



DPC: 24 / 30484412 DC

BSI Group Headquarters

389 Chiswick High Road London W4 4AL

Tel: +44 (0)20 8996 9000

Fax: +44 (0)20 8996 7400

www.bsigroup.com

Date: 29 January 2024

Origin: International

Latest date for receipt of comments: 20 March 2024

Project No. 2023/02970

Responsible committee: GEL/65/5 Devices and integration in enterprise systems

Interested committees:

Title: BS EN IEC 62541-4 OPC Unified Architecture

Part 4: Services

Please notify the secretary if you are aware of any keywords that might assist in classifying or identifying the standard or if the content of this standard

- i) has any issues related to 3rd party IPR, patent or copyright
- ii) affects other national standard(s)
- iii) requires additional national guidance or information

**WARNING: THIS IS A DRAFT AND MUST NOT BE REGARDED OR USED AS A BRITISH STANDARD.
THIS DRAFT IS NOT CURRENT BEYOND 20 March 2024**

This draft is issued to allow comments from interested parties; all comments will be given consideration prior to publication. No acknowledgement will normally be sent. **See overleaf for information on the submission of comments.**

No copying is allowed, in any form, without prior written permission from BSI except as permitted under the Copyright, Designs and Patent Act 1988 or for circulation within a nominating organization for briefing purposes. Electronic circulation is limited to dissemination by e-mail within such an organization by committee members.

Further copies of this draft may be purchased from BSI Shop <http://shop.bsigroup.com> or from BSI Customer Services, Tel: +44(0) 20 8996 9001 or email cservices@bsigroup.com. British, International and foreign standards are also available from BSI Customer Services.

Information on the co-operating organizations represented on the committees referenced above may be obtained from <http://standardsdevelopment.bsigroup.com>

Responsible Committee Secretary: Ms Charmaine Mukherjee (BSI)

Direct tel:

E-mail: charmaine.mukherjee@bsigroup.com

Introduction

This draft standard is based on national and international discussions. Your comments on this draft are invited and will assist in the preparation of the consequent standard.

For international standards, comments will be reviewed by the relevant UK national committee before sending the consensus UK vote and comments to the international committee, which will then decide appropriate action. If the international standard is approved, it is usual for the text to be published as a British Standard.

For national standards, comments will be reviewed by the relevant UK national committee and the resulting standards published as a British Standard.

UK Vote

Please indicate whether you consider the UK should submit a negative (with supporting technical reasons) or positive vote on this draft. Please indicate if you are aware of any reason why this draft standard should not be published as a British Standard.

Submission of Comments

- **Annotated drafts are not acceptable and will be rejected.**
- All comments should be submitted online at <https://standardsdevelopment.bsigroup.com/>. You will need to register in order to comment.

Template for comments and secretariat observations

Date: xx/xx/20xx	Document: ISO/DIS xxxx
------------------	------------------------

1	2	(3)	4	5	(6)	7
MB	Clause No./ Subclause No./Annex (e.g. 3.1)	Paragraph/Figure/ Table/Note	Type of comment	Commend (justification for change) by the MB	Proposed change by the MB	Secretariat observations on each comment submitted
	3.1	Definition 1	ed	Definition is ambiguous and needs clarifying.	Amend to read '...so that the mains connector to which no connection...'	
	6.4	Paragraph 2	te	The use of the UV photometer as an alternative cannot be supported as serious problems have been encountered in its use in the UK.	Delete reference to UV photometer.	



COMMITTEE DRAFT FOR VOTE (CDV)

PROJECT NUMBER: IEC 62541-4 ED4	
DATE OF CIRCULATION: 2024-01-26	CLOSING DATE FOR VOTING: 2024-04-19
SUPERSEDES DOCUMENTS: 65E/976/RR	

IEC SC 65E : DEVICES AND INTEGRATION IN ENTERPRISE SYSTEMS	
SECRETARIAT: United States of America	SECRETARY: Mr Donald (Bob) Lattimer
OF INTEREST TO THE FOLLOWING COMMITTEES:	PROPOSED HORIZONTAL STANDARD: ☐ Other TC/SCs are requested to indicate their interest, if any, in this CDV to the secretary.
FUNCTIONS CONCERNED: ☐ EMC ☐ ENVIRONMENT ☐ QUALITY ASSURANCE ☐ SAFETY	
☒ SUBMITTED FOR CENELEC PARALLEL VOTING Attention IEC-CENELEC parallel voting The attention of IEC National Committees, members of CENELEC, is drawn to the fact that this Committee Draft for Vote (CDV) is submitted for parallel voting. The CENELEC members are invited to vote through the CENELEC online voting system.	☐ NOT SUBMITTED FOR CENELEC PARALLEL VOTING

This document is still under study and subject to change. It should not be used for reference purposes.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Recipients of this document are invited to submit, with their comments, notification of any relevant "In Some Countries" clauses to be included should this proposal proceed. Recipients are reminded that the CDV stage is the final stage for submitting ISC clauses. (SEE [AC/22/2007](#) OR [NEW GUIDANCE DOC](#)).

TITLE:

OPC Unified Architecture - Part 4: Services

PROPOSED STABILITY DATE: 2026

NOTE FROM TC/SC OFFICERS:

Copyright © 2024 International Electrotechnical Commission, IEC. All rights reserved. It is permitted to download this electronic file, to make a copy and to print out the content for the sole purpose of preparing National Committee positions. You may not copy or "mirror" the file or printed version of the document, or any part of it, for any other purpose without permission in writing from IEC.

