de conditions qui soumettent à essai les valeurs des *Variables* d'état. Les *Variables* d'état sont définies en 5.13.1.3. Elles commencent par une lettre majuscule.

d) Les *Événements* internes et les demandes de *Services* peuvent être accompagnés de conditions représentées par des fonctions dont la valeur de retour est soumise à essai. Les fonctions sont identifiées par "()" après leur nom. Elles sont décrites en 5.13.1.4.

e) Lorsqu'un *Événement* est reçu, la première transition pour l'état courant est localisée et les transitions sont scrutées séquentiellement pour trouver la première transition qui satisfait aux critères de l'*Événement* ou des conditions. Si aucune transition n'est trouvée, l'*Événement* est ignoré.

f) Les actions sont décrites par des manipulations de fonctions et des *Variables* d'état.

g) L'*Événement LifetimeTimerExpires* est déclenché lorsque son compteur associé atteint zéro.

Tableau 80 – Table d'états d'un abonnement

#	État courant	Événement/Conditions	Action	Prochain état
1	CLOSED	Recevoir la demande CreateSubscription	CreateSubscription()	CREATING
2	CREATING	CreateSubscription échoue	ReturnNegativeResponse()	CLOSED
3	CREATING	CreateSubscription réussit	InitializeSubscription() MessageSent = FALSE ReturnResponse()	NORMAL
4	NORMAL	Recevoir la demande *Publish* && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && MoreNotifications == FALSE))	DeleteAckedNotificationMsgs() EnqueuePublishingReq()	NORMAL
5	NORMAL	Recevoir la demande *Publish* && PublishingEnabled == TRUE && MoreNotifications == TRUE	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
6	NORMAL	PublishingTimer expire && PublishingReqQueued == TRUE && PublishingEnabled == TRUE && NotificationsAvailable == TRUE	ResetLifetimeCounter() StartPublishingTimer() DequeuePublishReq() ReturnNotifications() MessageSent == TRUE	NORMAL
7	NORMAL	PublishingTimer expire && PublishingReqQueued == TRUE && MessageSent == FALSE && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE))	ResetLifetimeCounter() StartPublishingTimer() DequeuePublishReq() ReturnKeepAlive() MessageSent == TRUE	NORMAL
8	NORMAL	PublishingTimer expire && PublishingReqQueued == FALSE && (MessageSent == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == TRUE))	StartPublishingTimer()	LATE
9	NORMAL	PublishingTimer expire && MessageSent == TRUE && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE))	StartPublishingTimer() ResetKeepAliveCounter()	KEEPALIVE
10	LATE	Recevoir la demande *Publish* && PublishingEnabled == TRUE && (NotificationsAvailable == TRUE || MoreNotifications == TRUE)	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnNotifications() MessageSent = TRUE	NORMAL
11	LATE	Recevoir demande *Publish* && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE && MoreNotifications == FALSE))	ResetLifetimeCounter() DeleteAckedNotificationMsgs() ReturnKeepAlive() MessageSent = TRUE	KEEPALIVE
12	LATE	PublishingTimer expire	StartPublishingTimer()	LATE
13	KEEPALIVE	Recevoir la demande *Publish*	DeleteAckedNotificationMsgs() EnqueuePublishingReq()	KEEPALIVE

#	État courant	Événement/Conditions	Action	Prochain état
14	KEEPALIVE	PublishingTimer expire && PublishingEnabled == TRUE && NotificationsAvailable == TRUE && PublishingReqQueued == TRUE	ResetLifetimeCounter() StartPublishingTimer() DequeuePublishReq() ReturnNotifications() MessageSent == TRUE	NORMAL
15	KEEPALIVE	PublishingTimer expire && PublishingReqQueued == TRUE && KeepAliveCounter == 1 && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE)	StartPublishingTimer() DequeuePublishReq() ReturnKeepAlive() ResetKeepAliveCounter()	KEEPALIVE
16	KEEPALIVE	PublishingTimer expire && KeepAliveCounter > 1 && (PublishingEnabled == FALSE || (PublishingEnabled == TRUE && NotificationsAvailable == FALSE))	StartPublishingTimer() KeepAliveCounter--	KEEPALIVE
17	KEEPALIVE	PublishingTimer expire && PublishingReqQueued == FALSE && (KeepAliveCounter == 1 || (KeepAliveCounter > 1 && PublishingEnabled == TRUE && NotificationsAvailable == TRUE))	StartPublishingTimer()	LATE
18	NORMAL || LATE || KEEPALIVE	Recevoir la demande ModifySubscription	ResetLifetimeCounter() UpdateSubscriptionParams() ReturnResponse()	SAME
19	NORMAL || LATE || KEEPALIVE	Recevoir la demande SetPublishingMode	ResetLifetimeCounter() SetPublishingEnabled() MoreNotifications = FALSE ReturnResponse()	SAME
20	NORMAL || LATE || KEEPALIVE	Recevoir la demande Republish && RequestedMessageFound == TRUE	ResetLifetimeCounter() ReturnResponse()	SAME
21	NORMAL || LATE || KEEPALIVE	Recevoir la demande Republish && RequestedMessageFound == FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
22	NORMAL || LATE || KEEPALIVE	Recevoir la demande TransferSubscriptions && SessionChanged() == FALSE	ResetLifetimeCounter() ReturnNegativeResponse ()	SAME
23	NORMAL || LATE || KEEPALIVE	Recevoir la demande TransferSubscriptions && SessionChanged() == TRUE && ClientValidated() ==TRUE	SetSession() ResetLifetimeCounter() ReturnResponse() IssueStatusChangeNotification()	SAME
24	NORMAL || LATE || KEEPALIVE	Recevoir la demande TransferSubscriptions && SessionChanged() == TRUE && ClientValidated() == FALSE	ReturnNegativeResponse()	SAME
25	NORMAL || LATE || KEEPALIVE	Recevoir la demande DeleteSubscriptions && SubscriptionAssignedToClient ==TRUE	DeleteMonitoredItems() DeleteClientPublReqQueue()	CLOSED

#	État courant	Événement/Conditions	Action	Prochain état
26	NORMAL || LATE || KEEPALIVE	Recevoir la demande DeleteSubscriptions && SubscriptionAssignedToClient ==FALSE	ResetLifetimeCounter() ReturnNegativeResponse()	SAME
27	NORMAL || LATE || KEEPALIVE	LifetimeCounter == 1 Le LifetimeCounter est décrémenté si PublishingTimer expire et PublishingReqQueued == FALSE Le LifetimeCounter est réinitialisé si PublishingReqQueued == TRUE.	DeleteMonitoredItems() IssueStatusChangeNotification()	CLOSED

5.13.1.3 Variables d'état et paramètres

Les *Variables* d'état sont définies dans l'ordre alphabétique au Tableau 81.

Tableau 81 – Variables d'état et paramètres

Variable d'état	Description
MoreNotifications	Une valeur booléenne qui est mise à TRUE seulement par la fonction CreateNotificationMsg() lorsqu'il y avait trop de *Notifications* pour un seul *NotificationMessage*.
LatePublishRequest	Une valeur booléenne qui est mise à TRUE pour refléter que, la dernière fois que la minuterie d'édition a expiré, il n'y avait pas de demandes *Publish* en file d'attente.
LifetimeCounter	Une valeur qui contient le nombre d'expirations consécutives de la minuterie d'édition sans activité du *Client* avant que la *Subscription* soit terminée.
MessageSent	Une valeur booléenne qui est mise à TRUE pour signifier que soit un *NotificationMessage,* soit un *Message* d'entretien a été envoyé sur *Subscription*. Il s'agit d'un fanion qui est utilisé pour assurer qu'un *NotificationMessage* ou un *Message* d'entretien est envoyé la première fois que la minuterie d'édition expire.
NotificationsAvailable	Une valeur booléenne qui est mise à TRUE seulement lorsqu'il y a au moins un *MonitoredItem* qui est dans le mode production de rapport et qui a une *Notification* en file d'attente ou lorsqu'il y a au moins un élément à rapporter dont l'élément déclencheur a déclenché et a une *Notification* en file d'attente. La transition de cette *Variable* d'état de FALSE à TRUE crée l'*Événement* «New *Notification* Queued» (Nouvelle notification placée en file d'attente) dans la table d'états.
PublishingEnabled	Le paramètre qui demande que l'édition soit activée ou désactivée.
PublishingReqQueued	Une valeur booléenne qui est mise à TRUE seulement lorsqu'il y a un *Message* de demande *Publish* mis en file d'attente de *Subscription*.
RequestedMessageFound	Une valeur booléenne qui est mise à TRUE seulement lorsque le *Message* dont la retransmission est demandée a été trouvé dans la file d'attente de retransmission.
SeqNum	La valeur qui enregistre la valeur du numéro de séquence utilisé dans les *NotificationMessages*.
SubscriptionAssignedToClient	Une valeur booléenne qui est mise à TRUE seulement lorsque *Subscription* dont la suppression est demandée est affecté au *Client* qui a émis la demande. Un *Subscription* est affecté au *Client* qui l'a créé. Cette affectation peut seulement être changée par l'achèvement réussi du *Service* TransferSubscriptions.

5.13.1.4 Fonctions

Les fonctions d'action sont définies dans l'ordre alphabétique au Tableau 82.

Tableau 82 – Fonctions

Fonction	Description
ClientValidated()	Une fonction booléenne qui retourne TRUE seulement lorsque le *Client* qui présente une demande de TransferSubscriptions opère pour le compte du même utilisateur et prend en charge les mêmes *Profiles* que le *Client* de la *Session* précédente.
CreateNotificationMsg()	Incrémenter le SeqNum et créer un *NotificationMessage* à partir des *MonitoredItems* affectés à *Subscription*. Sauvegarder dans la file d'attente de retransmission le *NotificationMessage* nouvellement créé. Si toutes les *Notifications* disponibles peuvent être envoyées dans la réponse *Publish*, la *Variable* d'état MoreNotifications est mise à FALSE. Autrement, elle est mise à TRUE.
CreateSubscription()	Tenter de créer un *Subscription*.
DeleteAckedNotificationMsgs()	Supprimer de la file d'attente de retransmission les *NotificationMessages* qui ont été acquittés par la demande.
DeleteClientPublReqQueue()	Vider la file d'attente des demandes *Publish* pour le *Client* qui envoie la demande DeleteSubscriptions, s'il n'y a plus de *Subscriptions* affectés au *Client* en question.
DeleteMonitoredItems()	Supprimer tous les *MonitoredItems* affectés à la *Subscription*.
DequeuePublishReq()	Sortir de la file d'attente une demande d'édition selon la règle du premier entré premier sorti. Valider si la demande d'édition est encore valide en vérifiant le timeoutHint dans le RequestHeader. Si la demande a dépassé le délai, envoyer un résultat de service *Bad_Timeout* pour la demande et sortir de la file d'attente une autre demande d'édition. ResetLifetimeCounter()
EnqueuePublishingReq()	Placer en file d'attente la demande d'édition.
InitializeSubscription()	ResetLifetimeCounter() MoreNotifications = FALSE PublishRateChange = FALSE PublishingEnabled = valeur du paramètre publishingEnabled dans la demande CreateSubscription PublishingReqQueued = FALSE SeqNum = 0 SetSession() StartPublishingTimer()
IssueStatusChangeNotification()	Émettre un *notificationMessage StatusChangeNotification* avec un code de statut pour le changement de statut du *Subscription*. Le type *notificationMessage StatusChangeNotification* est défini en 7.19.4. Le code de statut Bad_Timeout est utilisé si la durée de vie expire et Good_SubscriptionTransferred est utilisé si les Subscriptions ont été transférés à une autre Session.
ResetKeepAliveCounter()	Réinitialiser le compteur d'entretien au compte d'entretien maximal de *Subscription*. Le compte d'entretien maximal est établi par le *Client* lorsque *Subscription* est créé et peut être modifié en utilisant le *Service* ModifySubscription.
ResetLifetimeCounter()	Réinitialiser la *Variable* LifetimeCounter à la valeur spécifiée pour la durée de vie d'un *Subscription* dans le *Service* CreateSubscription (5.13.2).
ReturnKeepAlive()	CreateKeepAliveMsg() ReturnResponse()
ReturnNegativeResponse ()	Retourner une réponse *Service* indiquant l'erreur de niveau *Service* appropriée. Il n'est retourné aucun paramètre autre que le responseHeader qui contient le *StatusCode* de niveau *Service*.

