

DIN EN IEC 62541-4

ICS 35.200; 35.240.50

EntwurfEinsprüche bis 2019-10-09
Vorgesehen als Ersatz für
DIN EN 62541-4:2015-10**OPC Unified Architecture –
Teil 4: Dienste
(IEC 65E/602/CDV:2018);
Englische Fassung prEN IEC 62541-4:2018**OPC Unified Architecture –
Part 4: Services
(IEC 65E/602/CDV:2018);
English version prEN IEC 62541-4:2018Architecture Unifiée OPC –
Partie 4: Services
(IEC 65E/602/CDV:2018);
Version anglaise prEN IEC 62541-4:2018**Anwendungswarnvermerk**

Dieser Norm-Entwurf mit Erscheinungsdatum 2019-08-09 wird der Öffentlichkeit zur Prüfung und Stellungnahme vorgelegt.

Weil die beabsichtigte Norm von der vorliegenden Fassung abweichen kann, ist die Anwendung dieses Entwurfs besonders zu vereinbaren.

Stellungnahmen werden erbeten

- vorzugsweise online im Norm-Entwurfs-Portal von DIN unter www.din.de/go/entwuerfe bzw. für Norm-Entwürfe der DKE auch im Norm-Entwurfs-Portal der DKE unter www.entwuerfe.normenbibliothek.de, sofern dort wiedergegeben;
- oder als Datei per E-Mail an dke@vde.com möglichst in Form einer Tabelle. Die Vorlage dieser Tabelle kann im Internet unter www.din.de/go/stellungnahmen-norm-entwuerfe oder für Stellungnahmen zu Norm-Entwürfen der DKE unter www.dke.de/stellungnahme abgerufen werden;
- oder in Papierform an die DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik in DIN und VDE, Stresemannallee 15, 60596 Frankfurt am Main.

Die Empfänger dieses Norm-Entwurfs werden gebeten, mit ihren Kommentaren jegliche relevanten Patentrechte, die sie kennen, mitzuteilen und unterstützende Dokumentationen zur Verfügung zu stellen.

Gesamtumfang 199 Seiten

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik in DIN und VDE

Nationales Vorwort

Die Englische Fassung des europäischen Dokuments prEN IEC 62541-4:2018 „OPC Unified Architecture – Teil 4: Dienste“ (Entwurf in der Umfrage) ist unverändert in diesen Norm-Entwurf übernommen worden.

Die Internationale Elektrotechnische Kommission (IEC) und das Europäische Komitee für Elektrotechnische Normung (CENELEC) haben vereinbart, dass ein auf IEC-Ebene erarbeiteter Entwurf für eine Internationale Norm zeitgleich (parallel) bei IEC und CENELEC zur Umfrage (CDV-Stadium) und Abstimmung als FDIS (en: Final Draft International Standard) bzw. Schluss-Entwurf für eine Europäische Norm gestellt wird, um eine Beschleunigung und Straffung der Normungsarbeit zu erreichen. Dem entsprechend ist das internationale Dokument IEC 65E/602/CDV:2018 „OPC Unified Architecture – Part 4: Services“ unverändert in den Entwurf prEN IEC 62541-4:2018 übernommen worden.

Das internationale Dokument wurde vom SC 65E „Devices and integration in enterprise systems“ der Internationalen Elektrotechnischen Kommission (IEC) erarbeitet.

Bei der Abstimmung zu dem Europäischen Schluss-Entwurf bei CENELEC und dem Internationalen Schluss-Entwurf bei IEC [Final Draft International Standard (FDIS)] sind jeweils nur „JA/NEIN“-Entscheidungen möglich, wobei „NEIN“-Entscheidungen fundiert begründet werden müssen. Dokumente, die bei CENELEC als Europäische Norm angenommen und ratifiziert werden, sind unverändert als Deutsche Normen zu übernehmen.

Für dieses Dokument ist das nationale Arbeitsgremium K 931 „Systemaspekte der Automatisierung“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik in DIN und VDE (www.dke.de) zuständig.

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf ein Dokument ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils aktuellste Ausgabe des in Bezug genommenen Dokuments.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe des Dokuments.

Der Zusammenhang der zitierten Dokumente mit den entsprechenden deutschen Dokumenten ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

Das Präsidium des DIN hat mit Präsidialbeschluss 1/2004 festgelegt, dass DIN-Normen, deren Inhalt sich auf internationale Arbeitsergebnisse der Informationsverarbeitung gründet, unter bestimmten Bedingungen allein in englischer Sprache veröffentlicht werden dürfen. Diese Bedingungen sind für den vorliegenden Norm-Entwurf erfüllt.

Da sich die Benutzer des vorliegenden Norm-Entwurfs der englischen Sprache als Fachsprache bedienen, wird die englische Fassung der prEN IEC 62541-4 veröffentlicht. Zu deren Abschnitt 1, der den Anwendungsbereich festlegt, und Unterabschnitt 3.1, der die Begriffe festlegt, wurde eine Übersetzung angefertigt und als informativer Nationaler Anhang NA des vorliegenden Norm-Entwurfs hinzugefügt. Für einige der verwendeten Begriffe existieren keine gebräuchlichen deutschen Benennungen, da sich die deutschen Anwender in der Regel ebenfalls der englischen Benennungen bedienen. Dieser Norm-Entwurf steht nicht in unmittelbarem Zusammenhang mit Rechtsvorschriften und ist nicht als Sicherheitsnorm anzusehen.

Es ist vorgesehen, auch bei der entsprechenden zukünftigen Deutschen Norm auf die deutsche Sprachfassung zu verzichten und diese in der englischsprachigen Fassung zu veröffentlichen.

Es wird auf die Möglichkeit hingewiesen, dass einige Elemente dieses Dokuments Patentrechte berühren können. IEC ist nicht dafür verantwortlich, einige oder alle diesbezüglichen Patentrechte zu identifizieren.

Das Original-Dokument enthält Bilder in Farbe, die in der Papierversion in einer Graustufen-Darstellung wiedergegeben werden. Elektronische Versionen dieses Dokuments enthalten die Bilder in der originalen Farbdarstellung.

Änderungen

Gegenüber DIN EN 62541-4:2015-10 wurden folgende Änderungen vorgenommen:

- a) Fähigkeit zur erneuten Sendung aller Daten von überwachten Einheiten in einer Subskription mittels des Verfahrens ResendData hinzugefügt;
- b) Unterstützung für dauerhafte Subskriptionen (Lebensdauer von Stunden oder Tagen) hinzugefügt;
- c) Register2 und FindServersOnNetwork-Dienste zur Unterstützung netzwerkweiter Erkennung mittels Fähigkeitsfiltern hinzugefügt;
- d) Definition von Softwarezertifikaten entfernt; muss in einer zukünftigen Version definiert werden;
- e) Definition der Redundanz erweitert und teilweise überarbeitet. Unterbereichsdefinitionen für ServiceLevel und mehr Begriffe für Redundanz hinzugefügt;
- f) Abschnitt zur Nutzung von Autorisierungsdiensten zur Anforderung von Benutzerzugangs-Tokens hinzugefügt;
- g) JSON-Web-Tokens (JWTs) als neue Benutzer-Token hinzugefügt;
- h) Konzept des sitzungslosen Dienstaufrufs hinzugefügt;
- i) generische Struktur, die die Weitergabe jeglicher Anzahl von Attributen zum Dienst AddNodes ermöglicht, hinzugefügt;
- j) Anforderung zum Schutz gegen Angriffe durch Benutzeridentitäts-Token hinzugefügt;
- k) neues Format EncryptedSecret für Benutzeridentitäts-Token hinzugefügt.

Nationaler Anhang NA (informativ)

Auszugsweise Übersetzung

Reihenfolge und Inhalt der folgenden Abschnitte sind identisch mit denen in den jeweiligen Abschnitten der Englischen Fassung.

1 Anwendungsbereich

Dieses Dokument legt die *Dienste* der OPC Unified Architecture (vereinheitlichte Architektur der plattformunabhängigen Kommunikation) fest. Die festgelegten *Dienste* sind eine Sammlung von abstrakten Fernverarbeitungsaufrufen (en: Remote Procedure Calls, RPC), welche durch OPC-UA-*Servers* implementiert werden und durch OPC-UA-*Clients* aufgerufen werden. Alle Interaktionen zwischen OPC-UA-*Clients* und OPC-UA-*Servers* erfolgen über diese *Dienste*. Die festgelegten *Dienste* werden als abstrakt betrachtet, weil in diesem Teil kein bestimmter RCP-Mechanismus für die Umsetzung festgelegt ist. In IEC 62541-6 sind eine oder mehr konkrete Abbildungen festgelegt, welche für eine Implementierung unterstützt werden. Zum Beispiel besteht in IEC 62541-6 eine Abbildung auf XML-Web-Dienste. In diesem Fall erscheinen die in diesem Teil festgelegten *Dienste* als die Webdienst-Methoden im WSDL-Vertrag.

Nicht von allen OPC-UA-*Servers* wird die Implementierung aller festgelegten *Dienste* gefordert. In IEC 62541-7 sind die *Profile* festgelegt, welche vorschreiben, welche Dienste für die Übereinstimmung mit einem bestimmten Profil implementiert werden müssen.

3 Begriffe und Konventionen

3.1 Begriffe

Für die Anwendung dieses Dokuments gelten die Begriffe nach IEC TR 62541-1, IEC TR 62541-2, IEC TR 62541-3 und die folgenden Begriffe.

ISO und IEC stellen terminologische Datenbanken für die Anwendung in der Normung unter den folgenden Adressen bereit:

- IEC Electropedia: verfügbar unter <http://www.electropedia.org/>
- ISO Online Browsing Platform: verfügbar unter <http://www.iso.org/obp>

3.1.1

aktiver Server

(en: Active Server)

Server, welcher aktuell Informationen beschafft

Anmerkung 1 zum Begriff: In redundanten OPC-UA-Systemen ist ein aktiver *Server* der *Server*, den ein *Client* als Datenquelle nutzt.

3.1.2

Totband

(en: Deadband)

zulässiger Bereich von Werteänderungen, der keine *Meldung* (en: Notification) einer Datenänderung auslöst

Anmerkung 1 zum Begriff: *Totband* kann als ein Filter beim Subskribieren von *Variablen* (en: Variables) angewendet werden, und wird verwendet, um bei verrauschten Signalen eine unnötige Aktualisierung des *Client* zu verhindern. Die vorliegende Norm legt *AbsolutesTotband* (en: AbsoluteDeadband) als einen gemeinsamen Filter fest. IEC 62541-8 legt einen zusätzlichen *Totband*-Filter fest.

3.1.3

ErkennungsEndpunkt

(en: DiscoveryEndpoint)

Endpunkt, welcher *Clients* den Zugang zu *Erkennungsdiensten* ohne Sicherheit ermöglicht

Anmerkung 1 zum Begriff: Ein *ErkennungsEndpunkt* ermöglicht den Zugang zu *Erkennungsdiensten* ohne *Sitzung* und ohne Nachrichtensicherheit.

3.1.4

Endpunkt

(en: Endpoint)

in einem Netzwerk verfügbare physikalische Adresse, die *Clients* erlaubt, auf einen oder mehrere *Dienste* zuzugreifen, die durch einen *Server* zur Verfügung gestellt werden

Anmerkung 1 zum Begriff: Jeder *Server* kann mehrere *Endpunkte* haben. Die Adresse eines *Endpunkts* muss einen *Hostnamen* einschließen.

3.1.5

ausgefallener Server

(en: Failed Server)

Server, der nicht in Betrieb ist

Anmerkung 1 zum Begriff: In einem redundanten OPC-UA-System ist ein ausgefallener *Server* ein *Server*, der nicht verfügbar oder nicht in der Lage ist, Daten zu liefern.