CONTENTS

3	FIGURES	v
4	TABLES	vi
5	FOREWORD	xiii
6	1 Scope	1
7	2 Normative references	1
8	3 Terms, definitions, abbreviated terms and conventions	1
9	3.1 Terms and definitions	1
10	3.2 Abbreviated terms	3
11	3.3 Conventions for Service definitions	3
12	4 Overview	4
13	4.1 Service Set model	4
14	4.2 Request/response Service procedures	7
15	5 Service Sets	8
16	5.1 General	8
17	5.2 Service request and response header	8
18	5.3 Service results	8
19	5.4 Discovery Service Set	9
20	5.4.1 Overview	9
21	5.4.2 FindServers	11
22	5.4.3 FindServersOnNetwork	12
23	5.4.4 GetEndpoints	13
24	5.4.5 RegisterServer	15
25	5.4.6 RegisterServer2	17
26	5.5 SecureChannel Service Set	18
27	5.5.1 Overview	18
28	5.5.2 OpenSecureChannel	20
29	5.5.3 CloseSecureChannel	22
30	5.6 Session Service Set	23
31	5.6.1 Overview	23
32	5.6.2 CreateSession	23
33	5.6.3 ActivateSession	27
34	5.6.4 CloseSession	30
35	5.6.5 Cancel	31
36	5.7 NodeManagement Service Set	31
37	5.7.1 Overview	31
38	5.7.2 AddNodes	31
39	5.7.3 AddReferences	33
40	5.7.4 DeleteNodes	35
41	5.7.5 DeleteReferences	36
42	5.8 View Service Set	37
43	5.8.1 Overview	37
44	5.8.2 Browse	37
45	5.8.3 BrowseNext	39
46	5.8.4 TranslateBrowsePathsToNodeIds	41
47	5.8.5 RegisterNodes	43

48	5.8.6	UnregisterNodes	44
49	5.9	Query Service Set	44
50	5.9.1	Overview	44
51	5.9.2	Querying Views	45
52	5.9.3	QueryFirst	45
53	5.9.4	QueryNext	49
54	5.10	Attribute Service Set	50
55	5.10.1	Overview	50
56	5.10.2	Read	50
57	5.10.3	HistoryRead	52
58	5.10.4	Write	55
59	5.10.5	HistoryUpdate	57
60	5.11	Method Service Set	58
61	5.11.1	Overview	58
62	5.11.2	Call	59
63	5.12	MonitoredItem Service Set	62
64	5.12.1	MonitoredItem model	62
65	5.12.2	CreateMonitoredItems	67
66	5.12.3	ModifyMonitoredItems	69
67	5.12.4	SetMonitoringMode	71
68	5.12.5	SetTriggering	72
69	5.12.6	DeleteMonitoredItems	73
70	5.13	Subscription Service Set	74
71	5.13.1	Subscription model	74
72	5.13.2	CreateSubscription	80
73	5.13.3	ModifySubscription	82
74	5.13.4	SetPublishingMode	83
75	5.13.5	Publish	85
76	5.13.6	Republish	87
77	5.13.7	TransferSubscriptions	87
78	5.13.8	DeleteSubscriptions	89
79	6	Service behaviours	90
80	6.1	Security	90
81	6.1.1	Overview	90
82	6.1.2	Obtaining and installing an Application Instance Certificate	90
83	6.1.3	Determining if a Certificate is trusted	91
84	6.1.4	Creating a SecureChannel	95
85	6.1.5	Creating a Session	97
86	6.1.6	Impersonating a User	97
87	6.1.7	Continuous security checks	98
88	6.2	Authorization Services	98
89	6.2.1	Overview	98
90	6.2.2	Indirect handshake with an Identity Provider	98
91	6.2.3	Direct handshake with an Identity Provider	99
92	6.3	Session-less Service invocation	100
93	6.3.1	Description	100
94	6.3.2	Parameters	101
95	6.3.3	Service results	102
96	6.4	Software Certificates	102

97	6.5	Auditing.....	102
98	6.5.1	Overview.....	102
99	6.5.2	General audit logs.....	102
100	6.5.3	General audit Events.....	102
101	6.5.4	Auditing for Discovery Service Set.....	102
102	6.5.5	Auditing for SecureChannel Service Set.....	102
103	6.5.6	Auditing for Session Service Set.....	103
104	6.5.7	Auditing for NodeManagement Service Set.....	103
105	6.5.8	Auditing for Attribute Service Set.....	103
106	6.5.9	Auditing for Method Service Set.....	104
107	6.5.10	Auditing for View, Query, MonitoredItem and Subscription Service Set.....	104
108	6.6	Redundancy.....	104
109	6.6.1	Redundancy overview.....	104
110	6.6.2	Server Redundancy.....	105
111	6.6.3	Client Redundancy.....	115
112	6.6.4	Network Redundancy.....	115
113	6.6.5	Manually Forcing Failover.....	116
114	6.7	Re-establishing connections.....	117
115	6.8	Durable Subscriptions.....	118
116	7	Common parameter type definitions.....	119
117	7.1	AdditionalParametersType.....	119
118	7.2	ApplicationDescription.....	120
119	7.3	ApplicationInstanceCertificate.....	120
120	7.4	ApplicationType.....	121
121	7.5	BrowseDirection.....	121
122	7.6	BrowseResult.....	121
123	7.7	ContentFilter.....	122
124	7.7.1	ContentFilter structure.....	122
125	7.7.2	ContentFilterResult.....	122
126	7.7.3	FilterOperator.....	123
127	7.7.4	FilterOperand parameters.....	130
128	7.8	Counter.....	131
129	7.9	ContinuationPoint.....	132
130	7.10	DataChangeTrigger.....	132
131	7.11	DataValue.....	132
132	7.11.1	General.....	132
133	7.11.2	PicoSeconds.....	133
134	7.11.3	SourceTimestamp.....	133
135	7.11.4	ServerTimestamp.....	133
136	7.11.5	StatusCode assigned to a value.....	134
137	7.12	DiagnosticInfo.....	134
138	7.13	DiscoveryConfiguration parameters.....	135
139	7.13.1	Overview.....	135
140	7.13.2	MdnsDiscoveryConfiguration.....	135
141	7.14	EndpointDescription.....	136
142	7.15	EphemeralKeyType.....	136
143	7.16	ExpandedNodeId.....	137
144	7.17	ExtensibleParameter.....	137
145	7.18	Index.....	137