ReturnNotifications()	CreateNotificationMsg()
	ReturnResponse()
	If (MoreNotifications == TRUE) && (PublishingReqQueued == TRUE)
	{
	DequeuePublishReq()
	Boucler de nouveau à travers cette fonction.
	}

ReturnResponse()	Retourner la réponse appropriée, établissant les valeurs appropriées de paramètres *StatusCodes* définis pour le *Service*.
SessionChanged()	Une fonction booléenne qui retourne TRUE seulement lorsque la *Session* utilisée pour envoyer une demande TransferSubscriptions est différente de la *Session Client* actuellement associée à *Subscription*.
SetPublishingEnabled ()	Mettre la *Variable* d'état PublishingEnabled à la valeur du paramètre publishingEnabled reçu dans la demande.
SetSession	Établir les informations de *Session* pour que *Subscription* corresponde à la *Session* sur laquelle la demande TransferSubscriptions a été émise.
StartPublishingTimer()	Démarrer ou redémarrer la minuterie d'édition et décrémenter la *Variable* LifetimeCounter.
UpdateSubscriptionParams()	Négocier et mettre à jour les paramètres de *Subscription*. Si le nouvel intervalle d'entretien est inférieur à la valeur courante du compteur d'entretien, exécuter les fonctions ResetKeepAliveCounter() et ResetLifetimeCounter().

5.13.2 Créer un abonnement (CreateSubscription)

5.13.2.1 Description

Ce *Service* est utilisé pour créer un *Subscription*. Les *Subscriptions* surveillent un jeu de *MonitoredItems* pour détecter des *Notifications* et les retourner au *Client* en réponse aux demandes *Publish*.

Les valeurs de demande illégales pour les paramètres pouvant être révisés ne génèrent pas d'erreur. En revanche le *Serveur* choisit des valeurs par défaut et les indique dans le paramètre révisé correspondant.

5.13.2.2 Paramètres

Le Tableau 83 définit les paramètres du *Service*.

Tableau 83 – Paramètres du service CreateSubscription

Nom	Type	Description
Request		
requestHeader	Request Header	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
requestedPublishing Interval	Duration	Cet intervalle définit la fréquence cyclique à laquelle il est demandé à *Subscription* de retourner des *Notifications* au *Client*. Cet intervalle est exprimé en millisecondes. Cet intervalle est représenté par la minuterie d'édition dans la table d'états de *Subscription* (voir 5.13.1.2). La valeur négociée pour ce paramètre retournée dans la réponse est utilisée comme intervalle d'échantillonnage par défaut pour les *MonitoredItems* affectés à ce *Subscription*. Si la valeur demandée est 0 ou négative, le *Serveur* doit réviser avec l'intervalle d'édition pris en charge le plus rapide.
requestedLifetimeCount	Counter	Compte de durée de vie demandé (voir 7.5 pour la définition de *Counter*). Le compte de durée de vie doit être au minimum égal à trois fois le compte d'entretien. Lorsque la minuterie d'édition a expiré ce nombre de fois sans qu'une demande *Publish* soit disponible pour envoyer un *NotificationMessage*, *Subscription* doit être supprimé par le *Serveur*.
requestedMaxKeepAlive Count	Counter	Compte d'entretien maximal demandé (voir 7.5 pour la définition de *Counter*). Lorsque la minuterie d'édition a expiré ce nombre de fois sans exiger qu'un moindre *NotificationMessage* soit envoyé, *Subscription* envoie un *Message* d'entretien au *Client*. La valeur négociée pour ce paramètre est retournée dans la réponse. Si la valeur demandée est 0, le *Serveur* doit réviser avec le compte d'entretien le plus petit pris en charge.
maxNotificationsPerPublish	Counter	Le nombre maximal des notifications que le *Client* souhaite recevoir dans une seule réponse *Publish*. Une valeur de zéro indique qu'il n'y a pas de limite. Le nombre de notifications par *Publish* est la somme de monitoredItems dans la DataChangeNotification et d'événements dans l'EventNotificationList.
publishingEnabled	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE l'édition est activée pour *Subscription.* FALSE l'édition est désactivée pour *Subscription*. La valeur de ce paramètre n'altère pas la valeur de l'*Attribut* de mode de surveillance des *MonitoredItems*.
priority	Byte	Indique la priorité relative de *Subscription*. Lorsqu'il est nécessaire que plus d'un *Subscription* envoie des *Notifications*, il convient que le *Serveur* sorte de la file d'attente une demande Publish adressée à *Subscription* ayant le nombre *priority* le plus élevé. Pour les *Subscriptions* d'égale *priority*, il convient que le *Serveur* sorte de la file d'attente les demandes Publish en «round-robin» (chacune à tour de rôle). Il convient qu'un Client qui n'exige pas de réglages de priorité spéciaux mette cette valeur à zéro.
Response		
responseHeader	Response Header	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour le *Subscription* (voir 7.13 pour la définition d'*IntegerId*). Cet identificateur doit être unique pour le *Serveur* tout entier, pas uniquement pour la *Session*, afin de permettre le transfert d'un *Subscription* vers une autre *Session* au moyen du service TransferSubscriptions.

Nom	Type	Description
Request		
revisedPublishingInterval	Duration	L'intervalle d'édition réel que le *Serveur* utilisera, exprimé en millisecondes. Il convient que le *Serveur* essaie d'honorer la demande du *Client* pour ce paramètre, mais il peut négocier cette valeur à la hausse ou à la baisse pour satisfaire à ses propres contraintes.
revisedLifetimeCount	Counter	La durée de vie d'un *Subscription* doit être au moins égale à trois fois l'intervalle d'entretien négocié par le *Serveur.*
revisedMaxKeepAliveCount	Counter	Le compte d'entretien maximal réel. (voir 7.5 pour la définition de *Counter*). Il convient que le *Serveur* essaie d'honorer la demande du *Client* pour ce paramètre, mais il peut négocier cette valeur à la hausse ou à la baisse pour satisfaire à ses propres contraintes.

5.13.2.3 Résultats des services

Le Tableau 84 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 84 – Codes de résultats des services CreateSubscription

Id symbolique	Description
Bad_TooManySubscriptions	Le *Serveur* a atteint le nombre maximal d'abonnements.

5.13.3 Modifier un abonnement (ModifySubscription)

5.13.3.1 Description

Ce *Service* est utilisé pour modifier un *Subscription*.

Les valeurs de demande illégales pour les paramètres pouvant être révisés ne génèrent pas d'erreur. En revanche le Serveur choisit des valeurs par défaut et les indique dans le paramètre révisé correspondant.

Les modifications apportées aux réglages de *Subscription* doivent être appliquées immédiatement par le *Serveur*. Elles prennent effet le plus tôt possible sans toutefois dépasser une durée égale au double du nouveau *revisedPublishingInterval.*

5.13.3.2 Paramètres

Le Tableau 85 définit les paramètres du *Service*.

Tableau 85 – Paramètres du service ModifySubscription

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* (voir 7.13 pour la définition d'*IntegerId*).
requestedPublishingInterval	Duration	Cet intervalle définit la fréquence cyclique à laquelle il est demandé au *Subscription* de retourner des *Notifications* au *Client*. Cet intervalle est exprimé en millisecondes. Cet intervalle est représenté par la minuterie d'édition dans la table d'états de *Subscription* (voir 5.13.1.2). La valeur négociée pour ce paramètre retournée dans la réponse est utilisée comme intervalle d'échantillonnage par défaut pour les *MonitoredItems* affectés à ce *Subscription*. Si la valeur demandée est 0 ou négative, le *Serveur* doit réviser l'intervalle d'édition le plus rapide pris en charge.
requestedLifetimeCount	Counter	Compte de durée de vie demandé (voir 7.5 pour la définition de *Counter*). Le compte de durée de vie doit être au minimum égal à trois fois le compte d'entretien. Lorsque la minuterie d'édition a expiré ce nombre de fois sans qu'une demande *Publish* soit disponible pour envoyer un *NotificationMessage*, *Subscription* doit être supprimé par le *Serveur*.
requestedMaxKeepAliveCount	Counter	Le compte d'entretien maximal demandé. (voir 7.5 pour la définition de *Counter*). Lorsque la minuterie d'édition a expiré ce nombre de fois

Nom	Type	Description
Request		
		sans exiger qu'un moindre *NotificationMessage* soit envoyé, *Subscription* envoie un *Message* d'entretien au *Client*. La valeur négociée pour ce paramètre est retournée dans la réponse. Si la valeur demandée est 0, le *Serveur* doit réviser avec le compte d'entretien le plus petit pris en charge.
maxNotificationsPerPublish	Counter	Le nombre maximal des notifications que le *Client* souhaite recevoir dans une seule réponse *Publish*. Une valeur de zéro indique qu'il n'y a pas de limite.
Priority	Byte	Indique la priorité relative de *Subscription*. Lorsqu'il est nécessaire que plus d'un *Subscription* envoie des *Notifications*, il convient que le *Serveur* sorte de la file d'attente une demande Publish adressée au *Subscription* ayant le nombre *priority* le plus élevé. Pour les *Subscriptions* d'égale *priority*, il convient que le *Serveur* sorte de la file d'attente les demandes Publish en «round-robin» (chacune à tour de rôle). Tout *Subscription* qui a besoin d'envoyer un *Message* d'entretien doit avoir la préséance quelle que soit sa *priority*, afin d'éviter que *Subscription* n'expire. Il convient qu'un *Client* qui n'exige pas de réglages de priorité spéciaux mette cette valeur à zéro.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
revisedPublishingInterval	Duration	L'intervalle d'édition réel que le *Serveur* utilisera, exprimé en millisecondes. Il convient que le *Serveur* essaie d'honorer la demande du *Client* pour ce paramètre, mais il peut négocier cette valeur à la hausse ou à la baisse pour satisfaire à ses propres contraintes.
revisedLifetimeCount	Counter	La durée de vie de *Subscription* doit être au moins égale à trois fois l'intervalle d'entretien négocié avec le *Serveur*.
revisedMaxKeepAliveCount	Counter	Le compte d'entretien maximal réel (voir 7.5 pour la définition de *Counter*). Il convient que le *Serveur* essaie d'honorer la demande du *Client* pour ce paramètre, mais il peut négocier cette valeur à la hausse ou à la baisse pour satisfaire à ses propres contraintes.

5.13.3.3 Résultats des services

Le Tableau 86 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 86 – Codes de résultats des services ModifySubscription

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.13.4 Demander un mode d'édition (SetPublishingMode)

5.13.4.1 Description

Ce *Service* est utilisé pour permettre l'envoi de *Notifications* sur un ou plusieurs *Subscriptions*.

5.13.4.2 Paramètres

Le Tableau 87 définit les paramètres du *Service*.