3.1.6

Fehlerumschaltung

(en: Failover)

Vorgang des Umschaltens der Quelle oder des Ziels von Informationen

Anmerkung 1 zum Begriff: In einem redundanten OPC-UA-System ist die Fehlerumschaltung der Vorgang, in dessen Rahmen ein *Client* von einem ausgefallenen oder beschädigten *Server* zu einem anderen *Server* im redundanten Satz umschaltet (Server-Fehlerumschaltung). In einigen Fällen hat der *Client* unter Umständen keine Kenntnis darüber, dass eine Fehlerumschaltung vorgenommen wird (transparente Redundanz). Der Vorgang, dass ein anderer *Client* eine bestehende ausgefallene oder beschädigte *Client*-Verbindung zu einem *a Server* ersetzt (*Client*-Fehlerumschaltung).

3.1.7

Gateway-Server

(en: Gateway Server)

Server, der als eine Zwischenstation für einen oder mehrere *Server* agiert

Anmerkung 1 zum Begriff: *Gateway-Server* können eingesetzt werden, um externen Zugriff einzuschränken, um Protokoll-Umwandlung zur Verfügung zu stellen oder um Eigenschaften zu Verfügung zu stellen, welche die zugrundeliegenden *Server* nicht bieten.

3.1.8

Hostname

(en: HostName)

eindeutiger Bezeichner für eine Maschine in einem Netzwerk

Anmerkung 1 zum Begriff: Dieser Bezeichner muss innerhalb eines lokalen Netzwerks eindeutig sein, er darf aber auch global eindeutig sein. Der Bezeichner kann eine IP-Adresse sein.

3.1.9

Sicherheits-Token

(en: Security Token)

Bezeichner für einen kryptografischen Schlüsselsatz

Anmerkung 1 zum Begriff: Alle *Sicherheits-Token* gehören zu einem Sicherheitskontext, der im Fall von OPC-UA der *Secure Channel* (Sicherer Kanal) ist.

3.1.10**SoftwareZertifikat**

(en: SoftwareCertificate)

digitales Zertifikat für ein Softwareprodukt, das auf verschiedenen Hosts installiert werden kann, um die Fähigkeiten des Softwareprodukts zu beschreiben

Anmerkung 1 zum Begriff: Unterschiedliche Installationen eines Softwareprodukts können dasselbe SoftwareZertifikat haben. SoftwareZertifikate sind nicht maßgeblich für die Sicherheit. Sie werden verwendet, um ein Softwareprodukt und seine unterstützten Fähigkeiten zu bezeichnen. SoftwareZertifikate sind in 6.4 festgelegt.

3.1.11**Redundanz**

das Vorhandensein doppelter Komponenten, welche den weiteren Betrieb nach einem Ausfall einer OPC-UA-Komponente ermöglichen

Anmerkung 1 zum Begriff: Dies kann für *Server*, *Clients* oder Netzwerke gelten.

3.1.12**redundanter Serversatz**

(en: Redundant Server Set)

zwei oder mehr *Server*, die für einander redundant sind

Anmerkung 1 zum Begriff: Ein redundanter Serversatz ist eine Reihe von *Servern*, die für *Redundanz* konfiguriert sind. Diese *Server* weisen Anforderungen auf, die mit dem Adressraum zusammenhängen, und liefern Fehlerumschaltungen.

CONTENTS

FIGURES	6
TABLES	7
FOREWORD	2
1 Scope	1
2 Normative references	1
3 Terms, definitions and conventions	1
3.1 Terms and definitions	1
3.2 Abbreviations and symbols	3
3.3 Conventions for Service definitions	3
4 Overview	4
4.1 Service Set model	4
4.2 Request/response Service procedures	7
5 Service Sets	7
5.1 General	7
5.2 Service request and response header	7
5.3 Service results	8
5.4 Discovery Service Set	9
5.4.1 Overview	9
5.4.2 FindServers	10
5.4.3 FindServersOnNetwork	11
5.4.4 GetEndpoints	12
5.4.5 RegisterServer	14
5.4.6 RegisterServer2	16
5.5 SecureChannel Service Set	17
5.5.1 Overview	17
5.5.2 OpenSecureChannel	19
5.5.3 CloseSecureChannel	21
5.6 Session Service Set	22
5.6.1 Overview	22
5.6.2 CreateSession	22
5.6.3 ActivateSession	26
5.6.4 CloseSession	29
5.6.5 Cancel	29
5.7 NodeManagement Service Set	30
5.7.1 Overview	30
5.7.2 AddNodes	30
5.7.3 AddReferences	32
5.7.4 DeleteNodes	34
5.7.5 DeleteReferences	35
5.8 View Service Set	36
5.8.1 Overview	36
5.8.2 Browse	36
5.8.3 BrowseNext	38
5.8.4 TranslateBrowsePathsToNodeIds	39

5.8.5	RegisterNodes	41
5.8.6	UnregisterNodes	42
5.9	Query Service Set	43
5.9.1	Overview	43
5.9.2	Querying Views	43
5.9.3	QueryFirst	43
5.9.4	QueryNext	47
5.10	Attribute Service Set	48
5.10.1	Overview	48
5.10.2	Read	48
5.10.3	HistoryRead	50
5.10.4	Write	52
5.10.5	HistoryUpdate	54
5.11	Method Service Set	55
5.11.1	Overview	55
5.11.2	Call	56
5.12	MonitoredItem Service Set	58
5.12.1	MonitoredItem model	58
5.12.2	CreateMonitoredItems	63
5.12.3	ModifyMonitoredItems	65
5.12.4	SetMonitoringMode	67
5.12.5	SetTriggering	68
5.12.6	DeleteMonitoredItems	69
5.13	Subscription Service Set	70
5.13.1	Subscription model	70
5.13.2	CreateSubscription	76
5.13.3	ModifySubscription	78
5.13.4	SetPublishingMode	79
5.13.5	Publish	81
5.13.6	Republish	82
5.13.7	TransferSubscriptions	83
5.13.8	DeleteSubscriptions	84
6	Service behaviours	85
6.1	Security	85
6.1.1	Overview	85
6.1.2	Obtaining and Installing an Application Instance Certificate	85
6.1.3	Determining if a Certificate is Trusted	87
6.1.4	Creating a SecureChannel	89
6.1.5	Creating a Session	91
6.1.6	Impersonating a User	92
6.2	Authorization Services	92
6.2.1	Overview	92
6.2.2	Indirect Handshake with an Identity Provider	92
6.2.3	Direct Handshake with an Identity Provider	93
6.3	Session-less Service Invocation	94
6.3.1	Description	94
6.3.2	Parameters	94
6.3.3	Service results	95
6.4	Software Certificates	95

6.5	Auditing.....	95
6.5.1	Overview.....	95
6.5.2	General audit logs.....	96
6.5.3	General audit Events.....	96
6.5.4	Auditing for Discovery Service Set	96
6.5.5	Auditing for SecureChannel Service Set.....	96
6.5.6	Auditing for Session Service Set	97
6.5.7	Auditing for NodeManagement Service Set	97
6.5.8	Auditing for Attribute Service Set	97
6.5.9	Auditing for Method Service Set	98
6.5.10	Auditing for View, Query, MonitoredItem and Subscription Service Set	98
6.6	Redundancy	98
6.6.1	Redundancy overview	98
6.6.2	Server Redundancy.....	99
6.6.3	Client Redundancy.....	109
6.6.4	Network Redundancy	109
6.6.5	Manually Forcing Failover	110
6.7	Re-establishing connections	111
6.8	Durable Subscriptions	112
7	Common parameter type definitions	113
7.1	ApplicationDescription.....	113
7.2	ApplicationInstanceCertificate	114
7.3	BrowseResult	114
7.4	ContentFilter	115
7.4.1	ContentFilter structure	115
7.4.2	ContentFilterResult	115
7.4.3	FilterOperator	116
7.4.4	FilterOperand parameters	123
7.5	Counter	124
7.6	ContinuationPoint.....	125
7.7	DataValue	125
7.7.1	General.....	125
7.7.2	PicoSeconds.....	125
7.7.3	SourceTimestamp	126
7.7.4	ServerTimestamp.....	126
7.7.5	StatusCode assigned to a value	126
7.8	DiagnosticInfo	127
7.9	DiscoveryConfiguration parameters.....	128
7.9.1	Overview.....	128
7.9.2	MdnsDiscoveryConfiguration.....	128
7.10	EndpointDescription	128
7.11	ExpandedNodeId.....	128
7.12	ExtensibleParameter	129
7.13	Index.....	129
7.14	IntegerId	129
7.15	MessageSecurityMode	129
7.16	MonitoringParameters	130
7.17	MonitoringFilter parameters.....	130
7.17.1	Overview.....	130

7.17.2	DataChangeFilter	131
7.17.3	EventFilter	132
7.17.4	AggregateFilter	133
7.18	MonitoringMode	135
7.19	NodeAttributes parameters	135
7.19.1	Overview	135
7.19.2	ObjectAttributes parameter	136
7.19.3	VariableAttributes parameter	136
7.19.4	MethodAttributes parameter	137
7.19.5	ObjectTypeAttributes parameter	137
7.19.6	VariableTypeAttributes parameter	137
7.19.7	ReferenceTypeAttributes parameter	138
7.19.8	DataTypeAttributes parameter	138
7.19.9	ViewAttributes parameter	138
7.19.10	GenericAttributes parameter	139
7.20	NotificationData parameters	139
7.20.1	Overview	139
7.20.2	DataChangeNotification parameter	139
7.20.3	EventNotificationList parameter	140
7.20.4	StatusChangeNotification parameter	140
7.21	NotificationMessage	140
7.22	NumericRange	141
7.23	QueryDataSet	142
7.24	ReadValueId	142
7.25	ReferenceDescription	143
7.26	RelativePath	143
7.27	RegisteredServer	144
7.28	RequestHeader	144
7.29	ResponseHeader	145
7.30	ServiceFault	146
7.31	SessionAuthenticationToken	146
7.32	SignatureData	147
7.33	SignedSoftwareCertificate	147
7.34	StatusCode	148
7.34.1	General	148
7.34.2	Common StatusCodes	150
7.35	TimestampsToReturn	153
7.36	UserIdentityToken parameters	153
7.36.1	Overview	153
7.36.2	Token Encryption and Proof of Possession	153
7.36.3	AnonymousIdentityToken	157
7.36.4	UserNameIdentityToken	157
7.36.5	X509IdentityTokens	158
7.36.6	IssuedIdentityToken	158
7.37	UserTokenPolicy	159
7.38	VersionTime	159
7.39	ViewDescription	160
Annex A (informative)	BNF definitions	161
A.1	Overview over BNF	161

A.2	BNF of RelativePath	161
A.3	BNF of NumericRange	162
Annex B (informative)	Content Filter and Query Examples	163
B.1	Simple ContentFilter examples	163
B.1.1	Overview	163
B.1.2	Example 1	163
B.1.3	Example 2	164
B.2	Complex Examples of Query Filters	164
B.2.1	Overview	164
B.2.2	Used type model	165
B.2.3	Example Notes	167
B.2.4	Example 1	168
B.2.5	Example 2	169
B.2.6	Example 3	170
B.2.7	Example 4	172
B.2.8	Example 5	173
B.2.9	Example 6	175
B.2.10	Example 7	176
B.2.11	Example 8	178
B.2.12	Example 9	179

FIGURES

Figure 1 – Discovery Service Set	4
Figure 2 – SecureChannel Service Set	5
Figure 3 – Session Service Set	5
Figure 4 – NodeManagement Service Set	5
Figure 5 – View Service Set	5
Figure 6 – Attribute Service Set	6
Figure 7 – Method Service Set	6
Figure 8 – MonitoredItem and Subscription Service Sets	7
Figure 9 – Discovery process	9
Figure 10 – Using a Gateway Server	14
Figure 11 – The Registration Process – Manually Launched Servers	15
Figure 12 – The Registration Process – Automatically Launched Servers	15
Figure 13 – SecureChannel and Session Services	18
Figure 14 – Multiplexing Users on a Session	24
Figure 15 – MonitoredItem Model	59
Figure 16 – Typical delay in change detection	60
Figure 17 – Queue overflow handling	61
Figure 18 – Triggering Model	62
Figure 19 – Obtaining and Installing an Application Instance Certificate	86
Figure 20 – Determining if a Application Instance Certificate is Trusted	89
Figure 21 – Establishing a SecureChannel	90
Figure 22 – Establishing a Session	91