146	7.19	IntegerId	137
147	7.20	MessageSecurityMode.....	138
148	7.21	MonitoringParameters.....	138
149	7.22	MonitoringFilter parameters	139
150	7.22.1	Overview.....	139
151	7.22.2	DataChangeFilter	140
152	7.22.3	EventFilter	140
153	7.22.4	AggregateFilter	142
154	7.23	MonitoringMode.....	143
155	7.24	NodeAttributes parameters	144
156	7.24.1	Overview.....	144
157	7.24.2	ObjectAttributes parameter	145
158	7.24.3	VariableAttributes parameter	145
159	7.24.4	MethodAttributes parameter.....	146
160	7.24.5	ObjectTypeAttributes parameter.....	146
161	7.24.6	VariableTypeAttributes parameter.....	146
162	7.24.7	ReferenceTypeAttributes parameter.....	147
163	7.24.8	DataTypeAttributes parameter	147
164	7.24.9	ViewAttributes parameter.....	147
165	7.24.10	GenericAttributes parameter	148
166	7.25	NotificationData parameters.....	148
167	7.25.1	Overview.....	148
168	7.25.2	DataChangeNotification parameter	148
169	7.25.3	EventNotificationList parameter	149
170	7.25.4	StatusChangeNotification parameter	149
171	7.26	NotificationMessage	150
172	7.27	NumericRange	150
173	7.28	QueryDataSet.....	151
174	7.29	ReadValueId	151
175	7.30	ReferenceDescription	152
176	7.31	RelativePath.....	153
177	7.32	RegisteredServer.....	154
178	7.33	RequestHeader	154
179	7.34	ResponseHeader.....	157
180	7.35	ServiceFault	157
181	7.36	SessionAuthenticationToken	157
182	7.37	SignatureData	158
183	7.38	SignedSoftwareCertificate	159
184	7.39	StatusCode	159
185	7.39.1	General.....	159
186	7.39.2	Common StatusCodes	161
187	7.40	TimestampsToReturn	163
188	7.41	UserIdentityToken parameters	164
189	7.41.1	Overview.....	164
190	7.41.2	Token Encryption and Proof of Possession	164
191	7.41.3	AnonymousIdentityToken.....	169
192	7.41.4	UserNameIdentityToken	169
193	7.41.5	X509IdentityTokens.....	170
194	7.41.6	IssuedIdentityToken	171

195	7.42	UserTokenPolicy	171
196	7.43	UserTokenType	172
197	7.44	VersionTime	172
198	7.45	ViewDescription	172
199	Annex A (informative)	BNF definitions	174
200	A.1	Overview over BNF	174
201	A.2	BNF of RelativePath	174
202	A.3	BNF of NumericRange	175
203	Annex B (informative)	ContentFilter and Query examples	176
204	B.1	Simple ContentFilter examples	176
205	B.1.1	Overview	176
206	B.1.2	Example 1	176
207	B.1.3	Example 2	177
208	B.2	Complex Examples of Query Filters	177
209	B.2.1	Overview	177
210	B.2.2	Used type model	178
211	B.2.3	Example Notes	180
212	B.2.4	Example 1	181
213	B.2.5	Example 2	182
214	B.2.6	Example 3	183
215	B.2.7	Example 4	185
216	B.2.8	Example 5	186
217	B.2.9	Example 6	188
218	B.2.10	Example 7	189
219	B.2.11	Example 8	191
220	B.2.12	Example 9	192

FIGURES

224	Figure 1 – Discovery Service Set	4
225	Figure 2 – SecureChannel Service Set	5
226	Figure 3 – Session Service Set	5
227	Figure 4 – NodeManagement Service Set	5
228	Figure 5 – View Service Set	6
229	Figure 6 – Attribute Service Set	6
230	Figure 7 – Method Service Set	7
231	Figure 8 – MonitoredItem and Subscription Service Sets	7
232	Figure 9 – Discovery process	10
233	Figure 10 – Using a Gateway Server	15
234	Figure 11 – The Registration Process – Manually Launched Servers	16
235	Figure 12 – The Registration Process – Automatically Launched Servers	16
236	Figure 13 – SecureChannel and Session Services	19
237	Figure 14 – Multiplexing users on a Session	25
238	Figure 15 – MonitoredItem model	62
239	Figure 16 – Typical delay in change detection	64
240	Figure 17 – Queue overflow handling	65

241	Figure 18 – Triggering model.....	66
242	Figure 19 – Obtaining and installing an Application Instance Certificate.....	91
243	Figure 20 – Determining if an Application Instance Certificate is trusted	95
244	Figure 21 – Establishing a SecureChannel.....	96
245	Figure 22 – Establishing a Session	97
246	Figure 23 – Impersonating a User.....	98
247	Figure 24 – Indirect handshake with an Identity Provider.....	99
248	Figure 25 – Direct handshake with an Identity Provider	100
249	Figure 26 – Transparent Redundancy setup example	106
250	Figure 27 – Non-Transparent Redundancy setup	107
251	Figure 28 – Client Start-up steps	110
252	Figure 29 – Cold Failover	111
253	Figure 30 – Warm Failover	112
254	Figure 31 – Hot Failover	113
255	Figure 32 – HotAndMirrored Failover	114
256	Figure 33 – Server proxy for Redundancy	114
257	Figure 34 – Transparent network Redundancy	115
258	Figure 35 – Non-transparent network Redundancy.....	116
259	Figure 36 – Reconnect sequence	117
260	Figure 37 – Logical layers of a <i>Server</i>	158
261	Figure 38 – Obtaining a SessionAuthenticationToken.....	158
262	Figure 39 – EncryptedSecret layout	166
263	Figure B.1 – Filter logic tree example	176
264	Figure B.2 – Filter logic tree example	177
265	Figure B.3 – Example Type Nodes.....	179
266	Figure B.4 – Example Instance Nodes	180
267	Figure B.5 – Example 1 Filter	181
268	Figure B.6 – Example 2 filter logic tree	183
269	Figure B.7 – Example 3 filter logic tree	184
270	Figure B.8 – Example 4 filter logic tree	186
271	Figure B.9 – Example 5 filter logic tree	187
272	Figure B.10 – Example 6 filter logic tree	188
273	Figure B.11 – Example 7 filter logic tree	190
274	Figure B.12 – Example 8 filter logic tree	191
275	Figure B.13 – Example 9 filter logic tree	193

TABLES

279	Table 1 – Service definition table.....	3
280	Table 2 – Parameter Types defined in OPC 10000-3	4
281	Table 3 – FindServers Service Parameters	12
282	Table 4 – FindServersOnNetwork Service Parameters	13
283	Table 5 – GetEndpoints Service Parameters	15