Tableau 87 – Paramètres du service SetPublishingMode

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
publishingEnabled	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE l'édition de *NotificationMessages* est activée pour *Subscription*. FALSE l'édition de *NotificationMessages* est désactivée pour *Subscription*. La valeur de ce paramètre n'altère pas la valeur de l'*Attribut* de mode de surveillance des *MonitoredItems*. La mise de cette valeur à FALSE n'interrompt

Nom	Type	Description
		pas l'envoi de *Messages* d'entretien.
subscriptionIds []	IntegerId	Liste d'identificateurs affectés par le *Serveur* pour les *Subscriptions* à activer ou désactiver (voir 7.13 pour la définition d'*IntegerId*).
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *Subscriptions* à activer/désactiver (voir 7.33 pour la définition des *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Subscriptions* à activer/désactiver (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.13.4.3 Résultats des services

Le Tableau 88 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 88 – Codes de résultats des services SetPublishingMode

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.13.4.4 StatusCodes

Le Tableau 89 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 89 – Codes de résultats de niveau opération de SetPublishingMode

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

5.13.5 Editer (Publish)

5.13.5.1 Description

Ce *Service* est utilisé à deux fins. Premièrement, il est utilisé pour acquitter réception des *NotificationMessages* pour un ou plusieurs *Subscriptions*. Deuxièmement, il est utilisé pour demander au *Serveur* de retourner un *NotificationMessage* ou un *Message* d'entretien. Comme les demandes *Publish* ne sont pas dirigées à un *Subscription* spécifique, elles peuvent être utilisées par n'importe quel *Subscription*. Le paragraphe 5.13.1.2 décrit l'utilisation du *Service Publish*.

Les stratégies du *Client* pour émettre des demandes *Publish* peuvent varier en fonction des retards réseau entre le *Client* et le *Serveur*. Dans de nombreux cas, le *Client* peut souhaiter émettre une demande *Publish* immédiatement après avoir créé un *Subscription* et, par la suite, immédiatement après avoir reçu une réponse *Publish*.

Dans d'autres cas, notamment dans les réseaux à temps d'attente élevée, le *Client* peut souhaiter placer en cascade les demandes *Publish* pour assurer une production cyclique de rapports par le *Serveur*. Le placement en cascade implique d'envoyer plus d'une demande *Publish* pour chaque *Subscription* avant de recevoir une réponse. Par exemple, si le réseau introduit entre le *Client* et le *Serveur* un retard de 5 secondes et l'intervalle d'édition pour une *Subscription* est 1 seconde le *Client* aura à émettre des demandes *Publish* toutes les secondes au lieu d'attendre qu'une réponse soit reçue avant d'envoyer la prochaine demande.

Il convient qu'un *Serveur* limite le nombre de demandes *Publish* actives afin d'éviter que le nombre soit infini, car les demandes *Publish* sont censées être placées en file d'attente dans le *Serveur*. Mais un *Serveur* doit accepter plus de demandes *Publish* mises en file d'attente que d'*Abonnements* créés. Il est attendu qu'un *Serveur* prenne en charge plusieurs demandes *Publish* par *Subscription*. Lorsqu'un *Serveur* reçoit une nouvelle demande *Publish* qui dépasse sa limite, il doit sortir de la file d'attente la demande *Publish* la plus ancienne et retourner une réponse avec le résultat mis à *Bad_TooManyPublishRequests*. Si le *Client* reçoit ce *Service* pour une demande *Publish*, il ne doit pas émettre d'autre demande *Publish* avant que l'une de ses demandes *Publish* en cours ne soit retournée du *Serveur*.

Les *Clients* peuvent limiter la taille des réponses *Publish* avec le paramètre *maxNotificationsPerPublish* passé au *Service CreateSubscription*. Cependant, cela peut encore produire un message trop volumineux pour être traité par le *Client* ou le *Serveur*. Dans cette situation, le *Client* s'apercevra soit que le *Canal Sécurisé* passe à un état de défaut et a besoin d'être rétabli, soit que la réponse *Publish* retourne une erreur et l'appel du *Service Republish* retourne aussi une erreur. Si l'une de ces situations se produit, le *Client* aura à ajuster ses limites de traitement de message ou ses paramètres pour *Subscription* et/ou les *MonitoredItems*.

Le réglage des informations de diagnostic de retour dans l'en-tête de demande des *CreateMonitoredItems* ou le dernier *Service ModifyMonitoredItems* est appliqué aux *Monitored Items* (Éléments surveillés) et sert de réglages d'informations de diagnostic lors de l'envoi de Notifications dans la réponse *Publish*.

5.13.5.2 Paramètres

Le Tableau 90 définit les paramètres du *Service*.

Tableau 90 – Paramètres du service Publish

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscription Acknowledgements []	Subscription Acknowledgement	La liste des acquittements pour un ou plusieurs *Subscriptions*. Cette liste peut contenir plusieurs acquittements pour le même *Subscription* (plusieurs entrées ayant le même *subscriptionId*). Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour un *Subscription* (voir 7.13 pour la définition d'*IntegerId*).
sequenceNumber	Counter	Le numéro de séquence acquitté (voir 7.5 pour la définition de *Counter*). Le *Serveur* peut supprimer de sa file d'attente de retransmission le *Message* ayant ce numéro de séquence.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* pour lequel les *Notifications* sont retournées (voir 7.13 pour la définition d'*IntegerId)*. La valeur 0 est utilisée pour indiquer qu'il n'y avait pas de *Subscriptions* définis pour lesquels une réponse peut être envoyée.
availableSequence Numbers []	Counter	Une liste de plages de numéros de séquence qui identifient des *NotificationMessages* non acquittés qui sont disponibles pour la retransmission à partir de la file d'attente de retransmission d'un *Subscription*. Cette liste est élaborée après traitement des acquittements dans la réponse (voir 7.5 pour la définition de *Counter)*.
moreNotifications	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE le nombre des *Notifications* qui étaient prêtes à être envoyées n'a pas pu être envoyé dans une seule réponse. FALSE toutes les *Notifications* qui étaient prêtes sont incluses dans la réponse.
notificationMessage	Notification Message	Le *NotificationMessage* qui contient la liste des *Notifications*. Le type de paramètre *NotificationMessage* est spécifié en 7.20.
results []	StatusCode	Liste de résultats pour les acquittements (voir 7.33 pour la définition de *StatusCode*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionAcknowledgements*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les acquittements (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionAcknowledgements*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.13.5.3 Résultats des services

Le Tableau 91 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 91 – Codes de résultats des services Publish

Id symbolique	Description
Bad_TooManyPublishRequests	Le serveur a atteint le nombre maximal de demandes d'édition placées en file d'attente.
Bad_NoSubscription	Aucun abonnement n'est disponible pour cette session.

5.13.5.4 StatusCodes

Le Tableau 92 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 92 – Codes de résultats de niveau opération de Publish

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_SequenceNumberUnknown	Le numéro de séquence est inconnu du serveur.

5.13.6 Éditer à nouveau (Republish)

5.13.6.1 Description

Ce *Service* demande au *Subscription* de rééditer un *NotificationMessage* à partir de sa file d'attente de retransmission. Si le *Serveur* n'a pas le *Message* demandé dans sa file d'attente de retransmission, il retourne une réponse d'erreur.

Voir 5.13.1.2 pour la description détaillée du comportement de ce *Service*.

Voir 6.5 pour une description des problèmes et des stratégies de la gestion de reconnexion et de *Republish*.

5.13.6.2 Paramètres

Le Tableau 93 définit les paramètres du *Service*.

Tableau 93 – Paramètres du service Republish

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionId	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* à rééditer (voir 7.13 pour la définition d'*IntegerId*).
retransmitSequence Number	Counter	Le numéro de séquence d'un *NotificationMessage* spécifique à rééditer (voir 7.5 pour la définition de *Counter*).
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
notificationMessage	Notification Message	Le *NotificationMessage* demandé. Le type de paramètre *NotificationMessage* est spécifié en 7.18.

5.13.6.3 Résultats des services

Le Tableau 94 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 94 – Codes de résultats des services Republish

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_MessageNotAvailable	Le message demandé n'est plus disponible.

5.13.7 Transférer un abonnement (TransferSubscriptions)

5.13.7.1 Description

Ce *Service* est utilisé pour transférer un *Subscription* et ses *MonitoredItems* d'une *Session* à l'autre. Par exemple, un *Client* peut avoir besoin de rouvrir une *Session* et ensuite de transférer ses *Subscriptions* à la *Session* en question. Il peut aussi être utilisé par un *Client* pour prendre un *Subscription* à un autre *Client* en transférant le *Subscription* à sa *Session*.

L'*authenticationToken* contenu dans l'en-tête de demande identifie la *Session* à laquelle le *Subscription* et les *MonitoredItems* doivent être transférés. Le *Serveur* doit valider que le *Client* de la *Session* en question opère pour le compte du même utilisateur et que le *Client* potentiellement nouveau prend en charge les *Profiles* qui sont nécessaires pour le *Subscription*. Si le *Serveur* transfère le *Subscription*, il retourne les numéros de séquence des *NotificationMessages* qui sont disponibles pour une retransmission. Il convient que le *Client* acquitte tous les *Messages* de la liste pour lesquels il ne demandera pas de retransmission.

Si le *Serveur* transfère le *Subscription* à la nouvelle *Session*, le *Serveur* doit émettre un *notificationMessage StatusChangeNotification* avec le code de statut Good_SubscriptionTransferred à destination de l'ancienne *Session*. Le type *notificationMessage StatusChangeNotification* est défini en 7.19.4.

5.13.7.2 Paramètres

Le Tableau 95 définit les paramètres du *Service*.

Tableau 95 – Paramètres du service TransferSubscriptions

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionIds []	IntegerId	Liste d'identificateurs pour les *Subscriptions* à transférer au nouveau *Client* (voir 7.13 pour la définition d'*IntegerId*). Ces identificateurs sont transférés du *Client* principal vers un *Client* de secours via des mécanismes externes.
sendInitialValues	Boolean	Un paramètre *Boolean* ayant les valeurs suivantes: TRUE la première réponse *Publish* après l'appel de TransferSubscriptions doit contenir les valeurs courantes de tous les *Monitored Items* (éléments surveillés) dans l'Abonnement dans lequel le Mode "Monitoring" est mis à "Reporting". FALSE la première réponse Publish après l'appel de TransferSubscriptions doit seulement contenir les changements de valeur depuis que la dernière réponse Publish a été envoyée. Ce paramètre ne s'applique qu'aux MonitoredItems utilisés pour surveiller les modifications d'Attribut.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	TransferResult	Liste de résultats pour les *Subscriptions* à transférer. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*. Cette structure est définie de façon cohérente par rapport aux éléments prévus suivants.
statusCode	StatusCode	*StatusCode* pour chaque *Subscription* à transférer (voir 7.33 pour la définition de *StatusCode*).
availableSequence Numbers []	Counter	Une liste de plages de numéros de séquence qui identifie des *NotificationMessages* qui sont dans la file d'attente de retransmission d'un *Subscription*. Ce paramètre est null si le transfert d'un *Subscription* a échoué. Le type *Counter* est défini en 7.5.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Subscriptions* à transférer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.13.7.3 Résultats des services

Le Tableau 96 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 96 – Codes de résultats des services TransferSubscriptions

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_InsufficientClientProfile	Le *Client* de la *Session* courante ne prend pas en charge un ou plusieurs *Profiles* nécessaires à un *Subscription*.

5.13.7.4 StatusCodes

Le Tableau 97 définit les valeurs de niveau opération pour le paramètre *statusCode* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 97 – Codes de résultats de niveau opération de TransferSubscriptions

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_UserAccessDenied	Voir le Tableau 165 pour la description de ce code de résultat. Le *Client* de la *Session* courante n'opère pas pour le compte du même utilisateur que la *Session* qui est propriétaire d'un *Subscription*.

5.13.8 Supprimer des abonnements (DeleteSubscriptions)

5.13.8.1 Description

Ce *Service* est invoqué pour supprimer un ou plusieurs *Subscriptions* qui appartiennent à la *Session* du *Client*.

L'achèvement réussi de ce *Service* entraîne la suppression de tous les *MonitoredItems* qui utilisent *Subscription*. S'il s'agit du dernier *Subscription* pour la *Session*, alors toutes les demandes *Publish* encore dans la file d'attente pour cette *Session* sont sorties de la file d'attente et doivent être retournées avec Bad_NoSubscription.

Les *Subscriptions* transférés à une autre *Session* doivent être supprimés par le *Client* qui est propriétaire de la *Session*.

5.13.8.2 Paramètres

Le Tableau 98 définit les paramètres du *Service*.