Figure 23 – Impersonating a User	92
Figure 24 – Indirect Handshake with an Identity Provider	93
Figure 25 – Direct Handshake with an Identity Provider	93
Figure 29 – Transparent Redundancy setup example	100
Figure 30 – Non-Transparent Redundancy setup	101
Figure 31 – Client Start-up Steps	104
Figure 32 – Cold Failover	105
Figure 33 – Warm Failover	106
Figure 34 – Hot Failover	107
Figure 35 – HotAndMirrored Failover	108
Figure 36 – Server proxy for Redundancy	108
Figure 37 – Transparent Network Redundancy	109
Figure 38 – Non-Transparent Network Redundancy	110
Figure 39 – Reconnect Sequence	111
Figure 40 – Logical layers of a <i>Server</i>	146
Figure 41 – Obtaining a SessionAuthenticationToken	147
Figure 42 – EncryptedSecret Layout	155
Figure B.1 – Filter Logic Tree Example	163
Figure B.2 – Filter Logic Tree Example	164
Figure B.3 – Example Type Nodes	166
Figure B.4 – Example Instance Nodes	167
Figure B.5 – Example 1 Filter	168
Figure B.6 – Example 2 Filter Logic Tree	170
Figure B.7 – Example 3 Filter Logic Tree	171
Figure B.8 – Example 4 Filter Logic Tree	173
Figure B.9 – Example 5 Filter Logic Tree	174
Figure B.10 – Example 6 Filter Logic Tree	175
Figure B.11 – Example 7 Filter Logic Tree	177
Figure B.12 – Example 8 Filter Logic Tree	178
Figure B.13 – Example 9 Filter Logic Tree	180

TABLES

Table 1 – Service Definition Table	3
Table 2 – Parameter Types defined in IEC 62541-3	4
Table 3 – FindServers Service Parameters	11
Table 4 – FindServersOnNetwork Service Parameters	12
Table 5 – GetEndpoints Service Parameters	14
Table 6 – RegisterServer Service Parameters	16
Table 7 – RegisterServer Service Result Codes	16
Table 8 – RegisterServer2	17
Table 9 – RegisterServer2 Service Result Codes	17
Table 10 – RegisterServer2 Operation Level Result Codes	17

Table 11 – OpenSecureChannel Service Parameters	20
Table 12 – OpenSecureChannel Service Result Codes	21
Table 13 – CloseSecureChannel Service Parameters	22
Table 14 – CloseSecureChannel Service Result Codes	22
Table 15 – CreateSession Service Parameters	25
Table 16 – CreateSession Service Result Codes	26
Table 17 – ActivateSession Service Parameters	28
Table 18 – ActivateSession Service Result Codes	29
Table 19 – CloseSession Service Parameters	29
Table 20 – CloseSession Service Result Codes	29
Table 21 – Cancel Service Parameters	30
Table 22 – AddNodes Service Parameters	31
Table 23 – AddNodes Service Result Codes	31
Table 24 – AddNodes Operation Level Result Codes	32
Table 25 – AddReferences Service Parameters	33
Table 26 – AddReferences Service Result Codes	33
Table 27 – AddReferences Operation Level Result Codes	33
Table 28 – DeleteNodes Service Parameters	34
Table 29 – DeleteNodes Service Result Codes	34
Table 30 – DeleteNodes Operation Level Result Codes	35
Table 31 – DeleteReferences Service Parameters	35
Table 32 – DeleteReferences Service Result Codes	35
Table 33 – DeleteReferences Operation Level Result Codes	36
Table 34 – Browse Service Parameters	37
Table 35 – Browse Service Result Codes	38
Table 36 – Browse Operation Level Result Codes	38
Table 37 – BrowseNext Service Parameters	39
Table 38 – BrowseNext Service Result Codes	39
Table 39 – BrowseNext Operation Level Result Codes	39
Table 40 – TranslateBrowsePathsToNodeIds Service Parameters	40
Table 41 – TranslateBrowsePathsToNodeIds Service Result Codes	41
Table 42 – TranslateBrowsePathsToNodeIds Operation Level Result Codes	41
Table 43 – RegisterNodes Service Parameters	42
Table 44 – RegisterNodes Service Result Codes	42
Table 45 – UnregisterNodes Service Parameters	42
Table 46 – UnregisterNodes Service Result Codes	43
Table 47 – QueryFirst Request Parameters	45
Table 48 – QueryFirst Response Parameters	46
Table 49 – QueryFirst Service Result Codes	47
Table 50 – QueryFirst Operation Level Result Codes	47
Table 51 – QueryNext Service Parameters	48
Table 52 – QueryNext Service Result Codes	48
Table 53 – Read Service Parameters	49

Table 54 – Read Service Result Codes	49
Table 55 – Read Operation Level Result Codes	50
Table 56 – HistoryRead Service Parameters	51
Table 57 – HistoryRead Service Result Codes	52
Table 58 – HistoryRead Operation Level Result Codes	52
Table 59 – Write Service Parameters	53
Table 60 – Write Service Result Codes	54
Table 61 – Write Operation Level Result Codes	54
Table 62 – HistoryUpdate Service Parameters	55
Table 63 – HistoryUpdate Service Result Codes	55
Table 64 – HistoryUpdate Operation Level Result Codes	55
Table 65 – Call Service Parameters	57
Table 66 – Call Service Result Codes	58
Table 67 – Call Operation Level Result Codes	58
Table 68 – Call Input Argument Result Codes	58
Table 69 – CreateMonitoredItems Service Parameters	64
Table 70 – CreateMonitoredItems Service Result Codes	64
Table 71 – CreateMonitoredItems Operation Level Result Codes	65
Table 72 – ModifyMonitoredItems Service Parameters	66
Table 73 – ModifyMonitoredItems Service Result Codes	66
Table 74 – ModifyMonitoredItems Operation Level Result Codes	67
Table 75 – SetMonitoringMode Service Parameters	67
Table 76 – SetMonitoringMode Service Result Codes	67
Table 77 – SetMonitoringMode Operation Level Result Codes	67
Table 78 – SetTriggering Service Parameters	68
Table 79 – SetTriggering Service Result Codes	68
Table 80 – SetTriggering Operation Level Result Codes	69
Table 81 – DeleteMonitoredItems Service Parameters	69
Table 82 – DeleteMonitoredItems Service Result Codes	69
Table 83 – DeleteMonitoredItems Operation Level Result Codes	69
Table 84 – Subscription States.....	72
Table 85 – Subscription State Table.....	73
Table 86 – State variables and parameters	75
Table 87 – Functions.....	76
Table 88 – CreateSubscription Service Parameters.....	77
Table 89 – CreateSubscription Service Result Codes.....	78
Table 90 – ModifySubscription Service Parameters.....	79
Table 91 – ModifySubscription Service Result Codes.....	79
Table 92 – SetPublishingMode Service Parameters	80
Table 93 – SetPublishingMode Service Result Codes.....	80
Table 94 – SetPublishingMode Operation Level Result Codes.....	81
Table 95 – Publish Service Parameters	82
Table 96 – Publish Service Result Codes	82

Table 97 – Publish Operation Level Result Codes	82
Table 98 – Republish Service Parameters	83
Table 99 – Republish Service Result Codes	83
Table 100 – TransferSubscriptions Service Parameters	84
Table 101 – TransferSubscriptions Service Result Codes.....	84
Table 102 – TransferSubscriptions Operation Level Result Codes.....	84
Table 103 – DeleteSubscriptions Service Parameters	85
Table 104 – DeleteSubscriptions Service Result Codes	85
Table 105 – DeleteSubscriptions Operation Level Result Codes.....	85
Table 106 – Certificate Validation Steps	88
Table 107 – SessionlessInvoke Service Parameters.....	95
Table 108 – SessionlessInvoke Service Result Codes	95
Table 109 – ServiceLevel Ranges	102
Table 110 – Server Failover Modes	103
Table 111 – Redundancy Failover actions	103
Table 112 – ApplicationDescription	114
Table 113 – ApplicationInstanceCertificate.....	114
Table 114 – BrowseResult.....	115
Table 115 – ContentFilter Structure.....	115
Table 116 – ContentFilterResult Structure.....	116
Table 117 – ContentFilterResult Result Codes	116
Table 118 – ContentFilterResult Operand Result Codes.....	116
Table 119 – Basic FilterOperator Definition	117
Table 120 – Complex FilterOperator Definition	119
Table 121 – Wildcard characters	120
Table 122 – Conversion Rules	121
Table 123 – Data Precedence Rules	122
Table 124 – Logical AND Truth Table	122
Table 125 – Logical OR Truth Table	123
Table 126 – FilterOperand parameter Typelds.....	123
Table 127 – ElementOperand.....	123
Table 128 – LiteralOperand	123
Table 129 – AttributeOperand	124
Table 130 – SimpleAttributeOperand.....	124
Table 131 – DataValue.....	125
Table 132 – DiagnosticInfo.....	127
Table 133 – DiscoveryConfiguration parameterTypelds.....	128
Table 134 – MdnsDiscoveryConfiguration.....	128
Table 135 – EndpointDescription.....	128
Table 136 – ExpandedNodeId	129
Table 137 – ExtensibleParameter Base Type	129
Table 138 – MessageSecurityMode Values	129
Table 139 – MonitoringParameters.....	130

Table 140 – MonitoringFilter parameterTypelds.....	131
Table 141 – DataChangeFilter.....	131
Table 142 – EventFilter structure	133
Table 143 – EventFilterResult structure.....	133
Table 144 – EventFilterResult Result Codes	133
Table 145 – AggregateFilter structure	134
Table 146 – AggregateFilterResult structure.....	135
Table 147 – MonitoringMode Values	135
Table 148 – NodeAttributes parameterTypelds.....	135
Table 149 – Bit mask for specified Attributes.....	136
Table 150 – ObjectAttributes	136
Table 151 – VariableAttributes	137
Table 152 – MethodAttributes.....	137
Table 153 – ObjectTypeAttributes	137
Table 154 – VariableTypeAttributes.....	138
Table 155 – ReferenceTypeAttributes	138
Table 156 – DataTypeAttributes	138
Table 157 – ViewAttributes.....	139
Table 158 – GenericAttributes	139
Table 159 – NotificationData parameterTypelds	139
Table 160 – DataChangeNotification	140
Table 161 – EventNotificationList	140
Table 162 – StatusChangeNotification.....	140
Table 163 – NotificationMessage.....	141
Table 164 – NumericRange	141
Table 165 – QueryDataSet	142
Table 166 – ReadValueId	142
Table 167 – ReferenceDescription	143
Table 168 – RelativePath	143
Table 169 – RegisteredServer.....	144
Table 170 – RequestHeader.....	145
Table 171 – ResponseHeader	146
Table 172 – ServiceFault.....	146
Table 173 – SignatureData.....	147
Table 174 – SignedSoftwareCertificate.....	148
Table 175 – StatusCode Bit Assignments	149
Table 176 – DataValue InfoBits	150
Table 177 – Common Service Result Codes.....	151
Table 178 – Common Operation Level Result Codes.....	152
Table 179 – TimestampsToReturn Values	153
Table 180 – UserIdentityToken parameterTypelds.....	153
Table 181 – Legacy UserIdentityToken Encrypted Token Secret Format	154
Table 182 – EncryptedSecret Layout.....	156