284	Table 6 – RegisterServer Service Parameters	17
285	Table 7 – RegisterServer Service Result Codes	17
286	Table 8 – RegisterServer2	18
287	Table 9 – RegisterServer2 Service Result Codes	18
288	Table 10 – RegisterServer2 Operation Level Result Codes	18
289	Table 11 – OpenSecureChannel Service Parameters	21
290	Table 12 – OpenSecureChannel Service Result Codes	22
291	Table 13 – CloseSecureChannel Service Parameters	23
292	Table 14 – CloseSecureChannel Service Result Codes	23
293	Table 15 – CreateSession Service Parameters	26
294	Table 16 – CreateSession Service Result Codes	27
295	Table 17 – ActivateSession Service Parameters	29
296	Table 18 – ActivateSession Service Result Codes	30
297	Table 19 – CloseSession Service Parameters	30
298	Table 20 – CloseSession Service Result Codes	30
299	Table 21 – Cancel Service Parameters	31
300	Table 22 – AddNodes Service Parameters	32
301	Table 23 – AddNodes Service Result Codes	32
302	Table 24 – AddNodes Operation Level Result Codes	33
303	Table 25 – AddReferences Service Parameters	34
304	Table 26 – AddReferences Service Result Codes	34
305	Table 27 – AddReferences Operation Level Result Codes	34
306	Table 28 – DeleteNodes Service Parameters	35
307	Table 29 – DeleteNodes Service Result Codes	35
308	Table 30 – DeleteNodes Operation Level Result Codes	36
309	Table 31 – DeleteReferences Service Parameters	36
310	Table 32 – DeleteReferences Service Result Codes	36
311	Table 33 – DeleteReferences Operation Level Result Codes	37
312	Table 34 – Browse Service Parameters	38
313	Table 35 – Browse Service Result Codes	39
314	Table 36 – Browse Operation Level Result Codes	39
315	Table 37 – BrowseNext Service Parameters	40
316	Table 38 – BrowseNext Service Result Codes	40
317	Table 39 – BrowseNext Operation Level Result Codes	41
318	Table 40 – TranslateBrowsePathsToNodeIds Service Parameters	42
319	Table 41 – TranslateBrowsePathsToNodeIds Service Result Codes	42
320	Table 42 – TranslateBrowsePathsToNodeIds Operation Level Result Codes	43
321	Table 43 – RegisterNodes Service Parameters	43
322	Table 44 – RegisterNodes Service Result Codes	44
323	Table 45 – UnregisterNodes Service Parameters	44
324	Table 46 – UnregisterNodes Service Result Codes	44
325	Table 47 – QueryFirst Request Parameters	47
326	Table 48 – QueryFirst Response Parameters	48

327	Table 49 – QueryFirst Service Result Codes	49
328	Table 50 – QueryFirst Operation Level Result Codes	49
329	Table 51 – QueryNext Service Parameters	50
330	Table 52 – QueryNext Service Result Codes.....	50
331	Table 53 – Read Service Parameters.....	51
332	Table 54 – Read Service Result Codes.....	51
333	Table 55 – Read Operation Level Result Codes	52
334	Table 56 – HistoryRead Service Parameters	53
335	Table 57 – HistoryRead Service Result Codes	55
336	Table 58 – HistoryRead Operation Level Result Codes	55
337	Table 59 – Write Service Parameters.....	56
338	Table 60 – Write Service Result Codes.....	57
339	Table 61 – Write Operation Level Result Codes	57
340	Table 62 – HistoryUpdate Service Parameters	58
341	Table 63 – HistoryUpdate Service Result Codes	58
342	Table 64 – HistoryUpdate Operation Level Result Codes	58
343	Table 65 – Call Service Parameters.....	60
344	Table 66 – Call Service Result Codes.....	61
345	Table 67 – Call Operation Level Result Codes	61
346	Table 68 – Call Input Argument Result Codes.....	61
347	Table 69 – CreateMonitoredItems Service Parameters	68
348	Table 70 – CreateMonitoredItems Service Result Codes	68
349	Table 71 – CreateMonitoredItems Operation Level Result Codes	69
350	Table 72 – ModifyMonitoredItems Service Parameters	70
351	Table 73 – ModifyMonitoredItems Service Result Codes	70
352	Table 74 – ModifyMonitoredItems Operation Level Result Codes.....	71
353	Table 75 – SetMonitoringMode Service Parameters	71
354	Table 76 – SetMonitoringMode Service Result Codes	71
355	Table 77 – SetMonitoringMode Operation Level Result Codes.....	71
356	Table 78 – SetTriggering Service Parameters	72
357	Table 79 – SetTriggering Service Result Codes	72
358	Table 80 – SetTriggering Operation Level Result Codes	73
359	Table 81 – DeleteMonitoredItems Service Parameters	73
360	Table 82 – DeleteMonitoredItems Service Result Codes	73
361	Table 83 – DeleteMonitoredItems Operation Level Result Codes.....	73
362	Table 84 – Subscription States	76
363	Table 85 – Subscription State Table	77
364	Table 86 – State variables and parameters	79
365	Table 87 – Functions.....	80
366	Table 88 – CreateSubscription Service Parameters.....	81
367	Table 89 – CreateSubscription Service Result Codes.....	82
368	Table 90 – ModifySubscription Service Parameters.....	83
369	Table 91 – ModifySubscription Service Result Codes	83

370	Table 92 – SetPublishingMode Service Parameters	84
371	Table 93 – SetPublishingMode Service Result Codes	84
372	Table 94 – SetPublishingMode Operation Level Result Codes	85
373	Table 95 – Publish Service Parameters	86
374	Table 96 – Publish Service Result Codes.....	86
375	Table 97 – Publish Operation Level Result Codes.....	87
376	Table 98 – Republish Service Parameters	87
377	Table 99 – Republish Service Result Codes.....	87
378	Table 100 – TransferSubscriptions Service Parameters.....	88
379	Table 101 – TransferSubscriptions Service Result Codes.....	88
380	Table 102 – TransferSubscriptions Operation Level Result Codes	89
381	Table 103 – DeleteSubscriptions Service Parameters	89
382	Table 104 – DeleteSubscriptions Service Result Codes	89
383	Table 105 – DeleteSubscriptions Operation Level Result Codes.....	89
384	Table 106 – Certificate validation steps	93
385	Table 107 – SessionlessInvoke Service Parameters.....	101
386	Table 108 – SessionlessInvoke Service Result Codes.....	102
387	Table 109 – ServiceLevel ranges.....	108
388	Table 110 – Server Failover modes	109
389	Table 111 – Redundancy Failover actions.....	110
390	Table 112 – AdditionalParametersType.....	120
391	Table 113 – ApplicationDescription.....	120
392	Table 114 – ApplicationInstanceCertificate	121
393	Table 115 – ApplicationType values	121
394	Table 116 – BrowseDirection values	121
395	Table 117 – BrowseResult.....	122
396	Table 118 – ContentFilter structure.....	122
397	Table 119 – ContentFilterResult structure	123
398	Table 120 – ContentFilterResult Result Codes	123
399	Table 121 – ContentFilterResult Operand Result Codes.....	123
400	Table 122 – Basic FilterOperator definition	124
401	Table 123 – Complex FilterOperator definition	126
402	Table 124 – Wildcard characters	127
403	Table 125 – Conversion rules.....	128
404	Table 126 – Data Precedence rules	129
405	Table 127 – Logical AND Truth table	129
406	Table 128 – Logical OR Truth table	130
407	Table 129 – FilterOperand parameter Typelds	130
408	Table 130 – ElementOperand.....	130
409	Table 131 – LiteralOperand	130
410	Table 132 – AttributeOperand.....	131
411	Table 133 – SimpleAttributeOperand	131
412	Table 134 – DataChangeTrigger values	132