Tableau 98 – Paramètres du service DeleteSubscriptions

Nom	Type	Description
Request		
requestHeader	RequestHeader	Paramètres communs aux demandes (voir 7.26 pour la définition de *RequestHeader*).
subscriptionIds []	IntegerId	L'identificateur affecté par le *Serveur* pour *Subscription* ((voir 7.13 pour la définition d'*IntegerId)*.
Response		
responseHeader	ResponseHeader	Paramètres communs aux réponses (voir 7.27 pour la définition de *ResponseHeader*).
results []	StatusCode	Liste des *StatusCodes* pour les *Subscriptions* à supprimer (voir 7.33 pour la définition de *StatusCode)*. La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*.
diagnosticInfos []	DiagnosticInfo	Liste d'informations de diagnostic pour les *Subscriptions* à supprimer (voir 7.8 pour la définition de *DiagnosticInfo*). La taille et l'ordre de la liste correspondent à la taille et à l'ordre du paramètre de demande *subscriptionIds*. Cette liste est vide si les informations de diagnostic n'ont pas été demandées dans l'en-tête de demande ou si aucune information de diagnostic n'a été rencontrée dans le traitement de la demande.

5.13.8.3 Résultats des services

Le Tableau 99 définit les résultats des *Services* spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 165.

Tableau 99 – Codes de résultats des services DeleteSubscriptions

Id symbolique	Description
Bad_NothingToDo	Voir le Tableau 165 pour la description de ce code de résultat.
Bad_TooManyOperations	Voir le Tableau 165 pour la description de ce code de résultat.

5.13.8.4 StatusCodes

Le Tableau 100 définit les valeurs pour le paramètre *results* qui sont spécifiques à ce *Service*. Les *StatusCodes* communs sont définis au Tableau 166.

Tableau 100 – Codes de résultats de niveau opération de DeleteSubscriptions

Id symbolique	Description
Bad_SubscriptionIdInvalid	Voir le Tableau 165 pour la description de ce code de résultat.

6 Comportements des services

6.1 Sécurité

6.1.1 Vue d'ensemble

Les *Services* OPC UA définissent un certain nombre de mécanismes pour satisfaire aux exigences de sécurité décrites dans les grandes lignes dans l'IEC TR 62541-2. Cet article décrit un certain nombre d'importantes procédures de sécurité que les *Applications OPC UA* doivent suivre.

6.1.2 Obtenir et installer un certificat d'instance d'application (Application Instance Certificate)

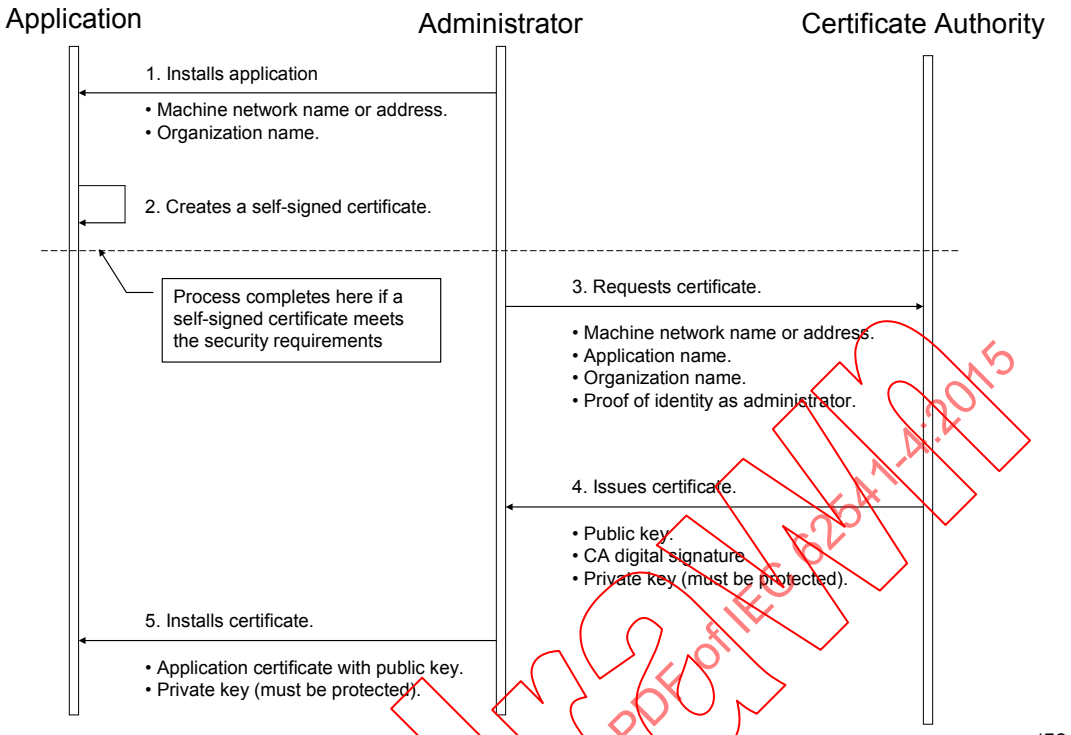
Toutes les *Applications OPC UA* exigent un *Certificat d'Instance d'Application* qui doit contenir les renseignements suivants:

- Le nom du réseau ou l'adresse de l'ordinateur sur lequel l'application s'exécute;
- Le nom de l'organisation qui gère ou est le propriétaire de l'application;
- Le nom de l'application;
- L'URI de l'instance d'application;
- Le nom de l'*Autorité de Certification* qui a émis le *Certificat*;
- La date d'émission et d'expiration du *Certificat*;
- La clé publique émise pour l'application par l'*Autorité de Certification* (CA, *certificate authority*);
- Une signature numérique créée par l'*Autorité de Certification* (CA).

En outre, chaque *Certificat d'Instance d'Application* a une clé privée qu'il convient de conserver dans un endroit uniquement accessible par l'application. Si l'intégrité de cette clé privée est compromise, l'administrateur doit affecter un nouveau *Certificat d'Instance d'Application* et une nouvelle clé privée à l'application.

Ce *Certificat* peut être généré automatiquement lorsque l'application est installée. Dans cette situation, la clé privée affectée au *Certificat* doit être utilisée pour créer la signature du *Certificat*. Les certificats créés de cette manière sont appelés *Certificats* autosignés.

Si l'administrateur responsable de l'application décide qu'un *Certificat* autosigné ne satisfait pas aux exigences de sécurité de l'organisation, il convient que l'administrateur installe un *Certificat* émis par une *Autorité de Certification*. Les étapes impliquées dans la demande du *Certificat d'Instance d'Application* auprès d'une *Autorité de Certification* sont montrées à la Figure 19.



Anglais	Français
Administrator	Administrateur
Certificate authority	Autorité de certification
Installs application	Installe l'application
Machine network name or address	Nom ou adresse réseau de la machine
Organization name	Nom d'organisation
Creates a self-signed certificate	Crée un certificat autosigné
Process completes here if a self-signed certificate meets the security requirements	Le processus se termine ici si un certificat autosigné satisfait aux exigences relatives à la sécurité
Requests certificate	Demande un certificat
Application name	Nom de l'application
Proof of identity as administrator	Preuve de l'identité comme administrateur
Issues certificate	Émet un certificat
Installs certificate	Installe un certificat
Application certificate with public key	Certificat d'application avec clé publique
Private key (must be protected)	Clé privée (doit être protégée)
Public key	Clé publique
CA digital signature	Signature numérique de CA

Figure 19 – Obtenir et installer un certificat d'instance d'application

La figure ci-dessus illustre les interactions entre l'application, l'*Administrateur* et l'*Autorité de Certification*. L'*Application* est en tant qu'*Application OPC UA* installée sur une seule machine. L'*Administrateur* est la personne chargée de gérer la machine et *l'Application OPC UA*. L'*Autorité de Certification* est une entité qui peut émettre des *Certificats* numériques qui satisfont aux exigences de l'organisation qui déploie l'*Application OPC UA*.

Si l'*Administrateur* décide qu'un *Certificat* autosigné satisfait aux exigences de sécurité pour l'organisation, l'*Administrateur* peut sauter les Étapes 3 à 5. Les fournisseurs d'application

doivent toujours créer un *Certificat* autosigné par défaut au cours du processus d'installation. Chaque *Application OPC UA* doit permettre aux *Administrateurs* de remplacer des *Certificats d'instance d'application* par des *Certificats* qui satisfont à leurs exigences.

Lorsque l'*Administrateur* demande un nouveau *Certificat* auprès d'une *Autorité de Certification*, l'*Autorité de Certification* peut exiger que l'*Administrateur* fournisse la preuve de l'autorisation de demander des *Certificats* pour l'organisation qui sera propriétaire du *Certificat*. Le mécanisme exact utilisé pour fournir cette preuve dépend de l'*Autorité de Certification*.

Les fournisseurs peuvent choisir d'automatiser le processus d'acquisition de *Certificats* auprès d'une autorité. Si tel est le cas, l'*Administrateur* passe toujours par les étapes illustrées à la Figure 19. En revanche, le programme d'installation pour l'application le fait automatiquement et sollicite seulement l'*Administrateur* pour qu'il fournisse les renseignements relatifs à l'instance d'application installée.

6.1.3 Déterminer si un Certificat est Fiable

Les *Applications* ne doivent jamais communiquer avec une autre application en laquelle elles n'ont pas confiance. Une *Application* décide qu'une autre application est fiable en vérifiant si le *Certificat d'instance d'application* de l'autre application est fiable. Les applications doivent s'appuyer sur les listes de *Certificat*s fournies par l'*Administrateur* pour déterminer la confiance. Il existe deux listes distinctes: une liste d'*Applications* fiables et une liste d'*Autorités de certification* (CA) fiables. Si une application n'est pas directement fiable (c'est-à-dire que son *Certificat* ne figure pas dans la liste d'applications fiables), l'application doit construire une chaîne de *Certificat*s ramenant à une CA fiable.

Lors de la construction d'une chaîne, chaque *Certificat* de la chaîne doit être validé. S'il se produit une quelconque erreur de validation, la vérification de la confiance a échoué. Certaines erreurs de validation ne sont pas critiques, ce qui signifie qu'elles peuvent être éliminées par l'utilisateur d'une *Application* ayant les privilèges appropriés. Les erreurs de validation supprimées sont toujours consignées dans un rapport via un audit (cela signifie qu'un événement d'Audit approprié est soulevé).

La construction d'une chaîne de confiance nécessite l'accès à tous les *Certificat*s de la chaîne. Ces *Certificats* peuvent être stockés localement ou ils peuvent être fournis avec le *Certificat* d'application. Le traitement échoue avec Bad_SecurityChecksFailed si un *Certificat* de CA ne peut pas être trouvé.