Table 183 – EncryptedSecret DataTypes.....	156
Table 184 – RsaEncryptedSecret Structure	157
Table 185 – AnonymousIdentityToken	157
Table 186 – UserNameIdentityToken.....	158
Table 187 – EncryptionAlgorithm selection	158
Table 188 – X.509 v3 Identity Token	158
Table 189 – IssuedIdentityToken.....	159
Table 190 – UserTokenPolicy.....	159
Table 191 – ViewDescription	160
Table A.1 – RelativePath.....	161
Table A.2 – <i>RelativePath</i> Examples.....	162
Table B.1 – ContentFilter Example	164
Table B.2 – ContentFilter Example	164
Table B.3 – Example 1 NodeTypeDescription	168
Table B.4 – Example 1 ContentFilter	168
Table B.5 – Example 1 QueryDataSets	169
Table B.6 – Example 2 NodeTypeDescription.....	169
Table B.7 – Example 2 ContentFilter	170
Table B.8 – Example 2 QueryDataSets	170
Table B.9 – Example 3 - NodeTypeDescription.....	171
Table B.10 – Example 3 ContentFilter	172
Table B.11 – Example 3 QueryDataSets.....	172
Table B.12 – Example 4 NodeTypeDescription	173
Table B.13 – Example 4 ContentFilter	173
Table B.14 – Example 4 QueryDataSets.....	173
Table B.15 – Example 5 NodeTypeDescription	174
Table B.16 – Example 5 ContentFilter	174
Table B.17 – Example 5 QueryDataSets.....	174
Table B.18 – Example 6 NodeTypeDescription	175
Table B.19 – Example 6 ContentFilter	175
Table B.20 – Example 6 QueryDataSets.....	176
Table B.21 – Example 6 QueryDataSets without Additional Information.....	176
Table B.22 – Example 7 NodeTypeDescription	177
Table B.23 – Example 7 ContentFilter	177
Table B.24 – Example 7 QueryDataSets.....	177
Table B.25 – Example 8 NodeTypeDescription	178
Table B.26 – Example 8 ContentFilter	179
Table B.27 – Example 8 QueryDataSets.....	179
Table B.28 – Example 9 NodeTypeDescription	179
Table B.29 – Example 9 ContentFilter	180
Table B.30 – Example 9 QueryDataSets.....	180

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.

The main task of IEC technical committees is to prepare International Standards. However, a technical committee may propose the publication of a technical report when it has collected data of a different kind from that which is normally published as an International Standard, for example "state of the art".

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.

The text of this technical report is based on the following documents:

Enquiry draft	Report on voting
65E/XX/DTR	65E/XX/RVC

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

This third edition cancels and replaces the second edition of IEC 62541, published in 2015 and includes the following significant technical changes with respect to the previous edition:

- a) Added ability to resend all data of monitored items in a Subscription using the Resend Data Method.

- b) Added support for durable Subscriptions (lifetime of hours or days)
- c) Added Register2 and FindServersOnNetwork Services to support network-wide discovery using capability filters.
- d) Removed definition of software certificates. Will have to be defined in a future version.
- e) Extended and partially revised the redundancy definition. Added sub-range definitions for ServiceLevel and added more terms for redundancy.
- f) Added a section on how to use Authorization Services to request user access tokens.
- g) Added JSON Web Tokens (JWTs) as a new user token.
- h) Added the concept of session-less service invocation.
- i) Added a generic structure that allows passing any number of attributes to the AddNodes Service.
- j) Added requirement to protect against user identity token attacks.
- k) Added new EncryptedSecret format for user identity tokens.

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

Throughout this document and the referenced other Parts of the series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in the "Terms and definition" clause in one of the parts of the series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The italicized terms and names are also often written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example the defined term is AddressSpace instead of Address Space. This makes it easier to understand that there is a single definition for AddressSpace, not separate definitions for Address and Space.

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.

The National Committees are requested to note that for this publication the stability date is 2021.

THIS TEXT IS INCLUDED FOR THE INFORMATION OF THE NATIONAL COMMITTEES AND WILL BE DELETED AT THE PUBLICATION STAGE.

A bilingual version of this publication may be issued at a later date.

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 document using a colour printer.

OPC Unified Architecture Specification

Part 4: Services

1 Scope

This specification defines the OPC Unified Architecture (OPC UA) *Services*. The *Services* described are the collection of abstract Remote Procedure Calls (RPC) that are implemented by OPC UA *Servers* and called by OPC UA *Clients*. All interactions between OPC UA *Clients* and *Servers* occur via these *Services*. The defined *Services* are considered abstract because no particular RPC mechanism for implementation is defined in this part. 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.

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 62541-12, OPC Unified Architecture – Part 12: Discovery and Global Services

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

IEC 62541-14, OPC Unified Architecture – Part 14: PubSub

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

Active Server

Server which is currently sourcing information

Note 1 to entry: In OPC UA redundant systems, an active *Server* is the *Server* that a *Client* is using as the source of data.

3.1.2

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.

3.1.3

DiscoveryEndpoint

Endpoint that allows *Clients* access to *Discovery Services* without security

Note 1 to entry: A *DiscoveryEndpoint* allows access to *Discovery Services* without a *Session* and without message security.

3.1.4

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* must include a *HostName*.

3.1.5

Failed Server

Server that is not operational.

Note 1 to entry: In OPC UA redundant system, a failed *Server* is a *Server* that is unavailable or is not able to serve data.

3.1.6

Failover

act of switching the source or target of information.

Note 1 to entry: In OPC UA redundant systems, a failover is the act of a *Client* switching away from a failed or degraded *Server* to another *Server* in the redundant set (*Server failover*). In some cases a *Client* may have no knowledge of a failover action occurring (transparent redundancy). The act of an alternate *Client* replacing an existing failed or degraded *Client* connection to a *Server* (*Client failover*).

3.1.7

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.

3.1.8

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.

3.1.9

Security Token

identifier for a cryptographic key set

Note 1 to entry: All *Security Tokens* belong to a security context. For OPC UA the security context is the *SecureChannel*.

3.1.10

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 6.4.

3.1.11

Redundancy

the presence of duplicate components enabling the continued operation after a failure of an OPC UA component

Note 1 This may apply to *Servers*, *Clients* or networks.

3.1.12

Redundant Server Set

two or more *Servers* that are redundant with each other

Note 1 A redundant server set is a group of *Servers* that are configured to provide *Redundancy*. These *Servers* have requirements related to the address space and provide failovers.

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 <i>Service</i>
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 <i>Service</i>

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.

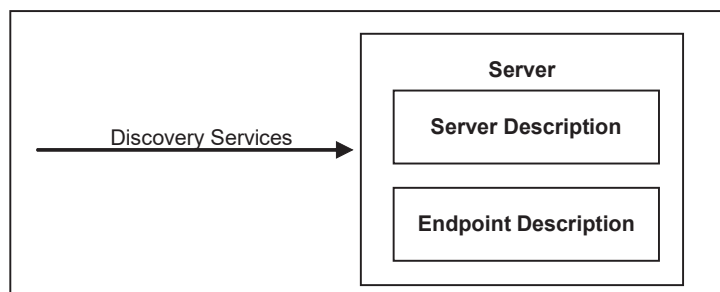
4 Overview

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*.

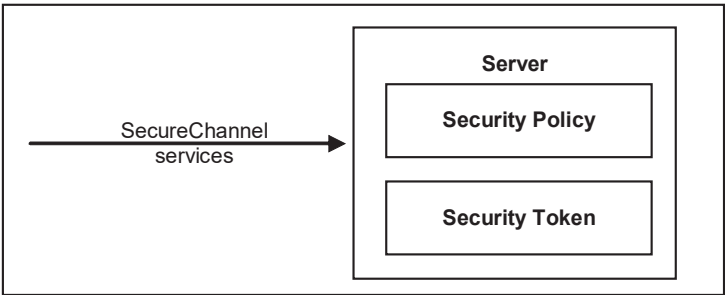


Figure 2 – SecureChannel Service Set

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*.

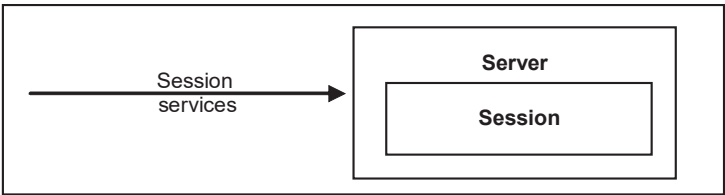


Figure 3 – Session Service Set

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

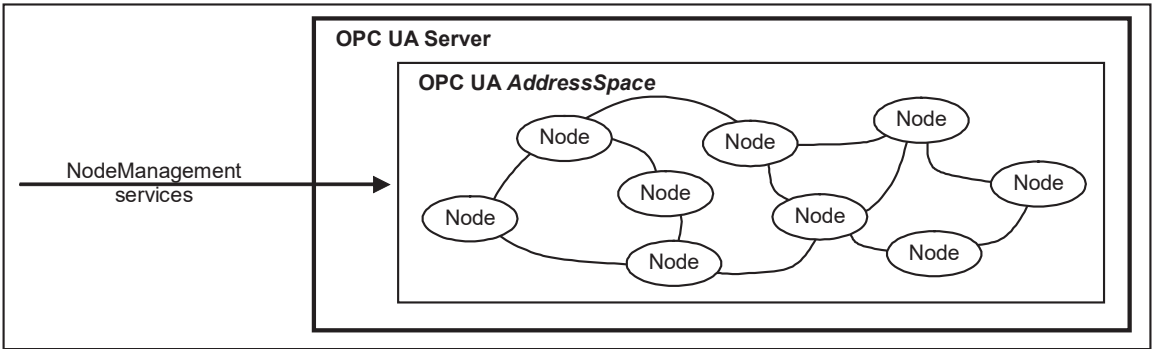


Figure 4 – NodeManagement Service Set

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*.

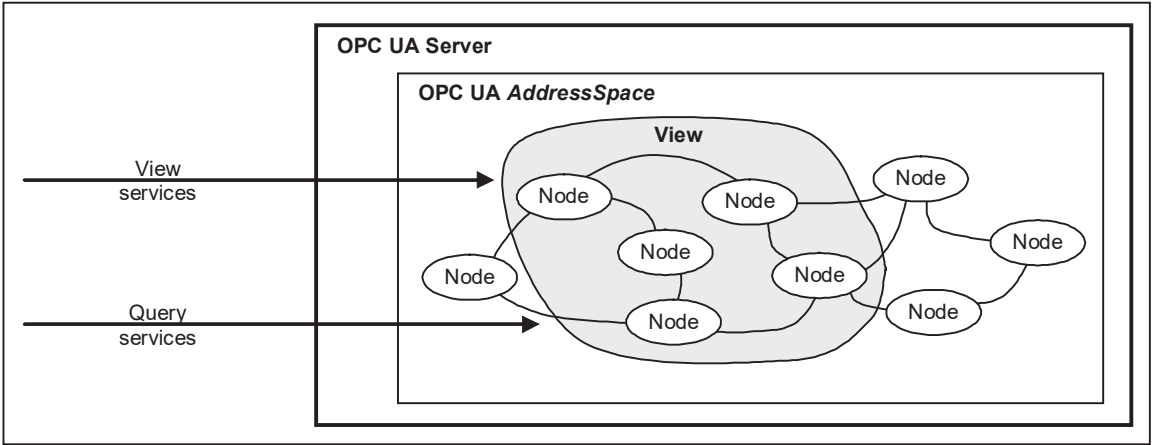


Figure 5 – View Service Set

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*.