413	Table 135 – DataValue	133
414	Table 136 – DiagnosticInfo	135
415	Table 137 – DiscoveryConfiguration parameterTypeIds	135
416	Table 138 – MdnsDiscoveryConfiguration	136
417	Table 139 – EndpointDescription	136
418	Table 140 – EphemeralKeyType	137
419	Table 141 – ExpandedNodeId	137
420	Table 142 – ExtensibleParameter base type	137
421	Table 143 – MessageSecurityMode values	138
422	Table 144 – MonitoringParameters	139
423	Table 145 – MonitoringFilter parameterTypeIds	140
424	Table 146 – DataChangeFilter	140
425	Table 147 – EventFilter structure	142
426	Table 148 – EventFilterResult structure	142
427	Table 149 – EventFilterResult Result Codes	142
428	Table 150 – AggregateFilter structure	143
429	Table 151 – AggregateFilterResult structure	143
430	Table 152 – MonitoringMode values	144
431	Table 153 – NodeAttributes parameterTypeIds	144
432	Table 154 – Bit mask for specified Attributes	145
433	Table 155 – ObjectAttributes	145
434	Table 156 – VariableAttributes	146
435	Table 157 – MethodAttributes	146
436	Table 158 – ObjectTypeAttributes	146
437	Table 159 – VariableTypeAttributes	147
438	Table 160 – ReferenceTypeAttributes	147
439	Table 161 – DataTypeAttributes	147
440	Table 162 – ViewAttributes	148
441	Table 163 – GenericAttributes	148
442	Table 164 – NotificationData parameterTypeIds	148
443	Table 165 – DataChangeNotification	149
444	Table 166 – EventNotificationList	149
445	Table 167 – StatusChangeNotification	150
446	Table 168 – NotificationMessage	150
447	Table 169 – NumericRange	151
448	Table 170 – QueryDataSet	151
449	Table 171 – ReadValueId	152
450	Table 172 – ReferenceDescription	152
451	Table 173 – RelativePath	153
452	Table 174 – RegisteredServer	154
453	Table 175 – RequestHeader	155
454	Table 176 – ResponseHeader	157
455	Table 177 – ServiceFault	157

456	Table 178 – SignatureData	159
457	Table 179 – SignedSoftwareCertificate	159
458	Table 180 – StatusCode bit assignments	160
459	Table 181 – DataValue InfoBits	161
460	Table 182 – Common Service Result Codes	161
461	Table 183 – Common Operation Level Result Codes	162
462	Table 184 – TimestampsToReturn values	164
463	Table 185 – UserIdentityToken parameterTypeIds	164
464	Table 186 – Legacy UserIdentityToken Encrypted Token Secret Format	165
465	Table 187 – EncryptedSecret layout	167
466	Table 188 – EncryptedSecret DataTypes	167
467	Table 189 – RsaEncryptedSecret structure	168
468	Table 190 – EccEncryptedSecret Layout	169
469	Table 191 – AnonymousIdentityToken	169
470	Table 192 – UserNameIdentityToken	170
471	Table 193 – EncryptionAlgorithm selection	170
472	Table 194 – X.509 v3 Identity Token	171
473	Table 195 – IssuedIdentityToken	171
474	Table 196 – UserTokenPolicy	172
475	Table 197 – UserTokenType values	172
476	Table 198 – ViewDescription	173
477	Table A.1 – RelativePath	174
478	Table A.2 – <i>RelativePath</i> Examples	175
479	Table B.1 – ContentFilter example	177
480	Table B.2 – ContentFilter example	177
481	Table B.3 – Example 1 NodeTypeDescription	181
482	Table B.4 – Example 1 ContentFilter	181
483	Table B.5 – Example 1 QueryDataSets	182
484	Table B.6 – Example 2 NodeTypeDescription	182
485	Table B.7 – Example 2 ContentFilter	183
486	Table B.8 – Example 2 QueryDataSets	183
487	Table B.9 – Example 3 - NodeTypeDescription	184
488	Table B.10 – Example 3 ContentFilter	185
489	Table B.11 – Example 3 QueryDataSets	185
490	Table B.12 – Example 4 NodeTypeDescription	186
491	Table B.13 – Example 4 ContentFilter	186
492	Table B.14 – Example 4 QueryDataSets	186
493	Table B.15 – Example 5 NodeTypeDescription	187
494	Table B.16 – Example 5 ContentFilter	187
495	Table B.17 – Example 5 QueryDataSets	187
496	Table B.18 – Example 6 NodeTypeDescription	188
497	Table B.19 – Example 6 ContentFilter	188
498	Table B.20 – Example 6 QueryDataSets	189

499	Table B.21 – Example 6 QueryDataSets without additional information.....	189
500	Table B.22 – Example 7 NodeTypeDescription.....	190
501	Table B.23 – Example 7 ContentFilter.....	190
502	Table B.24 – Example 7 QueryDataSets	190
503	Table B.25 – Example 8 NodeTypeDescription.....	191
504	Table B.26 – Example 8 ContentFilter.....	192
505	Table B.27 – Example 8 QueryDataSets	192
506	Table B.28 – Example 9 NodeTypeDescription.....	192
507	Table B.29 – Example 9 ContentFilter.....	193
508	Table B.30 – Example 9 QueryDataSets	193
509		
510		

INTERNATIONAL ELECTROTECHNICAL COMMISSION

OPC UNIFIED ARCHITECTURE –

Part 4: Services

FOREWORD

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

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

This fourth edition cancels and replaces the third edition published in 2020. This edition constitutes a technical revision.