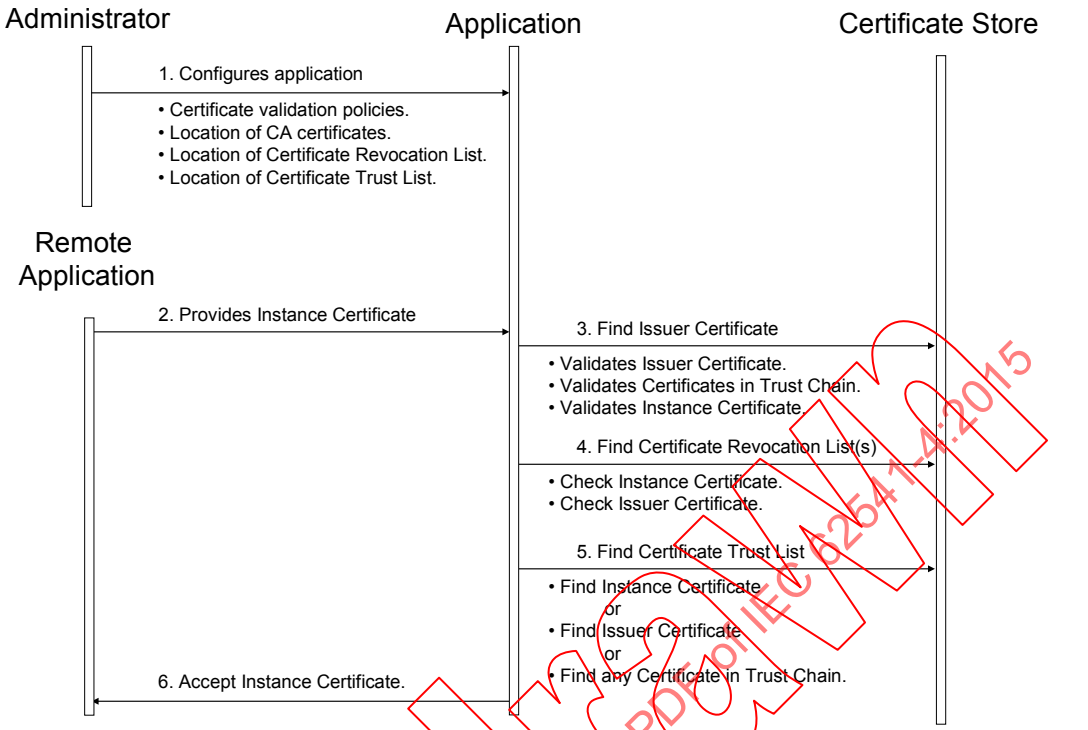
Le Tableau 101 spécifie les étapes utilisées pour valider un *Certificat* et l'ordre dans lequel on doit les suivre. Ces étapes sont répétées pour chaque *Certificat* de la chaîne. Chaque étape de validation a un statut d'erreur et un type d'événement d'audit qui sont uniques et doivent être consignés dans un rapport si la vérification échoue. L'événement d'audit s'ajoute à tout événement d'audit qui a été généré pour le *Service* particulier qui a été invoqué. L'événement d'audit du *Service* dans le texte de son message doit inclure l'*EventID* d'audit de l'*AuditCertificateEventType* (pour plus de détails, voir 6.2). Le traitement s'arrête si une erreur se produit, à moins qu'elle ne soit pas critique et qu'elle ait été éliminée.

Les *ApplicationInstanceCertificates* ne doivent pas être utilisés dans un *Client* ou un *Serveur* jusqu'à ce qu'ils soient évalués et marqués comme fiables. Cela peut se produire de manière automatique par une chaîne de confiance ICP ou de manière hors ligne où le *Certificat* est marqué comme étant fiable par un administrateur après évaluation.

Tableau 101 – Étapes de validation d'un certificat

Étape	Erreur/Événement d'audit	Description
Structure du certificat	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	La structure du *Certificat* est vérifiée. Cette erreur ne peut pas être supprimée.
Signature	Bad_SecurityChecksFailed AuditCertificateInvalidEventType	Un *Certificat* avec une signature non valide doit toujours être rejeté. Une signature de *Certificat* est non valide si le *Certificat* de l'*émetteur* est inconnu. Un *Certificat* autosigné est son propre émetteur.
Vérification de la liste de confiance	Bad_SecurityChecksFailed AuditCertificateUntrustedEventType	Si le *Certificat d'Instance d'Application* n'est pas fiable et qu'aucun des *Certificats* de CA dans la chaîne n'est fiable, le résultat de la validation du *Certificat* doit être Bad_SecurityChecksFailed.
Période de validité	Bad_CertificateTimeInvalid Bad_CertificateIssuerTimeInvalid AuditCertificateExpiredEventType	L'instant actuel doit se situer après le début de la période de validité et avant la fin. Cette erreur peut être supprimée.
Nom de l'hôte	Bad_CertificateHostNameInvalid AuditCertificateDataMismatchEventType	Le HostName dans l'URL utilisé pour se connecter au *Serveur* doit être le même que l'un des HostNames spécifiés dans le *Certificat*. Cette vérification est ignorée pour les *Certificats* de CA. Cette erreur peut être supprimée.
URI	Bad_CertificateUriInvalid AuditCertificateDataMismatchEventType	Les *Certificats d'Application* et *de Logiciel* contiennent un URI d'application ou de produit qui doit correspondre à l'URI spécifié dans l'*ApplicationDescription* fournie avec le *Certificat*. Cette vérification est ignorée pour les *Certificats de CA*. Cette erreur ne peut pas être supprimée. Le *gatewayServerUri* est utilisé pour valider un *Application Certificate* lors de la connexion à un *Gateway Server* (voir 7.1).
Utilisation du certificat	Bad_CertificateUseNotAllowed Bad_CertificateIssuerUseNotAllowed AuditCertificateMismatchEventType	Chaque *Certificat* a un ensemble d'utilisations pour le *Certificat* (voir IEC 62541-6). Ces utilisations doivent correspondre à l'utilisation demandée pour le *Certificat* (à savoir Application, Logiciel ou CA). Cette erreur peut être supprimée à moins que le *Certificat* n'indique que l'utilisation est obligatoire.
Trouver la Liste d'annulation	Bad_CertificateRevocationUnknown Bad_CertificateIssuerRevocationUnknown AuditCertificateRevokedEventType	Chaque *Certificat* de CA peut avoir une liste d'annulation. Cette vérification échoue si cette liste n'est pas disponible (à savoir, une interruption du réseau empêche l'application d'accéder à la liste). Aucune erreur n'est rapportée si l'*Administrateur* désactive les vérifications d'annulation pour un *Certificat* de CA. Cette erreur peut être supprimée.
Vérification d'annulation	Bad_CertificateRevoked Bad_CertificateIssuerRevoked AuditCertificateRevokedEventType	Le *Certificat* a été annulé et ne peut pas être utilisé. Cette erreur ne peut pas être supprimée.

Les *Certificats* sont en général placés dans un lieu central appelé *CertificateStore*. La Figure 20 illustre les interactions entre l'*Application*, l'*Administrateur* et le *CertificateStore*. Le *CertificateStore* peut être sur la machine locale ou dans quelque serveur central. Les mécanismes exacts utilisés pour accéder au *CertificateStore* dépendent de l'application et de l'environnement ICP installé par l'*Administrateur*.



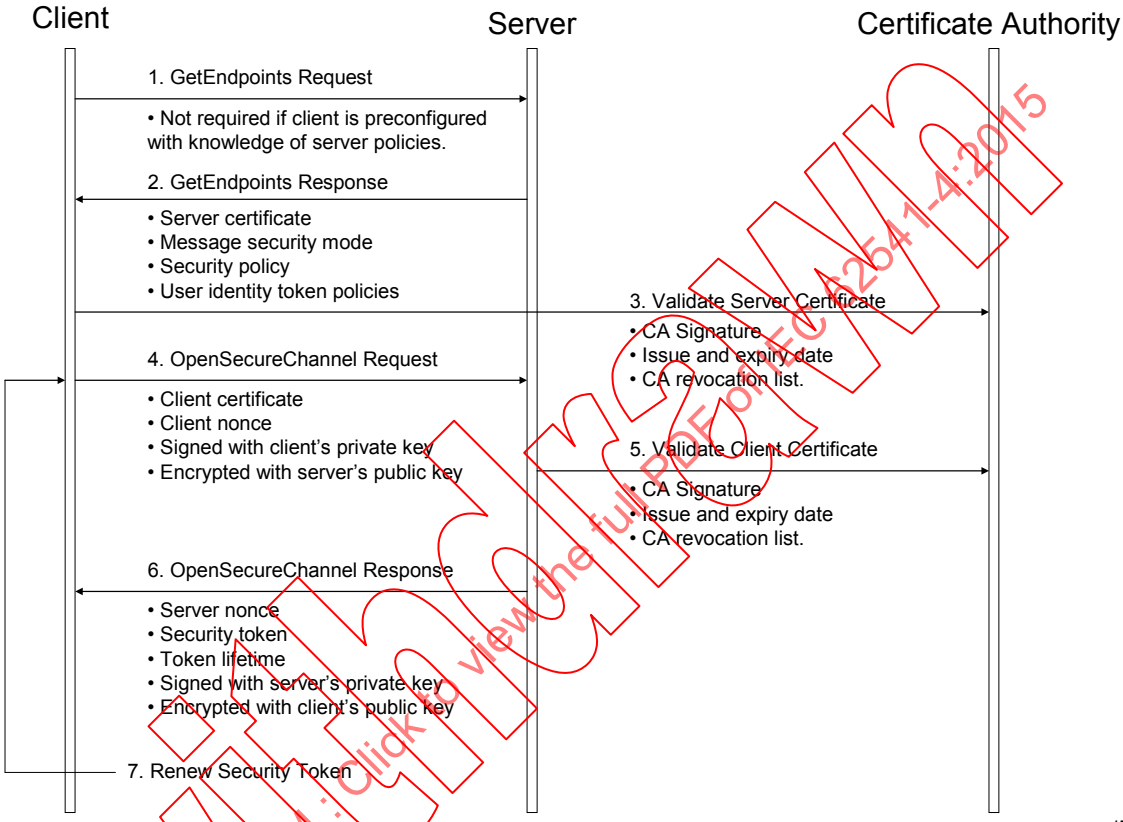
Anglais	Français
Administrator	Administrateur
Certificate store	Stockage de certificats
Configures application	Configure l'application
Certificate validation policies	Politiques de validation de certificat
Location of CA certificates	Emplacement des certificats de CA
Location of Certificate Revocation List	Emplacement de la Liste d'annulation de certificats
Location of Certificate Trust List	Emplacement de la Liste de certificats de confiance
Remote application	Application distante
Provides Instance Certificate	Fournit un certificat d'instance
Find Issuer Certificate	Trouver le certificat de l'émetteur
Validates Issuer Certificate	Valide le certificat d'émetteur
Validates certificates in Trust Chain	Valide les certificats dans la chaîne de confiance
Validates Instance Certificate	Valide le certificat d'instance
Find Certificate Revocation List(s)	Trouver la/les liste(s) d'annulation de certificats
Check Instance Certificate	Vérifier le certificat d'instance
Check Issuer Certificate	Vérifier le certificat d'émetteur
Find Certificate Trust List	Trouver la Liste de certificats de confiance
Find Instance Certificate	Trouver le certificat d'instance
Or	Ou
Find Issuer Certificate	Trouver le certificat de l'émetteur
Find any certificate in Trust Chain	Trouver n'importe quel certificat dans la chaîne de confiance
Accept instance certificate	Accepter le certificat d'instance

Figure 20 – Déterminer si un Certificat d'Instance d'Application est Fiable

6.1.4 Créer un Canal Sécurisé (SecureChannel)

Toutes les *Applications OPC UA* doivent établir un *Canal Sécurisé* avant de créer une *Session*. Ce *Canal Sécurisé* exige que les deux applications aient accès aux *Certificats* qui peuvent être utilisés pour chiffrer et signer l'échange des *Messages*. Les *Application Instance Certificates* installés en suivant le processus décrit en 6.1.2 peuvent être utilisés à cette fin.

Les étapes impliquées dans l'établissement d'un *Canal Sécurisé* sont montrées à la Figure 21.



IEC

Anglais	Français
Server	Serveur
Certificate authority	Autorité de certification
GetEndpoints Request	Demande de GetEndpoints
Not required if client is preconfigured with knowledge of server policies	Non requise si le client est préconfiguré avec connaissance des politiques serveur
GetEndpoints response	Réponse GetEndpoints
Server certificate	Certificat de Serveur
Message security mode	Mode sécurité de message
Security policy	Politique de sécurité
User identity token policies	Politiques de jeton d'identité d'utilisateur
Validate Server Certificate	Valider le certificat de Serveur
CA signature	Signature de CA
Issue and expiry date	Date d'émission et d'expiration
CA revocation list	Liste d'annulation de CA
OpenSecureChannel Request	Demande OpenSecureChannel
Client certificate	Certificat Client

Anglais	Français
Client nonce	Nonce de Client
Signed with client's private key	Signée avec la clé privée du client
Encrypted with server's public key	Chiffrée avec la clé publique du serveur
Validate Client Certificate	Valider le certificat de Client
Server nonce	Nonce de Serveur
Security token	Jeton de sécurité
Token lifetime	Durée de vie du jeton
Signed with server's private key	Signée avec la clé privée du serveur
Encrypted with client's public key	Chiffrée avec la clé publique du client
Renew Security Token	Renouveler le jeton de sécurité

Figure 21 – Etablir un Canal Sécurisé

La Figure 21 ci-dessus suppose que le *Client* et le *Serveur* ont un accès en ligne à une *Autorité de Certification* (CA). Si un accès en ligne n'est pas disponible et si l'administrateur a installé une clé publique de CA sur la machine locale, le *Client* et *Serveur* doivent encore valider les *Certificats* d'application en utilisant la clé en question. La figure montre une seule CA, mais il n'y a aucune exigence pour que les *Certificats* du *Client* et du *Serveur* soient émis par la même autorité. Un *Certificat d'Instance d'Application* autosigné n'a pas besoin d'être vérifié par une CA. Tout *Certificat* doit être rejeté s'il ne figure pas dans une liste de confiance fournie par l'administrateur.