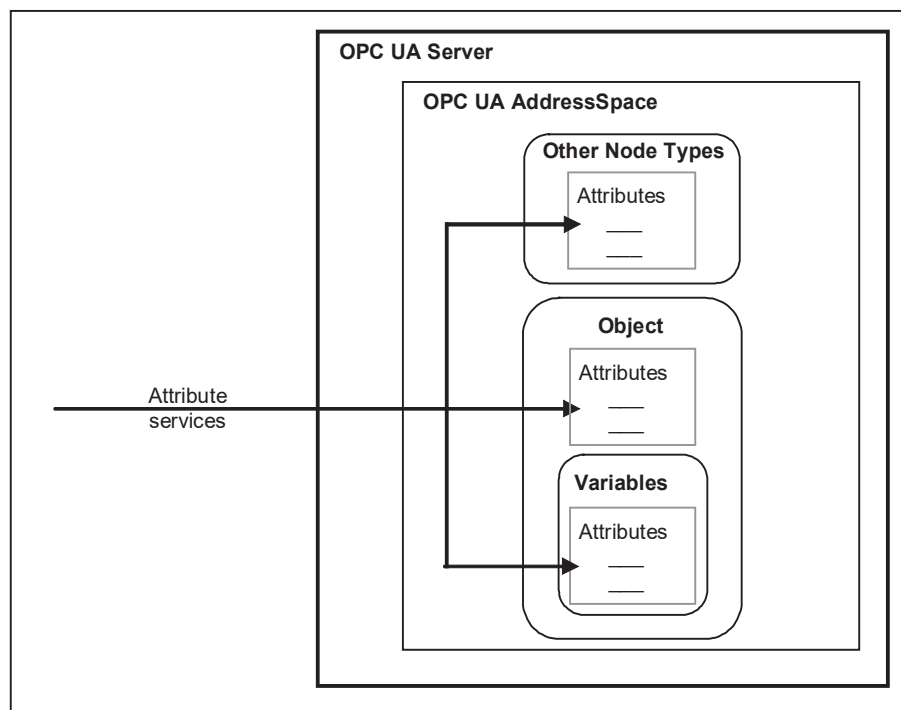


Figure 6 – Attribute Service Set

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.

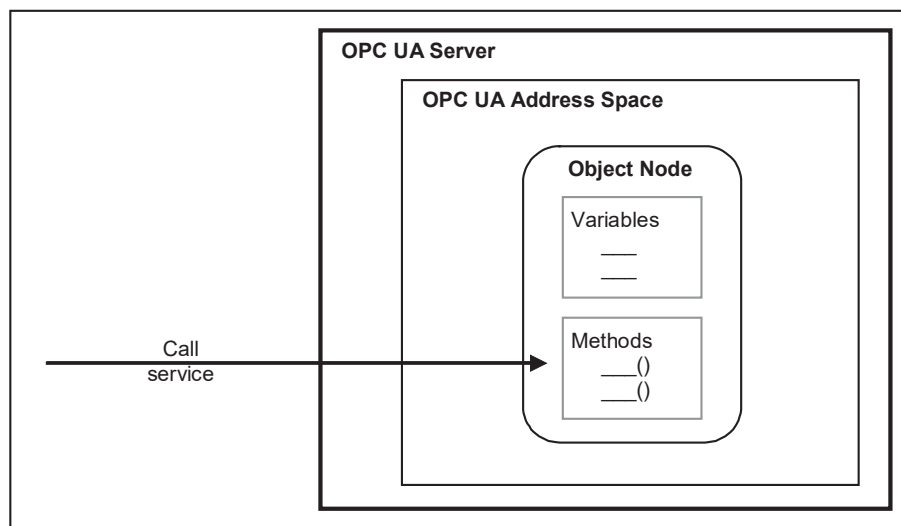


Figure 7 – Method Service Set

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.

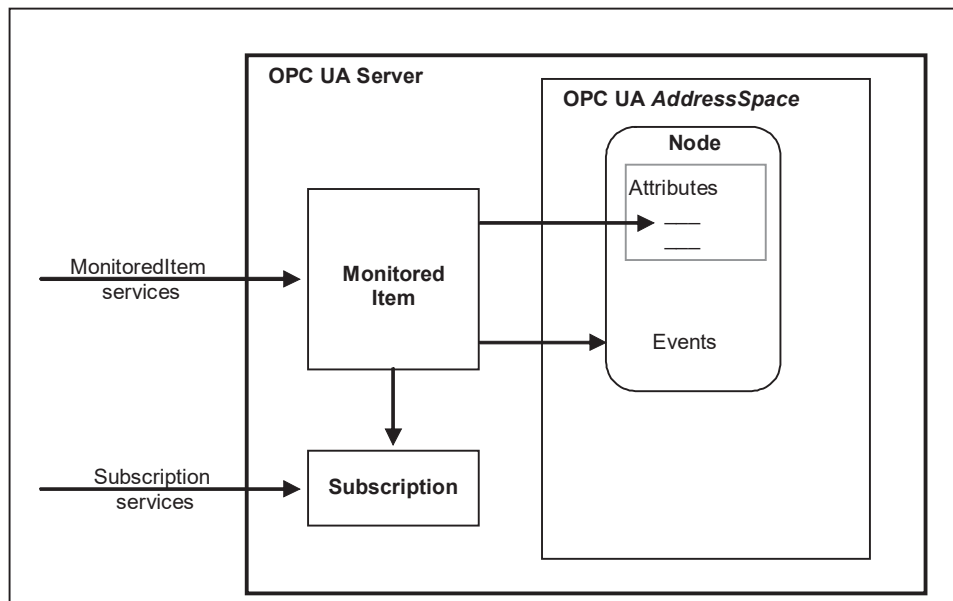


Figure 8 – MonitoredItem and Subscription Service Sets

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 6.5 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 7.28 and contains common request parameters such as *authenticationToken*, *timestamp* and *requestHandle*.

The *ResponseHeader* structure is defined in 7.29 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 7.34).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see 7.29). 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 *StatusCodes*.

The following cases define the use of these parameters.

- l) A bad code is returned in *serviceResult* if the *Service* itself failed. In this case, a *ServiceFault* is returned. The *ServiceFault* is defined in 7.30.
- m) 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 177 and Table 178. 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 communication fault occurs.

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

Every *Server* shall have a *DiscoveryEndpoint* 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 *DiscoveryEndpoint*.

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 discovery process using *FindServers* is illustrated in Figure 9. The establishment of a *SecureChannel* (with *MessageSecurityMode* NONE) for *FindServers* and *GetEndpoints* is omitted from the figure for clarity.

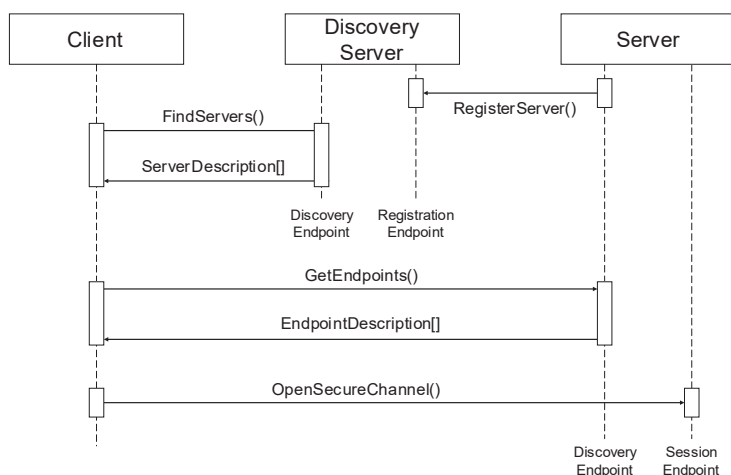


Figure 9 – Discovery process

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

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

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

A *Client* shall be careful when using the information returned from a *DiscoveryEndpoint* since it has no security. A *Client* does this by comparing the information returned from the *DiscoveryEndpoint* 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 *DiscoveryEndpoint* 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.

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 *DiscoveryEndpoint* 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 Server 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 message security but it may require transport layer security.

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

This *Service* can be used without security and it is therefore vulnerable to 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.

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 <i>authenticationToken</i> is always null. The <i>authenticationToken</i> shall be ignored if it is provided. The type <i>RequestHeader</i> is defined in 7.28.
endpointUrl	String	The network address that the <i>Client</i> used to access the <i>DiscoveryEndpoint</i> . The <i>Server</i> uses this information for diagnostics and to determine what URLs to return in the response. The <i>Server</i> should return a suitable default URL if it does not recognize the <i>HostName</i> in the URL.
localeIds []	LocaleId	List of locales to use. The <i>Server</i> should return the <i>applicationName</i> in the <i>ApplicationDescription</i> defined in 7.1 using one of locales specified. If the <i>Server</i> supports more than one of the requested locales then the <i>Server</i> shall use the locale that appears first in this list. If the <i>Server</i> does not support any of the requested locales it chooses an appropriate default locale. The <i>Server</i> chooses an appropriate default locale if this list is empty.
serverUri []	String	List of <i>Servers</i> to return. All known <i>Servers</i> are returned if the list is empty. A <i>serverUri</i> matches the <i>applicationUri</i> from the <i>ApplicationDescription</i> defined in 7.1.
Response		
responseHeader	ResponseHeader	Common response parameters. The <i>ResponseHeader</i> type is defined in 7.29.
servers []	ApplicationDescription	List of <i>Servers</i> that meet criteria specified in the request. This list is empty if no <i>Servers</i> meet the criteria. The <i>ApplicationDescription</i> type is defined in 7.1.

5.4.2.3 Service results

Common *StatusCodes* are defined in Table 177.

5.4.3 FindServersOnNetwork

5.4.3.1 Description

This *Service* returns the *Servers* known to a *Discovery Server*. Unlike *FindServers*, this *Service* is only implemented by *Discovery Servers*.

The *Client* may reduce the number of results returned by specifying filter criteria. An empty list is returned if no *Server* matches the criteria specified by the *Client*.

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

Each time the *Discovery Server* creates or updates a record in its cache it shall assign a monotonically increasing identifier to the record. This allows *Clients* to request records in batches by specifying the identifier for the last record received in the last call to *FindServersOnNetwork*. To support this the *Discovery Server* shall return records in numerical order starting from the lowest record identifier. The *Discovery Server* shall also return the last time the counter was reset for example due to a restart of the *Discovery Server*. If a *Client* detects that this time is more recent than the last time the *Client* called the *Service* it shall call the *Service* again with a *startingRecordId* of 0.

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.

5.4.3.2 Parameters

Table 4 defines the parameters for the *Service*.

Table 4 – FindServersOnNetwork Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The <i>authenticationToken</i> shall be ignored if it is provided. The type <i>RequestHeader</i> is defined in 7.28.
startingRecordId	Counter	Only records with an identifier greater than this number will be returned. Specify 0 to start with the first record in the cache.
maxRecordsToReturn	UInt32	The maximum number of records to return in the response. 0 indicates that there is no limit.
serverCapabilityFilter[]	String	List of <i>Server</i> capability filters. The set of allowed <i>Server</i> capabilities are defined in IEC 62541-12. Only records with all of the specified <i>Server</i> capabilities are returned. The comparison is case insensitive. If this list is empty then no filtering is performed.
Response		
responseHeader	ResponseHeader	Common response parameters. The <i>ResponseHeader</i> type is defined in 7.29.
lastCounterResetTime	UtcTime	The last time the counters were reset.
servers[]	ServerOnNetwork	List of DNS service records that meet criteria specified in the request. This list is empty if no <i>Servers</i> meet the criteria.
recordId	UInt32	A unique identifier for the record. This can be used to fetch the next batch of <i>Servers</i> in a subsequent call to <i>FindServersOnNetwork</i> .
serverName	String	The name of the <i>Server</i> specified in the mDNS announcement (see IEC 62541-12). This may be the same as the <i>ApplicationName</i> for the <i>Server</i> .
discoveryUrl	String	The URL of the <i>DiscoveryEndpoint</i> .
serverCapabilities	String[]	The set of <i>Server</i> capabilities supported by the <i>Server</i> . The set of allowed <i>Server</i> capabilities are defined in IEC 62541-12.

5.4.3.3 Service results

Common *StatusCodes* are defined in Table 177.

5.4.4 GetEndpoints

5.4.4.1 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 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 5.4.4.2.

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.

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.