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

- a) Added additional definitions to Method Call Service to allow optional Method arguments.
- b) Added reference to SystemStatusChangeEvent for event monitored item error scenarios.
- c) Enhanced general description of how determining if a Certificate is trusted.
- d) Added support for ECC.
- e) Added revisedAggregateConfiguration to AggregateFilterResult structure.
- f) Added INVALID to the BrowseDirection enumeration data type.
- g) Added INVALID to the TimestampsToReturn enumeration data type.
- h) Added definitions that make sure the subscription functionality works if retransmission queues are optional.
- i) Added client checks to be symmetric to the Server Certificate check.

- j) Added clarification that 'local' top level domain is not appended by server into certificate and not checked by client when returned from LDS-ME.
- k) Added definition for expiration behaviour of IssuedIdentityTokens.
- l) Added status code Good_PasswordChangeRequired to ActivateSession.
- m) Restrict AdditionalInfo to servers in debug mode.
- n) Add new status code Bad_ServerTooBusy.
- o) Added definition for cases where server certificate must be contained in GetEndpoints response

The text of this International Standard is based on the following documents:

CDV	Report on voting
65E/XX/CDV	65E/XX/RVC

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

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

Throughout this document and the other parts of the IEC 62541 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 IEC 62541 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, with a few exceptions, 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 document will remain unchanged until the stability date indicated on the IEC website under "<http://webstore.iec.ch>" in the data related to the specific document. At this date, the document will be

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

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

OPC Unified Architecture Specification

Part 4: Services

1 Scope

This part of IEC 62541 defines the OPC Unified Architecture (OPC UA) *Services*. The *Services* defined 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 document. IEC 62541-6 specifies one or more concrete mappings supported for implementation. For example, one mapping in IEC 62541-6 is to UA-TCP UA-SC UA-Binary. In that case the *Services* described in this document appear as OPC UA Binary encoded payload, secured with OPC UA Secure Conversation and transported via OPC UA TCP.

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

2 Normative references

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

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

IEC 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*

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

IEC 62541-18: *OPC Unified Architecture – Part 18: Role-Based Security*

3 Terms, definitions, abbreviated terms and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 62541-1, IEC 62541-2, IEC 62541-3, and 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 document 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*. Each *Endpoint* includes a *HostName*.

3.1.5

EphemeralKey

public-private key pair generated for each execution of a key establishment process.

Note 1 to entry: EphemeralKeys are necessary when using ECC based SecurityPolicies.

3.1.6

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.

3.1.7

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). A *Client* failover is the act of an alternate *Client* replacing an existing failed or degraded *Client* connection to a *Server*.

3.1.8

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.

3.1.9

HostName

unique identifier for a machine on a network

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

3.1.10

Redundancy

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

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

3.1.11

RedundantServerSet

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

Note 1 to entry: A *RedundantServerSet* is a group of *Servers* that are configured to provide *Redundancy*. These *Servers* have requirements related to the address space and provide *Failovers*.

3.1.12**SecurityToken**

identifier for a cryptographic key set

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

3.1.13**ServerUri**

ApplicationUri for a *Server*

3.1.14**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.

3.2 Abbreviated terms

API	Application Programming Interface
BNF	Backus-Naur Form
CA	Certificate Authority
CRL	Certificate Revocation List
CTL	Certificate Trust List
DA	Data Access
ECC	Elliptic Curve Cryptography
GDS	Global Discovery Server
JWT	JSON Web Token
NAT	Network Address Translation
RSA	Rivest, Shamir and Adleman [Public Key Encryption System]
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 organized 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.

These *Services* are organized into *Service Sets*. Each *Service Set* defines a set of related *Services*. The organization in *Service Sets* is a logical grouping used in this document 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*.

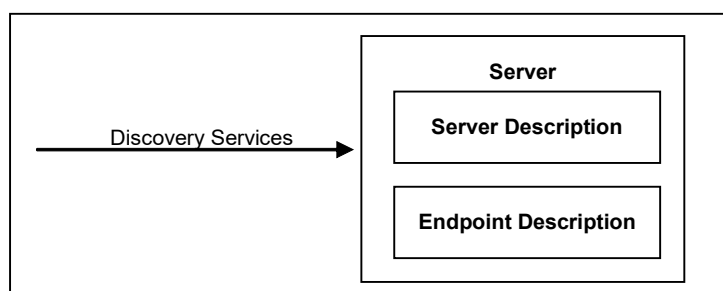


Figure 1 – Discovery Service Set

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*.

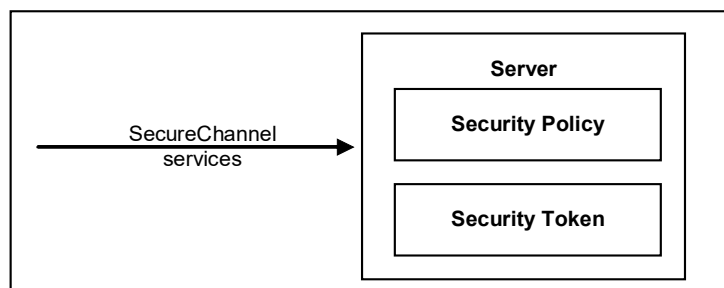


Figure 2 – SecureChannel Service Set

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*.

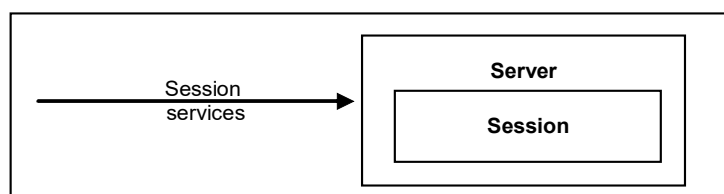


Figure 3 – Session Service Set

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

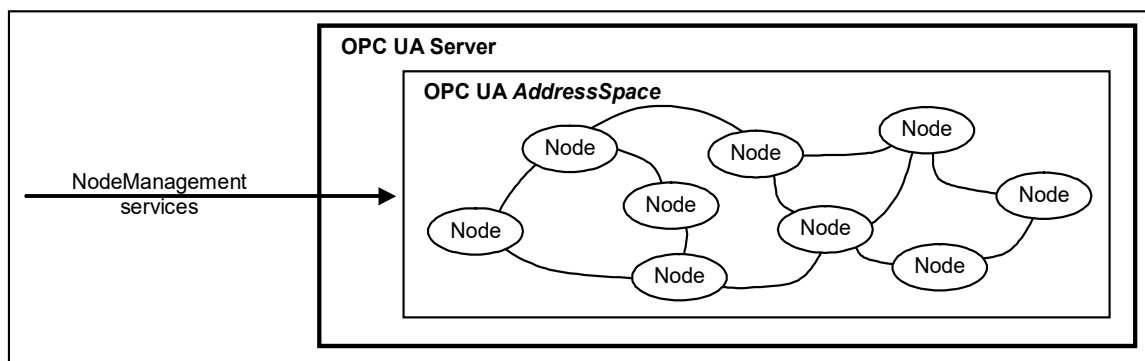


Figure 4 – NodeManagement Service Set

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*.