Le *Client* et le *Serveur* doivent avoir une liste de *Certificats* à laquelle ils font confiance (parfois appelée Liste de *Certificats* de confiance ou CTL) en raison de leur configuration. Ces *Certificats* de confiance peuvent être des *Certificats* pour des *Autorités de Certification* ou bien ils peuvent être des *Certificats d'Instance d'Application OPC UA*. Les *Applications OPC UA* doivent être configurées pour refuser les connexions à des applications qui n'ont pas de *Certificat* fiable.

L'intégrité des *certificats* peut être compromise, ce qui signifie qu'il convient de ne plus leur accorder de confiance. Les administrateurs peuvent annuler un *Certificat* en le retirant de la liste de confiance pour toutes les applications ou bien la CA peut ajouter le *Certificat* à la Liste d'annulation de *Certificats* (CRL) pour le *Certificat de l'émetteur*. Les administrateurs peuvent sauvegarder une copie de la CRL pour chaque *Certificat d'émetteur* lorsque l'accès en ligne n'est pas disponible.

Un *Client* n'a pas besoin d'appeler *GetEndpoints* chaque fois qu'il se connecte au *Serveur*. Il convient que cette information change rarement et le *Client* peut la placer en cache localement. Si le *Serveur* rejette la demande *OpenSecureChannel*, il convient que le *Client* appelle *GetEndpoints* et s'assure que la configuration du *Serveur* n'a pas changé.

Il existe deux risques pour la sécurité dont un *Client* doit être conscient lorsqu'il utilise le *Service GetEndpoints*. Le premier peut provenir d'un *Server Discovery* malveillant qui essaye de diriger le *Client* vers un *Serveur* malveillant. Pour cette raison, le *Client* doit vérifier que le *ServerCertificate* dans l'*EndpointDescription* est un *Certificat* fiable avant d'appeler *CreateSession*.

Le second risque pour la sécurité provient d'une tierce partie qui altère le contenu des *EndpointDescriptions* à mesure qu'elles sont transférées sur le réseau pour revenir au *Client*. Le *Client* s'en protège en comparant la liste des *EndpointDescriptions* retournée par le *Service GetEndpoints* à la liste retournée dans la réponse *CreateSession*.

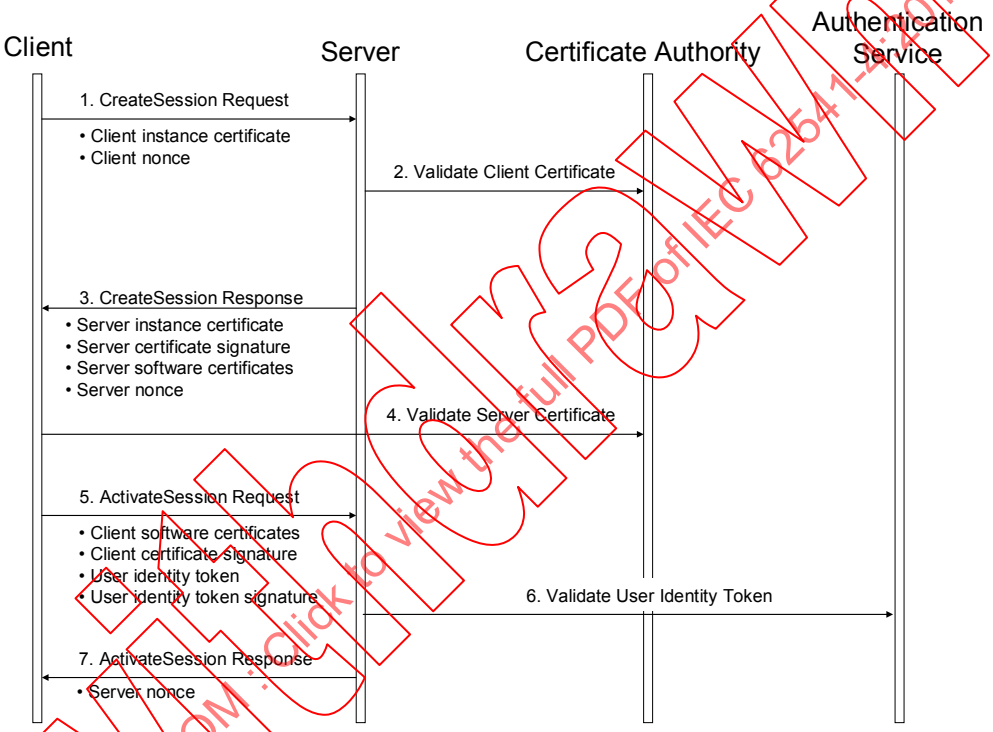
Les mécanismes exacts pour utiliser le jeton de sécurité pour signer et chiffrer les *Messages* échangés sur le *Canal Sécurisé* sont décrits dans l'IEC 62541-6. Le processus de renouvellement des jetons est également décrit en détail dans l'IEC 62541-6.

Dans de nombreux cas, les *Certificats* utilisés pour établir le *Canal Sécurisé* sont les *Certificats d'Instance d'Application*. Cependant, certaines *Piles de Communication* peuvent ne pas prendre en charge des *Certificats* qui sont spécifiques à une seule application. Elles attendent plutôt que toute la communication soit sécurisée par un *Certificat* spécifique à un utilisateur ou à toute la machine. Pour cette raison, les *Applications OPC UA* ont besoin d'échanger leurs *Certificats d'Instance d'Application* pour créer une *Session*.

6.1.5 Créer une Session

Une fois qu'un *Client* OPC UA a établi un *Canal Sécurisé* avec un *Serveur*, il peut créer une *Session* OPC UA.

Les étapes impliquées dans l'établissement d'une *Session* sont montrées à la Figure 22.



Anglais	Français
Server	Serveur
Certificate authority	Autorité de certification
Authentication service	Service d'authentification
CreateSession Request	Demande CreateSession
Client instance certificate	Certificat d'instance client
Client nonce	Nonce Client
Validate Client Certificate	Valider le certificat client
CreateSession Response	Réponse CreateSession
Server instance certificate	Certificat d'instance Serveur
Server certificate signature	Signature de certificat Serveur
Server software certificates	Certificats de logiciel Serveur
Server nonce	Nonce de Serveur
Validate Server Certificate	Valider le certificat de Serveur
ActivateSession Request	Demande ActivateSession
Client software certificates	Certificats de logiciel Client

Anglais	Français
Client certificate signature	Signature de certificat Client
User identity token	Jeton d'identité d'utilisateur
User identity token signature	Signature de jeton d'identité d'utilisateur
Validate user identity token	Valider le jeton d'identité d'utilisateur
ActivateSession Response	Réponse ActivateSession

Figure 22 – Etablir une Session

La Figure 22 ci-dessus illustre les interactions entre un *Client*, un *Serveur*, une *Autorité de Certification* (CA) et un service d'authentification. La CA est chargée d'émettre les *Certificats d'Instance d'Application*. Si le *Client* ou le *Serveur* n'a pas un accès en ligne à la CA, il doit valider les *Certificats d'Instance d'Application* en utilisant la clé publique de la CA que l'administrateur doit installer sur la machine locale.

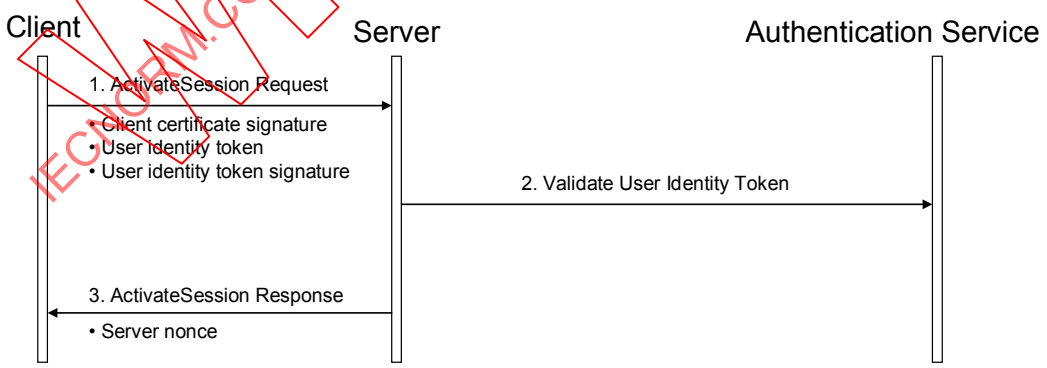
Le service d'authentification est une base de données centrale qui peut vérifier ce jeton d'utilisateur fourni par le *Client*. Ce service d'authentification peut également indiquer au *Serveur* les droits d'accès que l'utilisateur possède. Le service d'authentification dépend du jeton d'identité d'utilisateur. Il peut s'agir d'une *Autorité de Certification*, d'un service d'octroi de ticket Kerberos, d'un *Serveur* WS-Trust ou d'une base de données propriétaires quelconque.

Le *Client* et le *Serveur* doivent prouver la possession de leurs *Certificats d'Instance d'Application* en signant les *Certificats* avec un nonce accolé. Le mécanisme exact utilisé pour créer les signatures de preuve de possession est décrit en 5.6.2. De même, le *Client* doit prouver la possession de certains types de jetons d'identité d'utilisateur en créant des signatures avec le secret associé au jeton.

6.1.6 Se substituer à un utilisateur

Une fois qu'un *Client* OPC UA a établi une *Session* avec un *Serveur*, il peut changer l'identité d'utilisateur associée à la *Session* en appelant le service *ActivateSession*.

Les étapes impliquées dans la substitution à un utilisateur sont montrées à la Figure 23.



IEC

Anglais	Français
Server	Serveur
Authentication service	Service d'authentification
ActivateSession Request	Demande ActivateSession
Client certificate signature	Signature de certificat Client

Anglais	Français
User identity token	Jeton d'identité d'utilisateur
User identity token signature	Signature de jeton d'identité d'utilisateur
Validate User Identity Token	Valider le jeton d'identité d'utilisateur
ActivateSession Response	Réponse ActivateSession
Server nonce	Nonce de serveur

Figure 23 – Se substituer à un utilisateur

6.2 Certificats de logiciel (Software Certificates)

6.2.1 Vue d'ensemble

Toutes les *Applications OPC UA* peuvent avoir un ou plusieurs *Certificats* de logiciel qui sont émis par des autorités de certification et spécifient les profils que l'application prend en charge. Le présent Article décrit comment sont obtenus, installés et validés les *SoftwareCertificates*.

Les *SoftwareCertificates* sont utilisés pour identifier un produit logiciel qui est une *Application OPC UA* et pour décrire les fonctions du produit. Les fonctions sont décrites par des *Profils* définis dans l'IEC 62541-7. Les *SoftwareCertificates* ne sont pas pertinents pour la sécurité en OPC UA.