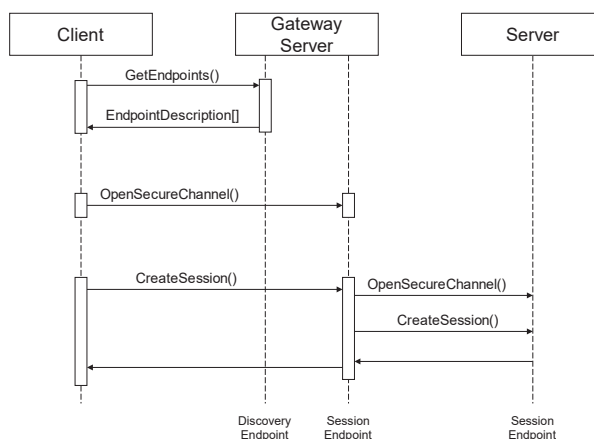


Figure 10 – Using a Gateway Server

5.4.4.2 Parameters

Table 5 defines the parameters for the *Service*.

Table 5 – GetEndpoints Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The <i>authenticationToken</i> shall be ignored if it is provided. The type <i>RequestHeader</i> is defined in 7.28.
endpointUrl	String	The network address that the <i>Client</i> used to access the <i>DiscoveryEndpoint</i> . The <i>Server</i> uses this information for diagnostics and to determine what URLs to return in the response. The <i>Server</i> should return a suitable default URL if it does not recognize the <i>HostName</i> 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 <i>Transport Profile</i> that the returned <i>Endpoints</i> shall support. IEC 62541-7 defines URIs for the <i>Transport Profiles</i> . All <i>Endpoints</i> are returned if the list is empty. If the URI is a URL, this URL may have a query string appended. The <i>Transport Profiles</i> that support query strings are defined in IEC 62541-7.
Response		
responseHeader	ResponseHeader	Common response parameters. The <i>ResponseHeader</i> type is defined in 7.29.
Endpoints []	EndpointDescription	List of <i>Endpoints</i> that meet criteria specified in the request. This list is empty if no <i>Endpoints</i> meet the criteria. The <i>EndpointDescription</i> type is defined in 7.10.

5.4.4.3 Service Results

Common *StatusCodes* are defined in Table 177.

5.4.5 RegisterServer

5.4.5.1 Description

This Service is implemented by *Discovery Servers*.

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*.

This *Service* can only be invoked via *SecureChannels* that support *Client* authentication (i.e. HTTPS cannot be used to call this *Service*).

A *Server* only provides its *serverUri* and the URLs of the *DiscoveryEndpoints* 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.

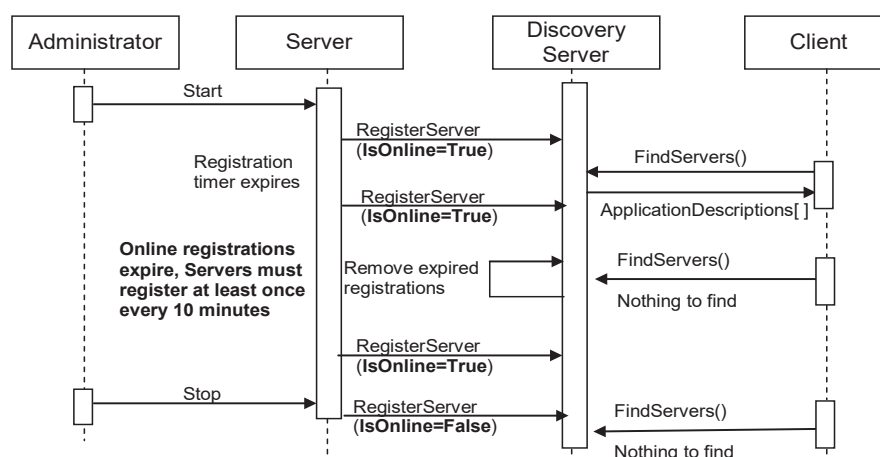


Figure 11 – The Registration Process – Manually Launched Servers

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

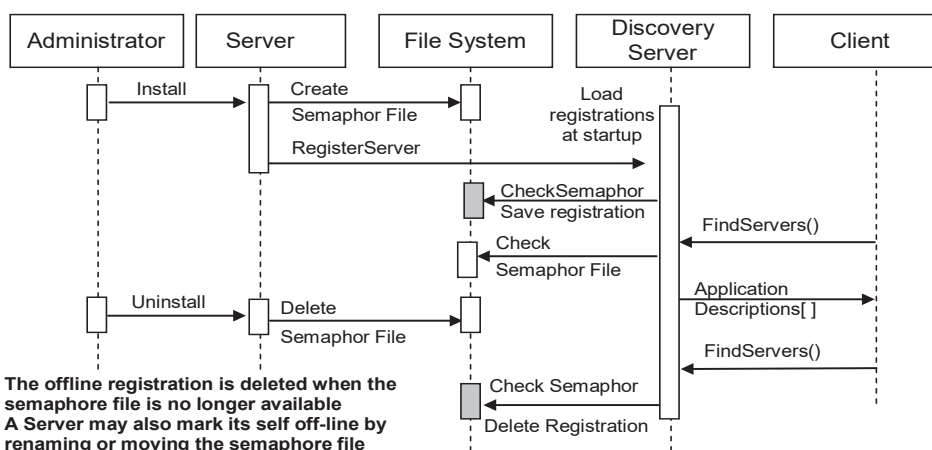


Figure 12 – The Registration Process – Automatically Launched Servers

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 period of the recurring registration 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.

5.4.5.2 Parameters

Table 6 defines the parameters for the *Service*.

Table 6 – RegisterServer Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The type <i>RequestHeader</i> is defined in 7.28.
Server	RegisteredServer	The <i>Server</i> to register. The type <i>RegisteredServer</i> is defined in 7.27.
Response		
ResponseHeader	ResponseHeader	Common response parameters. The type <i>ResponseHeader</i> is defined in 7.29.

5.4.5.3 Service Results

Table 7 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 7 – RegisterServer Service Result Codes

Symbolic Id	Description
Bad_InvalidArgument	See Table 177 for the description of this result code.
Bad_ServerUrlInvalid	See Table 177 for the description of this result code.
Bad_ServerNameMissing	No <i>ServerName</i> was specified.
Bad_DiscoveryUrlMissing	No discovery URL was specified.
Bad_SemaphoreFileMissing	The semaphore file specified is not valid.

5.4.6 RegisterServer2

5.4.6.1 Description

This *Service* is implemented by *Discovery Servers*.

This *Service* allows a *Server* to register its *DiscoveryUrls* and capabilities with a *Discovery Server*. It extends the registration information from *RegisterServer* with information necessary for *FindServersOnNetwork*. This *Service* will be called by a *Server* or a separate configuration utility. *Clients* will not use this *Service*.

Servers that support *RegisterServer2* shall try to register with the *Discovery Server* using this *Service* and shall fall back to *RegisterServer* if *RegisterServer2* fails with the status *Bad_ServiceUnsupported*.

A *Discovery Server* that implements this *Service* needs to assign unique record ids each time this *Service* is called. See 5.4.3 for more details.

This *Service* can only be invoked via *SecureChannels* that support *Client* authentication (i.e. HTTPS cannot be used to call this *Service*).

5.4.6.2 Parameters

Table 8 defines the parameters for the *Service*.

Table 8 – RegisterServer2

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The type <i>RequestHeader</i> is defined in 7.28.
Server	RegisteredServer	The <i>Server</i> to register. The type <i>RegisteredServer</i> is defined in 7.27.
discoveryConfiguration []	ExtensibleParameter DiscoveryConfiguration	Additional configuration settings for the <i>Server</i> to register. The <i>discoveryConfiguration</i> is an extensible parameter type defined in 7.9. Discovery <i>Servers</i> that do not understand a configuration shall return <i>Bad_NotSupported</i> for this configuration.
Response		
ResponseHeader	ResponseHeader	Common response parameters. The type <i>ResponseHeader</i> is defined in 7.29.
configurationResults []	StatusCode	List of results for the <i>discoveryConfiguration</i> parameters.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>discoveryConfiguration</i> parameters. 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.

5.4.6.3 Service results

Table 9 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 9 – RegisterServer2 Service Result Codes

Symbolic Id	Description
Bad_InvalidArgument	See Table 177 for the description of this result code.
Bad_ServerUriInvalid	See Table 177 for the description of this result code.
Bad_ServerNameMissing	No <i>ServerName</i> was specified.
Bad_DiscoveryUriMissing	No discovery URL was specified.
Bad_SemaphoreFileMissing	The semaphore file specified is not valid.
Bad_ServiceUnsupported	See Table 177 for the description of this result code.

5.4.6.4 StatusCodes

Table 10 defines values for the operation level *statusCode* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 178.

Table 10 – RegisterServer2 Operation Level Result Codes

Symbolic Id	Description
Bad_NotSupported	See Table 178 for the description of this result code.

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*.

OPC UA Application is built. For example, an *OPC UA Server* may be built on a stack that allows applications to establish a *SecureChannel* using HTTPS. In these cases, the *OPC UA Application* shall verify that the *Message* it received was in the context of an HTTPS connection. 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.

Logical connections may be initiated by the *Client* or by the *Server* as described in IEC 62541-6. After the connection is initiated, the *SecureChannel* is opened and closed by the *Client* using the *SecureChannel Services*.

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*.

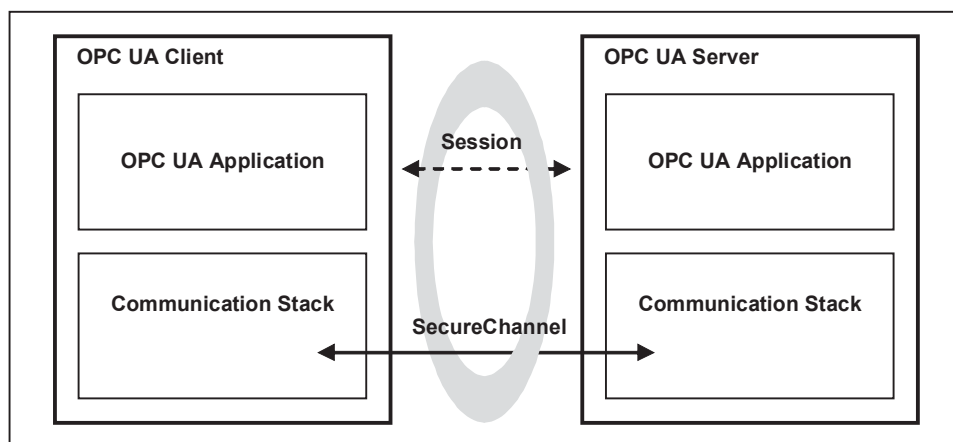


Figure 13 – SecureChannel and Session Services

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 private key. These *Messages* shall always be encrypted. If the transport layer does not provide encryption, then these *Messages* shall be encrypted with the receiver's public key. These requirements for *OpenSecureChannel* only apply if the *securityPolicyUri* is not None.

If the protocol defined in IEC 62541-6 requires that *Application Instance Certificates* are used in the *OpenSecureChannel Service*, then *Clients* and *Servers* shall verify that the same *Certificates* are 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 11 defines the parameters for the *Service*.