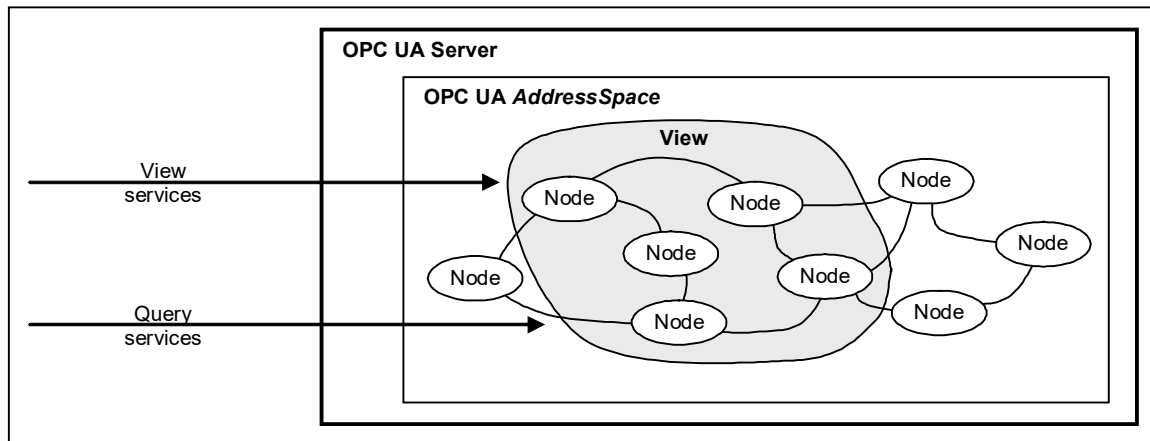


Figure 5 – View Service Set

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*.

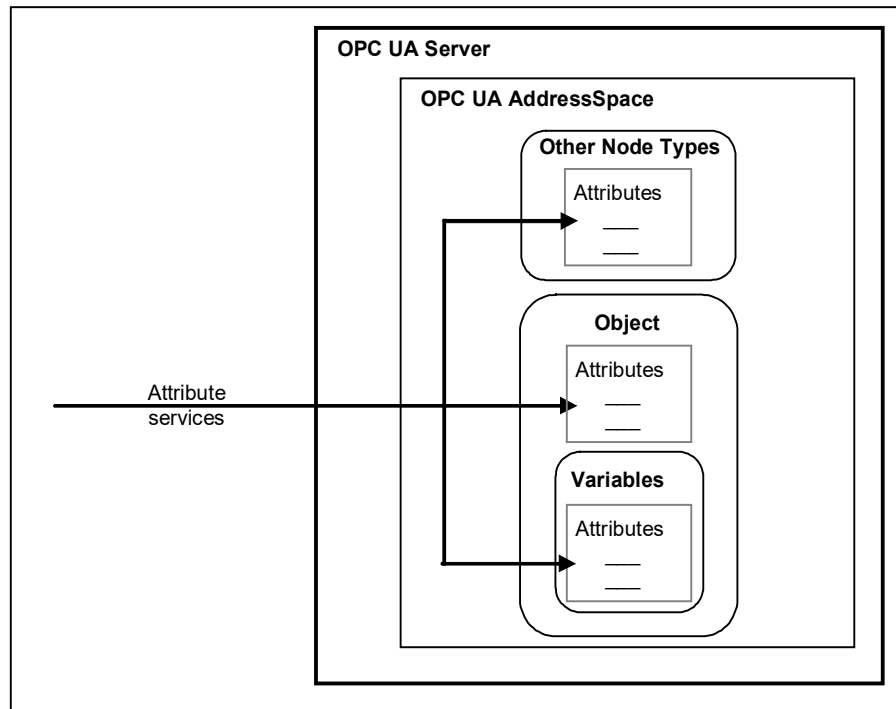


Figure 6 – Attribute Service Set

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.

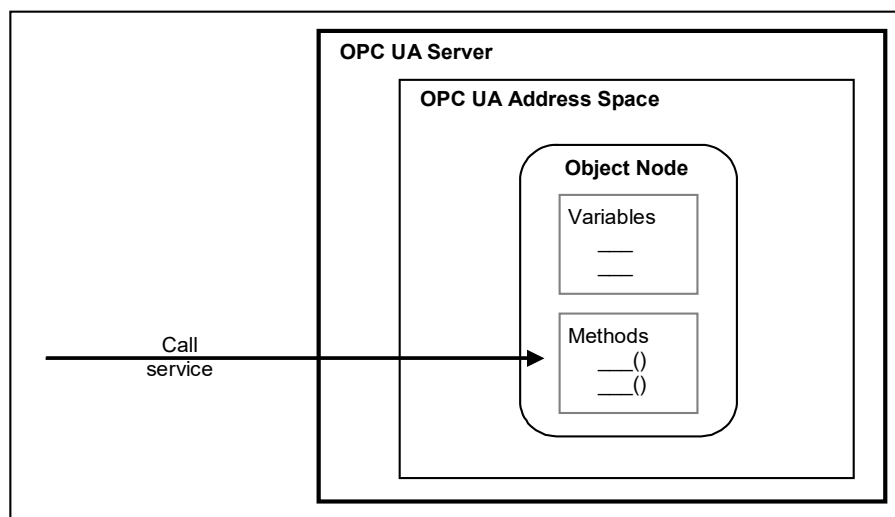


Figure 7 – Method Service Set

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.