6.2.2 Obtenir et installer un Certificat de logiciel

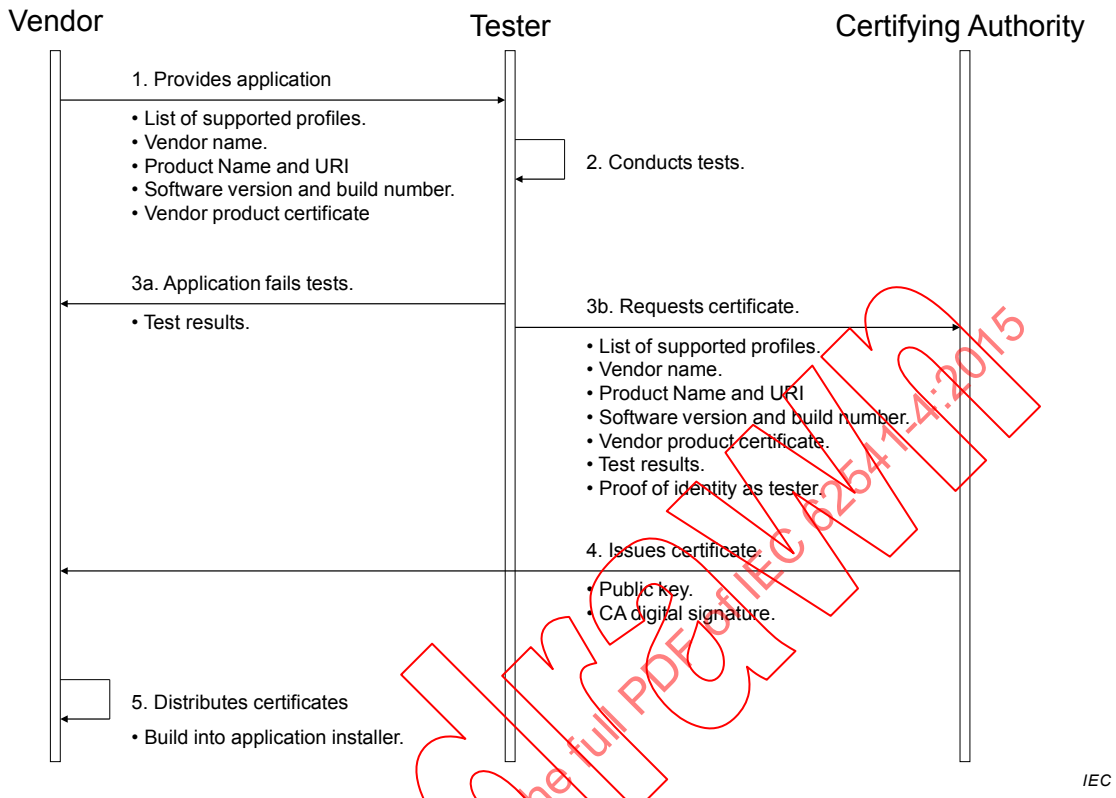
Les *Certificats* de logiciel contiennent les informations suivantes:

- Le nom du fournisseur responsable du produit;
- Le nom du produit;
- L'URI du produit;
- Le numéro de version et de mouture du logiciel;
- Le *Certificat* du produit du fournisseur;
- Une liste des profils pris en charge par l'application;
- Le statut d'essais de certification pour chaque profil pris en charge;
- Le nom de l'autorité de contrôle qui a émis le *Certificat*;
- La date d'émission et d'expiration du *Certificat*;
- La clé publique émise pour l'application par l'autorité de certification;
- Une signature numérique créée par l'autorité de certification.

Le fournisseur du produit a la responsabilité de mener à bien le processus de certification et de demander des *certificats* de logiciel auprès des autorités de certification. Le fournisseur doit installer les *Certificats* de logiciel lorsqu'une application est installée sur une machine. Lorsqu'ils distribuent ces *Certificats*, il convient que les fournisseurs d'application prennent des précautions pour empêcher que des utilisateurs non autorisés acquièrent leurs *Certificats* de logiciel et les utilisent pour des applications que le fournisseur n'a pas développées. Les *Certificats* de logiciel improprement utilisés ne sont pas un risque pour la sécurité, mais les fournisseurs peuvent être réprimandés pour des problèmes d'interopérabilité dus à l'utilisation de *Certificats* par des applications non autorisées.

Des fournisseurs peuvent choisir d'affecter leur propre *Certificat* à l'application (appelé *Certificat* Produit du Fournisseur). Ce *Certificat* serait alors incorporé dans le *SoftwareCertificate*. La clé privée pour le *Certificat* produit du fournisseur est gérée par le fournisseur.

Les étapes impliquées dans l'acquisition et l'installation d'un *Certificat* de logiciel auprès d'une *Autorité de Certification* sont montrées à la Figure 24.



Anglais	Français
Vendor	Fournisseur
Tester	Vérificateur
Certifying authority	Autorité de certification
Provides application	Fournit l'application
List of supported profiles	Liste des profils pris en charge
Vendor name	Nom de fournisseur
Product name and URI	Nom et URI de produit
Software version and build number	Numéro de version et de mouture du logiciel
Vendor product certificate	Certificat de produit du fournisseur
Conduct tests	Conduit les essais
Application fails tests	L'application échoue aux essais
Test results	Résultats d'essai
Requests certificate	Demande un certificat
Proof of identity as tester	Preuve de l'identité comme vérificateur
Issues certificate	Émet un certificat
Public key	Clé publique
CA digital signature	Signature numérique de CA
Distributes certificates	Distribue des certificats
Build into application installer	Bâtir dans un installateur d'application

Figure 24 – Obtenir et installer un Certificat de logiciel

La figure ci-dessus illustre les interactions entre le *Fournisseur (Vendor)*, le *Vérificateur (Tester)* et l'autorité de certification (*CertifyingAuthority)*. Le *Fournisseur* est l'organisation

responsable de l'*Application OPC UA*. Le *Vérificateur* peut être le fournisseur (pour l'autocertification) ou peut être une installation d'essai de tierce partie. La *CertifyingAuthority* est une Autorité de certification ICP (infrastructure à clé publique) gérée par l'organisation qui a créé les profils d'*Application OPC UA* et les programmes de certification.

L'autorité de certification émettra des *Certificats* uniquement pour les fournisseurs en qui elle a confiance. Pour cette raison, le *Vérificateur* doit fournir une preuve d'identité avant que l'*Autorité de Certification* n'émette un *Certificat*.

6.2.3 Valider un Certificat de logiciel

Les *Applications OPC UA* doivent valider les *Certificats de logiciel* fournis par les applications avec lesquelles elles communiquent.

Un *Certificat de logiciel* est valide si:

- La signature sur le *Certificat de logiciel* est valide;
- le *Certificat de logiciel* a dépassé sa date d'émission et il n'a pas expiré;
- le *Certificat de logiciel* n'a pas été annulé par l'émetteur;
- le *Certificat d'émetteur* est valide et n'a pas été annulé par la CA qui l'avait émis.

Les étapes utilisées pour valider des *Certificats de logiciel* sont pratiquement les mêmes que celles décrites pour les *Certificats d'Instance d'Application* en 6.1.3. La seule différence est que les *Certificats de logiciel* ne sont pas vérifiés par rapport à la *TrustList* de l'*Application.*

Les *Certificats de logiciel* contiennent toujours un *Certificat d'Instance d'Application* qui est la propriété du fournisseur qui distribue l'application. Le *Certificat* de l'application peut aussi être dans la chaîne des *Certificats* utilisée pour émettre des *Certificats d'Instance d'Application*. Pour cette raison, les *Applications OPC UA* peuvent rejeter des *SoftwareCertificates* fournis par des applications si le *Certificat* de l'application ne fait pas partie de la chaîne de *Certificats* pour le *Certificat d'Instance d'Application*. Il convient que les *Applications OPC UA* permettent aux *Administrateurs* d'exiger ce comportement. Les profils définis dans l'IEC 62541-7 peuvent spécifier plus précisément les comportements de gestion attendus des *Certificats*.

6.3 Audits

6.3.1 Vue d'ensemble

L'audit est une exigence dans de nombreux systèmes. Il fournit un moyen de suivre les activités qui ont lieu comme partie intégrante du fonctionnement normal du système. Il fournit aussi un moyen de suivre un comportement anormal. Il est également une exigence du point de vue de la sécurité. Pour plus d'informations sur les aspects sécuritaires des audits, voir l'IEC TR 62541-2. Ce paragraphe décrit ce qui est attendu d'un *Serveur* et d'un *Client* OPC UA quant aux audits et il détaille les exigences relatives à l'audit pour chaque jeu de services. L'audit peut être accompli en utilisant l'une des deux méthodes suivantes ou les deux:

a) L'*Application OPC UA* qui génère un événement d'audit peut journaliser l'entrée d'audit dans un fichier journal ou autre lieu de stockage;

b) Le *Serveur* OPC UA qui génère l'événement d'audit peut l'éditer en utilisant le mécanisme d'événements OPC UA. Cela permet à un *Client* OPC UA de s'abonner aux entrées d'audit et de les journaliser dans un fichier journal ou dans un autre lieu de stockage.

6.3.2 Journaux d'audits généraux

Chaque demande de *Service* OPC UA contient un paramètre chaîne qui est utilisé pour contenir un id d'enregistrement d'audit. Un *Client* ou tout *Serveur* opérant comme *Client*, tel qu'un *Serveur* combiné par exemple, peut créer une entrée locale de journal d'audits pour une demande qu'il présente. Ce paramètre permet à ce *Client* de transmettre l'identificateur pour

cette entrée avec la demande. Si ce *Serveur* maintient aussi un journal d'audits, il convient qu'il inclue cet identificateur dans son entrée du journal d'audits qu'il écrit. Lorsque ce journal est examiné et l'entrée trouvée, l'examinateur peut relier celle-ci directement à l'entrée de journal d'audits créée par le *Client*. Cette fonction permet la traçabilité à travers les journaux d'audits dans un système.

6.3.3 Événements d'audit généraux

Un *Serveur* qui maintient un journal d'audits doit fournir les entrées de journal d'audits via des *Messages* d'*Événement*. L'*AuditEventType* et ses sous-types sont définis dans l'IEC 62541-3. Un *Message* d'*Événement* d'audit inclut aussi l'identificateur d'enregistrement d'audit. Les détails de l'*AuditEventType* et de ses sous-types sont définis dans l'IEC 62541-5. Un *Serveur* qui est un *Serveur* combiné qui prend en charge les audits doit aussi s'abonner à des événements d'audit pour tous les *Serveurs* qu'il combine (en supposant qu'ils fournissent des audits). Il convient que le flux combiné soit disponible à partir du *Serveur* combiné.

6.3.4 Audits pour jeu de services Discovery

Ce *Jeu de Services* peut être divisé en deux groupes: les *Services* appelés par les *Clients* OPC UA et les *Services* qui sont invoqués par des *Serveurs* OPC UA. Les *Services* *FindServers* et *GetEndpoints* qui sont appelés par les *Clients* OPC UA peuvent générer des entrées d'audit pour des invocations de *Service* non abouties. Le *Service RegisterServer* qui est invoqué par les *Serveurs* OPC UA doit générer des entrées d'audit pour tous les nouveaux enregistrements et pour les invocations de *Service* non abouties. Ces entrées d'audit doivent inclure l'URI de *Serveur*, les noms de *Serveur*, les URI *Discovery* et le statut isOnline. Il convient que des entrées d'audit ne soient pas générées pour l'invocation *RegisterServer* qui n'induit pas de changements aux *Serveurs* enregistrés.