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

Table 11 – OpenSecureChannel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The type <i>RequestHeader</i> is defined in 7.28.
clientCertificate	ApplicationInstanceCertificate	A <i>Certificate</i> that identifies the <i>Client</i> . The <i>OpenSecureChannel</i> request shall be signed with the private key for this <i>Certificate</i> . The <i>ApplicationInstanceCertificate</i> type is defined in 7.2. If the <i>securityPolicyUri</i> is None, the <i>Server</i> shall ignore the <i>ApplicationInstanceCertificate</i> .
requestType	Enum SecurityToken RequestType	The type of <i>SecurityToken</i> request: An enumeration that shall be one of the following: ISSUE_0 creates a new <i>SecurityToken</i> for a new <i>SecureChannel</i> . RENEW_1 creates a new <i>SecurityToken</i> for an existing <i>SecureChannel</i> .
secureChannelId	BaseDataType	The identifier for the <i>SecureChannel</i> that the new token should belong to. This parameter shall be null when creating a new <i>SecureChannel</i> . The concrete security protocol definition in IEC 62541-6 chooses the concrete <i>DataType</i> .
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type <i>MessageSecurityMode</i> type is defined in 7.15. A <i>SecureChannel</i> may have to be created even if the <i>securityMode</i> is NONE. The exact behaviour depends on the mapping used and is described in the IEC 62541-6.
securityPolicyUri	String	The URI for <i>SecurityPolicy</i> to use when securing messages sent over the <i>SecureChannel</i> . The set of known URIs and the <i>SecurityPolicies</i> 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 <i>clientNonce</i> shall be generated for each time a <i>SecureChannel</i> is renewed. This parameter shall have a length equal to the <i>SecureChannelNonceLength</i> defined for the <i>SecurityPolicy</i> in IEC 62541-7. The <i>SecurityPolicy</i> is identified by the <i>securityPolicyUri</i> .
requestedLifetime	Duration	The requested lifetime, in milliseconds, for the new <i>SecurityToken</i> . It specifies when the <i>Client</i> expects to renew the <i>SecureChannel</i> by calling the <i>OpenSecureChannel Service</i> again. If a <i>SecureChannel</i> is not renewed, then all <i>Messages</i> sent using the current <i>SecurityTokens</i> shall be rejected by the receiver. Several cryptanalytic attacks become easier as more material encrypted with a specific key is available. By limiting the amount of data processed using a particular key, those attacks are made more difficult. Therefore the volume of data exchanged between <i>Client</i> and <i>Server</i> must be limited by establishing a new <i>SecurityToken</i> after the lifetime. The setting of the requested lifetime depends on the expected number of exchanged messages and their size in the lifetime. A higher volume of data requires shorter lifetime.

Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> type definition).
securityToken	ChannelSecurityToken	Describes the new <i>SecurityToken</i> issued by the <i>Server</i> . This structure is defined in-line with the following indented items.
channelId	BaseDataType	A unique identifier for the <i>SecureChannel</i> . This is the identifier that shall be supplied whenever the <i>SecureChannel</i> is renewed. The concrete security protocol definition in IEC 62541-6 chooses the concrete <i>DataType</i> .
tokenId	ByteString	A unique identifier for a single <i>SecurityToken</i> within the channel. This is the identifier that shall be passed with each <i>Message</i> secured with the <i>SecurityToken</i> .
createdAt	UtcTime	The time when the <i>SecurityToken</i> was created.
revisedLifetime	Duration	The lifetime of the <i>SecurityToken</i> in milliseconds. The UTC expiration time for the token may be calculated by adding the lifetime to the <i>createdAt</i> time.
serverNonce	ByteString	A random number that shall not be used in any other request. A new <i>serverNonce</i> shall be generated for each time a <i>SecureChannel</i> is renewed. This parameter shall have a length equal to the <i>SecureChannelNonceLength</i> defined for the <i>SecurityPolicy</i> in IEC 62541-7. The <i>SecurityPolicy</i> is identified by the <i>securityPolicyUri</i> .

5.5.2.3 Service results

Table 12 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 12 – OpenSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecurityChecksFailed	See Table 177 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 177 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 177 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 177 for the description of this result code.
Bad_CertificateUriInvalid	See Table 177 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 177 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 177 for the description of this result code.
Bad_CertificateUntrusted	See Table 177 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 177 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 177 for the description of this result code.
Bad_CertificateRevoked	See Table 177 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 177 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 <i>Server</i> .
Bad_SecurityPolicyRejected	The security policy does not meet the requirements set by the <i>Server</i> .
Bad_SecureChannelIdInvalid	See Table 177 for the description of this result code.
Bad_NonceInvalid	See Table 177 for the description of this result code. A <i>Server</i> shall check the minimum length of the <i>Client</i> nonce and return this status if the length is below 32 bytes. A check for duplicated nonce can only be done in <i>OpenSecureChannel</i> 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 13 defines the parameters for the *Service*.

Specific protocol mappings defined in IEC 62541-6 may choose to omit the response.

Table 13 – CloseSecureChannel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The type <i>RequestHeader</i> is defined in 7.28.
secureChannelId	BaseDataType	The identifier for the <i>SecureChannel</i> to close. The concrete security protocol definition in IEC 62541-6 chooses the concrete <i>DataType</i> .
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> definition).

5.5.3.3 Service results

Table 14 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 14 – CloseSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 177 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 AddressSpace*. 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 7.31 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 *DiscoveryEndpoint* 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.

A *Server* application should limit the number of *Sessions*. To protect against misbehaving *Clients* and denial of service attacks, the *Server* shall close the oldest *Session* that is not activated before reaching the maximum number of supported *Sessions*.

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.

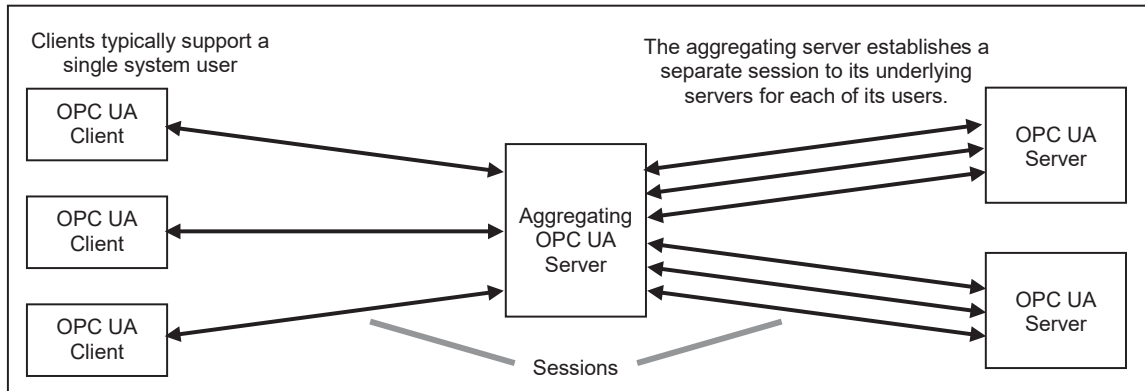


Figure 14 – Multiplexing Users on a Session

5.6.2.2 Parameters

Table 15 defines the parameters for the *Service*.

Table 15 – CreateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The <i>authenticationToken</i> is always null. The type <i>RequestHeader</i> is defined in 7.28.
clientDescription	Application Description	Information that describes the <i>Client</i> application. The type <i>ApplicationDescription</i> is defined in 7.1.
serverUri	String	This value is only specified if the <i>EndpointDescription</i> has a <i>gatewayServerUri</i> . This value is the <i>applicationUri</i> from the <i>EndpointDescription</i> which is the <i>applicationUri</i> for the underlying <i>Server</i> . The type <i>EndpointDescription</i> is defined in 7.10.
endpointUrl	String	The network address that the <i>Client</i> used to access the <i>Session Endpoint</i> . The <i>HostName</i> portion of the URL should be one of the <i>HostNames</i> for the application that are specified in the <i>Server's ApplicationInstanceCertificate</i> (see 7.2). The <i>Server</i> shall raise an <i>AuditUrlMismatchEventType</i> event if the URL does not match the <i>Server's HostNames</i> . <i>AuditUrlMismatchEventType</i> event type is defined in IEC 62541-5. The <i>Server</i> uses this information for diagnostics and to determine the set of <i>EndpointDescriptions</i> to return in the response.
sessionName	String	Human readable string that identifies the <i>Session</i> . The <i>Server</i> makes this name and the <i>sessionId</i> visible in its <i>AddressSpace</i> for diagnostic purposes. The <i>Client</i> should provide a name that is unique for the instance of the <i>Client</i> . If this parameter is not specified the <i>Server</i> 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 <i>Server</i> shall use this value to prove possession of its <i>Application Instance Certificate</i> in the response.
clientCertificate	ApplicationInstance Certificate	The <i>Application Instance Certificate</i> issued to the <i>Client</i> . The <i>ApplicationInstanceCertificate</i> type is defined in 7.2. If the <i>securityPolicyUri</i> is None, the <i>Server</i> shall ignore the <i>ApplicationInstanceCertificate</i> .
Requested SessionTimeout	Duration	Requested maximum number of milliseconds that a <i>Session</i> should remain open without activity. If the <i>Client</i> fails to issue a <i>Service</i> request within this interval, then the <i>Server</i> shall automatically terminate the <i>Client Session</i> .
maxResponse MessageSize	UInt32	The maximum size, in bytes, for the body of any response message. The <i>Server</i> should return a <i>Bad_ResponseTooLarge</i> service fault if a response message exceeds this limit. The value zero indicates that this parameter is not used. The transport protocols defined in IEC 62541-6 may imply minimum message sizes. More information on the use of this parameter is provided in 5.3.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> type).
sessionId	NodeId	A unique <i>NodeId</i> assigned by the <i>Server</i> to the <i>Session</i> . This identifier is used to access the diagnostics information for the <i>Session</i> in the <i>Server AddressSpace</i> . It is also used in the audit logs and any events that report information related to the <i>Session</i> . The <i>Session</i> diagnostic information is described in IEC 62541-5. Audit logs and their related events are described in 6.5.
authentication Token	Session AuthenticationToken	A unique identifier assigned by the <i>Server</i> to the <i>Session</i> . This identifier shall be passed in the <i>RequestHeader</i> of each request and is used with the <i>SecureChannelId</i> to determine whether a <i>Client</i> has access to the <i>Session</i> . This identifier shall not be reused in a way that the <i>Client</i> or the <i>Server</i> has a chance of confusing them with a previous or existing <i>Session</i> . The <i>SessionAuthenticationToken</i> type is described in 7.31.
revisedSession Timeout	Duration	Actual maximum number of milliseconds that a <i>Session</i> shall remain open without activity. The <i>Server</i> should attempt to honour the <i>Client</i> request for this parameter, but may negotiate this value up or down to meet its own constraints.
serverNonce	ByteString	A random number that should never be used in any other request. This number shall have a minimum length of 32 bytes. The <i>Client</i> shall use this value to prove possession of its <i>Application Instance Certificate</i> in the <i>ActivateSession</i> request. This value may also be used to prove possession of the <i>userIdentityToken</i> it specified in the <i>ActivateSession</i> request.
serverCertificate	ApplicationInstance Certificate	The <i>Application Instance Certificate</i> issued to the <i>Server</i> . A <i>Server</i> shall prove possession by using the private key to sign the <i>Nonce</i> provided by the <i>Client</i> in the request. The <i>Client</i> shall verify that this <i>Certificate</i> is the same as the one it used to create the <i>SecureChannel</i> . The <i>ApplicationInstanceCertificate</i> type is defined in 7.2. If the <i>securityPolicyUri</i> is NONE and none of the <i>UserTokenPolicies</i> requires encryption, the <i>Client</i> shall ignore the <i>ApplicationInstanceCertificate</i> .
serverEndpoints []	EndpointDescription	List of <i>Endpoints</i> that the <i>Server</i> supports. The <i>Server</i> shall return a set of <i>EndpointDescriptions</i> available for the <i>serverUri</i> specified in the request. The <i>EndpointDescription</i> type is defined in 7.10. The <i>Client</i>

Name	Type	Description
		shall verify this list with the list from a <i>DiscoveryEndpoint</i> if it used a <i>DiscoveryEndpoint</i> to fetch the <i>EndpointDescriptions</i> . It is recommended that <i>Servers</i> only include the <i>server.applicationUri</i> , <i>endpointUrl</i> , <i>securityMode</i> , <i>securityPolicyUri</i> , <i>userIdentityTokens</i> , <i>transportProfileUri</i> and <i>securityLevel</i> with all other parameters set to null. Only the recommended parameters shall be verified by the client.
serverSoftwareCertificates []	SignedSoftwareCertificate	This parameter is deprecated and the array shall be empty. The <i>SoftwareCertificates</i> are provided in the <i>Server AddressSpace</i> as defined in IEC 62541-5.
serverSignature	SignatureData	This is a signature generated with the private key associated with the <i>serverCertificate</i> . This parameter is calculated by appending the <i>clientNonce</i> to the <i>clientCertificate</i> and signing the resulting sequence of bytes. If the <i>clientCertificate</i> contains a chain, the signature calculation shall be done only with the leaf <i>Certificate</i> . For backward compatibility a <i>Client</i> shall check the signature with the full chain if the check with the leaf <i>Certificate</i> fails. The <i>SignatureAlgorithm</i> shall be the <i>AsymmetricSignatureAlgorithm</i> specified in the <i>SecurityPolicy</i> for the <i>Endpoint</i> . The <i>SignatureData</i> type is defined in 7.32.
maxRequestMessageSize	UInt32	The maximum size, in bytes, for the body of any request message. The <i>Client Communication Stack</i> should return a <i>Bad_RequestTooLarge</i> error to the application if a request message exceeds this limit. The value zero indicates that this parameter is not used. See IEC 62541-6 for protocol specific minimum or default values. 5.3 provides more information on the use of this parameter.