6.3.5 Audits pour jeu de services SecureChannel

Tous les *Services* dans ce *Service Set* pour des *Serveurs* qui prennent en charge les audits peuvent générer des entrées d'audit et doivent générer des *Événements* d'audit pour les invocations de service non abouties et pour l'invocation réussie des Services *OpenSecureChannel* et *CloseSecureChannel*. Il convient que les entrées d'audit générées par le *Client* soient installées avant l'appel effectif, permettant la fourniture de l'id correct d'enregistrement d'audit. Le Service *OpenSecureChannel* doit générer un *Événement* d'audit du type *AuditOpenSecureChannelEventType* ou d'un sous-type de celui-ci pour *requestType* ISSUE_0. Les *Événements* d'audit pour *requestType* RENEW_1 ne sont créés que si renew échoue. Le service *CloseSecureChannel* doit générer un *Événement* d'audit du type *AuditChannelEventType* ou d'un sous-type de celui-ci. Ces deux types d'*Événement* sont des sous-types de l'*AuditChannelEventType*. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et paramètres supplémentaires. Pour les cas d'échec, il convient d'inclure dans le *Message* pour les *Événements* de ce type une description des raisons de l'échec du service. Il convient que la description soit plus détaillée que celle qui a été retournée au client. D'un point de vue sécurité, un client a seulement besoin de savoir qu'il y a eu échec. En revanche, du point de vue de l'*Audit*, il est nécessaire de connaître les détails exacts de l'échec. Dans le cas d'erreurs de validation de *Certificat*, il convient d'inclure dans la description l'*EventId* d'audit de l'*AuditCertificateEventType* spécifique qui a été généré pour rapporter l'erreur du *Certificat*. L'*AuditCertificateEvent* doit aussi contenir dans le détail l'erreur de validation du *Certificat*. Il convient d'inclure dans les paramètres supplémentaires les détails de la demande. Il est bien entendu que ces événements peuvent être générés dans de nombreux cas par les *Piles de Communication* sous-jacentes, mais ils doivent être à la disposition du *Serveur* et le *Serveur* doit les consigner dans un rapport.

6.3.6 Audits pour jeu de services Session

Tous les *Services* de ce *Service Set* pour les *Serveurs* qui prennent en charge les audits peuvent générer des entrées d'audit et doivent générer des *Événements* d'audit pour les invocations de *Service* abouties et non abouties. Ces *Services* doivent générer un *Événement* d'audit du type *AuditSessionEventType* ou d'un sous-type de celui-ci. En particulier, ils doivent générer l'*EventType* de base ou le sous-type approprié, en fonction du service qui a

été invoqué. Le service *CreateSession* doit générer des événements de l'*AuditActivateSessionEventType* ou des sous-types de celui-ci. Le service *ActivateSession* doit générer des événements de l'*AuditActivateSessionEventType* ou des sous-types de celui-ci. Lorsque le *Service ActivateSession* est appelé pour changer l'identité d'utilisateur, le serveur doit générer des événements de l'*AuditActivateSessionEventType* ou sous-types de celui-ci. Le service CloseSession service doit générer l'*EventType* de base de l'*AuditSessionEventType* ou sous-types de celui-ci. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et paramètres supplémentaires. Pour les cas d'échec, il convient d'inclure dans le *Message* pour les *Événements* de ce type une description des raisons de l'échec du *Service*. Il convient d'inclure dans les paramètres supplémentaires les détails de la demande.

Ce *Service Set* doit également générer des événements d'audit supplémentaires lorsqu'il se produit des erreurs de validation de *Certificat*. Ces événements d'audit sont générés en plus des *AuditSessionEventTypes*. Voir l'IEC 62541-3 pour la définition de l'*AuditCertificateEventType* et de ses sous-types.

Pour les *Clients* prenant en charge les audits, l'accès aux services dans le *Session Service Set* doit générer des entrées d'audit pour les invocations abouties et non abouties du *Service*. Il convient que ces entrées d'audit soient installées avant l'invocation effective du *Service*, permettant à l'invocation de contenir l'id correct d'enregistrement d'audit.

6.3.7 Audits pour jeu de services NodeManagement

Tous les Services de ce *Service Set* pour les *Serveurs* qui prennent en charge les audits peuvent générer des entrées d'audit et doivent générer des *Événements* d'audit pour les invocations de *Service* abouties et non abouties. Ces *Services* doivent générer un *Événement* d'audit du type *AuditNodeManagementEventType* ou sous-types de celui-ci. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et paramètres supplémentaires. Pour les cas d'échec, il convient d'inclure dans le *Message* pour les *Événements* de ce type une description des raisons de l'échec du service. Il convient d'inclure dans les paramètres supplémentaires les détails de la demande.

Pour les *Clients* prenant en charge les audits, l'accès aux *Services* dans le *NodeManagement Service Set,* doit générer des entrées d'audit pour les invocations abouties et non abouties du *Service*. Il convient que toutes les entrées d'audit soient installées avant l'invocation effective du *Service*, permettant à l'invocation de contenir l'id correct d'enregistrement d'audit.

6.3.8 Audits pour jeu de services Attribute

Les Services *Write* ou *HistoryUpdate* de ce *Service Set* pour les *Serveurs* qui prennent en charge les audits peuvent générer des entrées d'audit et doivent générer des *Événements* d'audit pour les invocations de *Service* abouties et non abouties. Ces *Services* doivent générer un *Événement* d'audit du type *AuditUpdateEventType* ou sous-types de celui-ci. En particulier, le *Service Write* doit générer un événement d'audit du type *AuditWriteUpdateEventType* ou d'un sous-type de celui-ci. Le *Service HistoryUpdate* doit générer un *Événement* d'audit du type *AuditHistoryUpdateEvent* ou d'un sous-type de celui-ci. Trois sous-types de l'*AuditHistoryUpdateEvent* sont définis, à savoir *AuditHistoryEventUpdateEventType*, *AuditHistoryValueUpdateEventType* et *AuditHistoryDeleteEventType*. Le sous-type dépend du type d'opération effectuée, à savoir mise à jour d'événements historiques, mise à jour de valeurs de données historiques ou suppression d'historique. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et des paramètres supplémentaires. Pour les cas d'échec, il convient d'inclure dans le *Message* pour les *Événements* de ce type une description des raisons de l'échec du *Service*. Il convient d'inclure dans les paramètres supplémentaires les détails de la demande.

Les *Services Read* et *HistoryRead* peuvent générer des entrées d'audit et des *Événements* d'audit pour les invocations de *Service* qui n'ont pas abouti. Il convient que ces *Services* génèrent un *Événement* d'audit du type *AuditEventType* ou d'un sous-type de celui-ci. Voir

l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et paramètres supplémentaires. Il convient d'inclure dans le *Message* pour les *Événements* de ce type une description des raisons de l'échec du *Service*.

Pour les *Clients* prenant en charge les audits, l'accès aux services *Write* ou *HistoryUpdate* dans l'*Attribute Service Set* doit générer des entrées d'audit pour les invocations abouties et non abouties du *Service*. Les invocations des autres *Services* dans ce *Service Set* peuvent générer des entrées d'audit. Il convient que toutes les entrées d'audit soient installées avant l'invocation effective du *Service*, permettant à l'invocation de contenir l'id correct d'enregistrement d'audit.

6.3.9 Audits pour jeu de services de Méthodes

Tous les *Services* de ce *Service Set* pour les *Serveurs* qui prennent en charge les audits peuvent générer des entrées d'audit et doivent générer des *Événements* d'audit pour les invocations abouties et non abouties si l'invocation modifie l'espace d'adresses, écrit une valeur ou modifie l'état du système (acquittement d'alarme, séquencement par lots ou autres modifications de système). Ces appels de méthodes doivent générer un *Événement* d'audit du type *AuditUpdateMethodEventType* ou sous-types de celui-ci. Les méthodes qui ne modifient pas *l'Espace d'Adresses*, n'écrivent pas de valeurs ou ne modifient pas l'état du système peuvent générer des événements. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et des paramètres supplémentaires.

Pour les *Clients* prenant en charge les audits, l'accès au *jeu de services de méthodes* doit générer des entrées d'audit pour les invocations abouties et non abouties du *Service* si l'invocation modifie l'espace d'adresses, écrit une valeur ou modifie l'état du système (acquittement d'alarme, séquencement par lots ou autres modifications de système). Les invocations des autres *Méthodes* peuvent générer des entrées d'audit. Il convient que toutes les entrées d'audit soient installées avant l'invocation effective du *Service*, permettant à l'invocation de contenir l'id correct d'enregistrement d'audit.

6.3.10 Audits pour le View Service Set, le Query Service Set, le MonitoredItem Service Set et le Subscription Service Set

Tous les *Services* de ces quatre *Service Sets* fournissent seulement des informations au *Client*, avec l'exception du *Service TransferSubscriptions* dans le *Subscription Service Set*. En général, ces services ne génèrent pas d'entrées d'audit ou de *Messages* d'événement d'audit. Le *Service TransferSubscriptions* doit générer un *Événement* d'audit du type *AuditSessionEventType* ou sous-types de celui-ci pour les invocations de *Service* abouties et non abouties. Voir l'IEC 62541-5 pour l'affectation détaillée des paramètres *SourceNode*, *SourceName* et des paramètres supplémentaires. Pour les cas d'échec, il convient d'inclure dans le *Message* pour les Événements de ce type une description des raisons de l'échec du service.

Pour les *Clients* prenant en charge les audits, l'accès au *Service TransferSubscriptions* dans le *Subscription Service Set* doit générer des entrées d'audit pour les invocations abouties et non abouties du *Service*. Les invocations des autres *Services* dans ce *Service Set* n'exigent pas d'entrées d'audit. Il convient que toutes les entrées d'audit soient installées avant l'invocation effective du *Service*, permettant à l'invocation de contenir l'id correct d'enregistrement d'audit.

6.4 Redondance

6.4.1 Vue d'ensemble de la redondance

La redondance dans OPC UA garantit que les Clients tout comme le Serveur peuvent être redondants. OPC UA ne fournit pas la redondance, elle fournit les structures de données et les services permettant d'obtenir la redondance d'une manière normalisée.

6.4.2 Vue d'ensemble de la redondance du serveur

6.4.2.1 Généralités

La redondance du serveur se présente sous deux modes, transparent et non transparent. Par définition, dans la redondance transparente, le basculement des responsabilités de *Serveur* d'un *Serveur* à un autre est transparent au *Client*: le *Client* ne se soucie pas de savoir, ou même ne sait pas, qu'un basculement a eu lieu; le *Client* n'a pas besoin de faire quoi que ce soit pour maintenir le flux de données. En revanche, le basculement non transparent exige une action du *Client*.

L'*Object ServerRedundancy* défini dans l'IEC 62541-5 indique le mode pris en charge par le *Serveur*. L'ObjectType *ServerRedundancyType* et ses sous-types *TransparentRedundancyType* et *NonTransparentRedundancyType* définis dans l'IEC 62541-5 spécifient les informations pour le mode de redondance pris en charge.

Les deux domaines où la redondance crée des besoins spécifiques sont le maintien de la synchronisation des informations du *Serveur* et du *Client* parmi les *Serveurs* et la régulation du basculement de flux de données d'un *Serveur* à l'autre.

Quel que soit le mode de redondance utilisé, il est prévu que tous les *Serveurs* dans le jeu redondant aient des espaces d'adresses identiques, y compris les *NodeIds* identiques et une logique identique pour le réglage du niveau de service.

6.4.2.2 Redondance transparente

Pour la redondance transparente, tout ce qu'OPC UA fournit se résume aux structures de données pour permettre au *Client* d'identifier les *Serveurs* dans le jeu redondant, le niveau de service de chaque *Serveur* et le *Serveur* actuellement responsable de la *Session Client*. Ces informations sont spécifiées dans l'*ObjectType* de *TransparentRedundancyType* défini dans l'IEC 62541-5.

Toutes les interactions OPC UA au sein d'une session donnée doivent être prises en charge par un *Serveur* donné et le *Client* est à même d'identifier le *Serveur* en question, ce qui permet une piste d'audit complète pour les données. Il est de la responsabilité des *Serveurs* d'assurer que les informations sont synchronisées entre les *Serveurs*. Un *Serveur* fonctionnel reprendra la *Session* et les *Subscriptions* du *Serveur* défaillant. Le basculement peut exiger une reconnexion à la couche transport du *Client,* mais l'URL d'*Endpoint* du *Serveur* ne doit pas changer. Voir 6.5 pour plus de détails sur le fait de rétablir une connexion.

La Figure 25 montre une configuration type de la redondance transparente.