5.6.2.3 Service results

Table 16 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 16 – CreateSession Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 177 for the description of this result code.
Bad_NonceInvalid	See Table 177 for the description of this result code. A <i>Server</i> shall check the minimum length of the <i>Client</i> nonce and return this status if the length is below 32 bytes. A check for a duplicated nonce is optional and requires access to the nonce used to create the secure channel.
Bad_SecurityChecksFailed	See Table 177 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 177 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 177 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 177 for the description of this result code.
Bad_CertificateUriInvalid	See Table 177 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 177 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 177 for the description of this result code.
Bad_CertificateUntrusted	See Table 177 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 177 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 177 for the description of this result code.
Bad_CertificateRevoked	See Table 177 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 177 for the description of this result code.
Bad_TooManySessions	The <i>Server</i> has reached its maximum number of <i>Sessions</i> .
Bad_ServerUriInvalid	See Table 177 for the description of this result code.

5.6.3 ActivateSession

5.6.3.1 Description

This *Service* is used by the *Client* to specify the identity of the user associated with the *Session*. This *Service* request shall be issued by the *Client* before it issues any *Service* request other than *CloseSession* 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*.

Servers shall take proper measures to protect against attacks on user identity tokens. Such an attack is assumed if repeated connection attempts with invalid user identity tokens happen. One option is to lock out an OPC UA *Client* for a period of time if the user identity token validation fails several times. The OPC UA *Client* is either detected by IP address for unsecured connections or by the *ApplicationInstanceUri* for secured connections. Another option is delaying the *Service* response when the validation of a user identity fails. This delay time could be increased with repeated failures. Sporadic failures shall not delay connections with valid tokens.

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 17 defines the parameters for the *Service*.

Table 17 – ActivateSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters. The type <i>RequestHeader</i> is defined in 7.28.
clientSignature	SignatureData	This is a signature generated with the private key associated with the <i>clientCertificate</i> . This parameter is calculated by appending the <i>serverNonce</i> to the <i>serverCertificate</i> and signing the resulting sequence of bytes. If the <i>serverCertificate</i> contains a chain, the signature calculation shall be done only with the leaf <i>Certificate</i> . For backward compatibility a <i>Server</i> shall check the signature with the full chain if the check with the leaf <i>Certificate</i> fails. The <i>SignatureAlgorithm</i> shall be the <i>AsymmetricSignatureAlgorithm</i> specified in the <i>SecurityPolicy</i> for the <i>Endpoint</i> . The <i>SignatureData</i> type is defined in 7.32.
clientSoftwareCertificates []	SignedSoftwareCertificate	Reserved for future use. The <i>SignedSoftwareCertificate</i> type is defined in 7.33.
localeIds []	LocaleId	List of locale ids in priority order for localized strings. The first <i>LocaleId</i> in the list has the highest priority. If the <i>Server</i> returns a localized string to the <i>Client</i> , the <i>Server</i> 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 <i>Client</i> fails to specify at least one locale id, the <i>Server</i> shall use any that it has. This parameter only needs to be specified during the first call to <i>ActivateSession</i> during a single application <i>Session</i> . If it is not specified the <i>Server</i> shall keep using the current <i>localeIds</i> for the <i>Session</i> .
userIdentityToken	Extensible Parameter UserIdentityToken	The credentials of the user associated with the <i>Client</i> application. The <i>Server</i> uses these credentials to determine whether the <i>Client</i> should be allowed to activate a <i>Session</i> and what resources the <i>Client</i> has access to during this <i>Session</i> . The <i>UserIdentityToken</i> is an extensible parameter type defined in 7.36. The <i>EndpointDescription</i> specifies what <i>UserIdentityTokens</i> the <i>Server</i> shall accept. Null or empty user token shall always be interpreted as anonymous.
userTokenSignature	SignatureData	If the <i>Client</i> 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 null. The <i>SignatureAlgorithm</i> depends on the identity token type. The <i>SignatureData</i> type is defined in 7.32.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> 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 <i>Client</i> shall use this value to prove possession of its <i>Application Instance Certificate</i> in the next call to <i>ActivateSession</i> request.
results []	StatusCode	List of validation results for the <i>SoftwareCertificates</i> (see 7.34 for <i>StatusCode</i> definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information associated with <i>SoftwareCertificate</i> validation errors (see 7.8 for <i>DiagnosticInfo</i> 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 18 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 18 – ActivateSession Service Result Codes

Symbolic Id	Description
Bad_IdentityTokenInvalid	See Table 177 for the description of this result code.
Bad_IdentityTokenRejected	See Table 177 for the description of this result code.
Bad_UserAccessDenied	See Table 177 for the description of this result code.
Bad_ApplicationSignatureInvalid	The signature provided by the <i>Client</i> application is missing or invalid.
Bad_UserSignatureInvalid	The user token signature is missing or invalid.
Bad_NoValidCertificates	The <i>Client</i> did not provide at least one <i>Software Certificate</i> that is valid and meets the profile requirements for the <i>Server</i> .
Bad_IdentityChangeNotSupported	The <i>Server</i> 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:

- It stops accepting requests for the *Session*. All subsequent requests received for the *Session* are discarded.
- 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.
- It removes the entry for the *Client* in its *SessionDiagnosticsArray Variable*.

When the *CloseSession Service* is called before the *Session* is successfully activated, the *Server* shall reject the request if the *SecureChannel* is not the same as the one associated with the *CreateSession* request.

5.6.4.2 Parameters

Table 19 defines the parameters for the *Service*.

Table 19 – CloseSession Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.28 for <i>RequestHeader</i> definition).
deleteSubscriptions	Boolean	If the value is TRUE, the <i>Server</i> deletes all Subscriptions associated with the <i>Session</i> . If the value is FALSE, the <i>Server</i> keeps the Subscriptions associated with the <i>Session</i> until they timeout based on their own lifetime.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> definition).

5.6.4.3 Service results

Table 20 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 20 – CloseSession Service Result Codes

Symbolic Id	Description
Bad_SessionIdInvalid	See Table 177 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 21 defines the parameters for the *Service*.

Table 21 – Cancel Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.28 for <i>RequestHeader</i> definition).
requestHandle	IntegerId	The <i>requestHandle</i> assigned to one or more requests that should be cancelled. All outstanding requests with the matching <i>requestHandle</i> shall be cancelled.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.29 for <i>ResponseHeader</i> definition).
cancelCount	UInt32	Number of cancelled requests.

5.6.5.3 Service results

Common *StatusCodes* are defined in Table 177.

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).

When a *Server* creates an instance of a *TypeDefinitionNode* it shall create the same hierarchy of *Nodes* beneath the new *Object* or *Variable* depending on the *ModellingRule* of each *InstanceDeclaration*. All *Nodes* with a *ModellingRule* of *Mandatory* shall be created or an existing *Node* shall be referenced that conforms to the *InstanceDeclaration*. The creation of *Nodes* with other *ModellingRules* is *Server* specific.

5.7.2.2 Parameters

Table 22 defines the parameters for the *Service*.

Table 22 – AddNodes Service Parameters

Name	Type	Description
Request		
requestHeader	RequestHeader	Common request parameters (see 7.28 for <i>RequestHeader</i> definition).
nodesToAdd []	AddNodesItem	List of <i>Nodes</i> to add. All <i>Nodes</i> are added as a <i>Reference</i> to an existing <i>Node</i> using a hierarchical <i>ReferenceType</i> . This structure is defined in-line with the following indented items.
parentNodeid	Expanded Nodeid	<i>ExpandedNodeid</i> of the parent <i>Node</i> for the <i>Reference</i> . The <i>ExpandedNodeid</i> type is defined in 7.11.
referenceTypeid	Nodeid	<i>Nodeid</i> of the hierarchical <i>ReferenceType</i> to use for the <i>Reference</i> from the parent <i>Node</i> to the new <i>Node</i> .
requestedNewNodeid	Expanded Nodeid	<i>Client</i> requested expanded <i>Nodeid</i> of the <i>Node</i> to add. The <i>serverIndex</i> in the expanded <i>Nodeid</i> shall be 0. If the <i>Server</i> cannot use this <i>Nodeid</i> , it rejects this <i>Node</i> and returns the appropriate error code. If the <i>Client</i> does not want to request a <i>Nodeid</i> , then it sets the value of this parameter to the null expanded <i>Nodeid</i> . If the <i>Node</i> to add is a <i>ReferenceType Node</i> , its <i>Nodeid</i> should be a numeric id. See IEC 62541-3 for a description of <i>ReferenceType Nodeids</i> .
browseName	QualifiedName	The browse name of the <i>Node</i> to add.
nodeClass	NodeClass	<i>NodeClass</i> of the <i>Node</i> to add.
nodeAttributes	Extensible Parameter NodeAttributes	The <i>Attributes</i> that are specific to the <i>NodeClass</i> . The <i>NodeAttributes</i> parameter type is an extensible parameter type specified in 7.19. A <i>Client</i> is allowed to omit values for some or all <i>Attributes</i> . If an <i>Attribute</i> value is null, the <i>Server</i> shall use the default values from the <i>TypeDefinitionNode</i> . If a <i>TypeDefinitionNode</i> was not provided the <i>Server</i> shall choose a suitable default value. The <i>Server</i> may still add an optional <i>Attribute</i> to the <i>Node</i> with an appropriate default value even if the <i>Client</i> does not specify a value.
typeDefinition	Expanded Nodeid	<i>Nodeid</i> of the <i>TypeDefinitionNode</i> for the <i>Node</i> to add. This parameter shall be null for all <i>NodeClasses</i> other than <i>Object</i> and <i>Variable</i> in which case it shall be provided.
Response		
responseHeader	Response Header	Common response parameters (see 7.29 for <i>ResponseHeader</i> definition).
results []	AddNodesResult	List of results for the <i>Nodes</i> to add. The size and order of the list matches the size and order of the <i>nodesToAdd</i> request parameter. This structure is defined in-line with the following indented items.
statusCode	StatusCode	<i>StatusCode</i> for the <i>Node</i> to add (see 7.34 for <i>StatusCode</i> definition).
addedNodeid	Nodeid	<i>Server</i> assigned <i>Nodeid</i> of the added <i>Node</i> . Null <i>Nodeid</i> if the operation failed.
diagnosticInfos []	DiagnosticInfo	List of diagnostic information for the <i>Nodes</i> to add (see 7.8 for <i>DiagnosticInfo</i> definition). The size and order of the list matches the size and order of the <i>nodesToAdd</i> request parameter. This list is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request.

5.7.2.3 Service results

Table 23 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 177.

Table 23 – AddNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 177 for the description of this result code.
Bad_TooManyOperations	See Table 177 for the description of this result code.

5.7.2.4 StatusCodes

Table 24 defines values for the operation level *statusCode* parameter that are specific to this *Service*. Common *StatusCodes* are defined in Table 178.