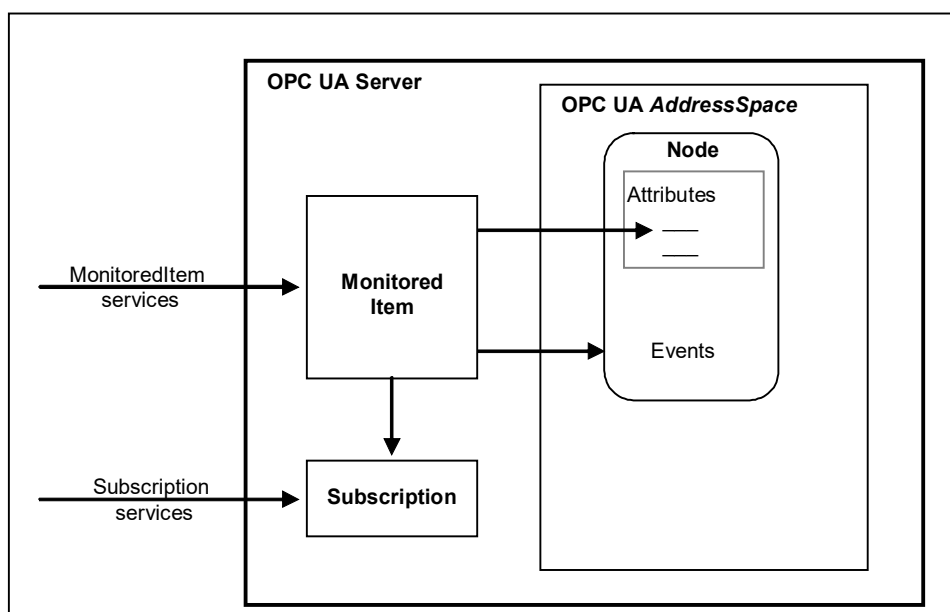


Figure 8 – MonitoredItem and Subscription Service Sets

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*. Subclause 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 7.33 and contains common request parameters such as *authenticationToken*, *timestamp* and *requestHandle*.

The *ResponseHeader* structure is defined in 7.34 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 7.39).

The status of the *Service* call is represented by the *serviceResult* contained in the *ResponseHeader* (see 7.34). 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.

- p) A bad code is returned in *serviceResult* if the *Service* itself failed. In this case, a *ServiceFault* shall be returned instead of the *Service* response message. The *ServiceFault* is defined in 7.35.
- q) 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 182 and Table 183. 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 need to reformulate their requests. The administrator may need to increase the limits for the *Client* if it receives a response from the *Server* with this fault.

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

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

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

5.4 Discovery Service Set

5.4.1 Overview

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

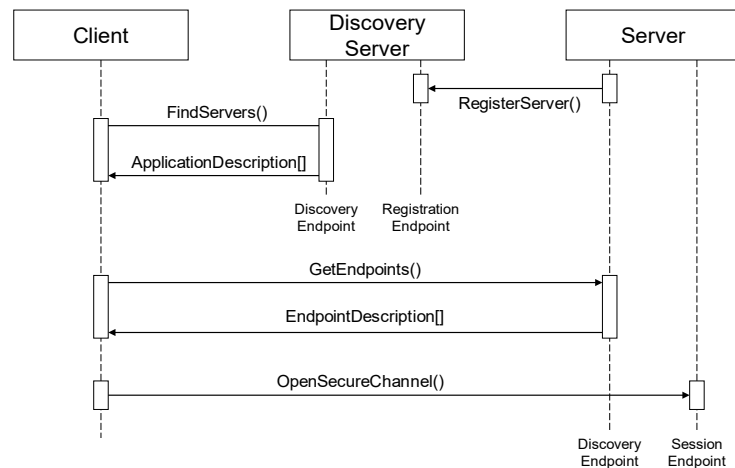


Figure 9 – Discovery process

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 *DiscoveryEndpoint*. If *GetEndpoints* is disabled and the *Server Certificate* is updated either automatically with *Certificate Manager* or manually, *Clients* will no longer be able to connect to the *Server* without manual re-configuration of the *Client*.

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:

- r) The *ApplicationUri* specified in the *Server Certificate* is the same as the *ApplicationUri* provided in the *EndpointDescription*.
- s) The *Server Certificate* returned in *CreateSession* response is the same as the *Certificate* used to create the *SecureChannel*.
- t) The *EndpointDescriptions* returned from the *DiscoveryEndpoint* are the same as the *EndpointDescriptions* returned in the *CreateSession* response, but they may be in a different order. For the content, the fields *ApplicationUri*, *EndpointUrl*, *SecurityMode*, *SecurityPolicyUri*, *UserIdentityTokens*, *TransportProfileUri* and *SecurityLevel* shall be compared for exact match. All other fields are ignored for the comparison.

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 *RedundantServerSet*.

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 7.2).

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 7.2). 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 7.33.
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 7.2 using one of the 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.
serverUris []	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 7.2.
Response		
responseHeader	ResponseHeader	Common response parameters. The <i>ResponseHeader</i> type is defined in 7.34.
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 7.2.

5.4.2.3 Service results

Common *StatusCodes* are defined in Table 182.

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 7.33.
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 7.34.
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 182.

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*. If the *securityPolicyUri* is not *None* or one of the *UserTokenPolicies* requires encryption, the *Server* shall include the *ApplicationInstanceCertificate* in the *EndpointDescription*.

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.

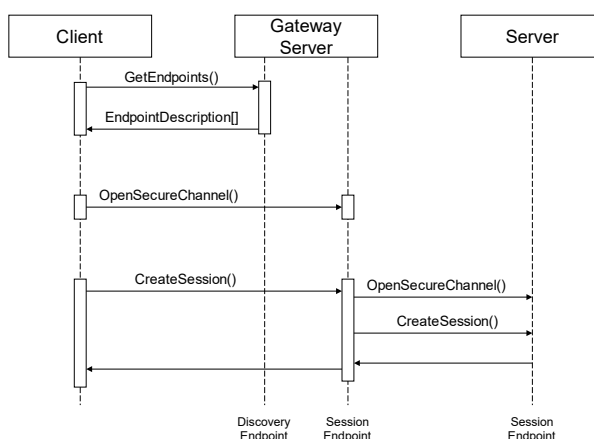


Figure 10 – Using a Gateway Server

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 7.33.
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 7.34.
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 7.14.

5.4.4.3 Service Results

Common *StatusCodes* are defined in Table 182.

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.

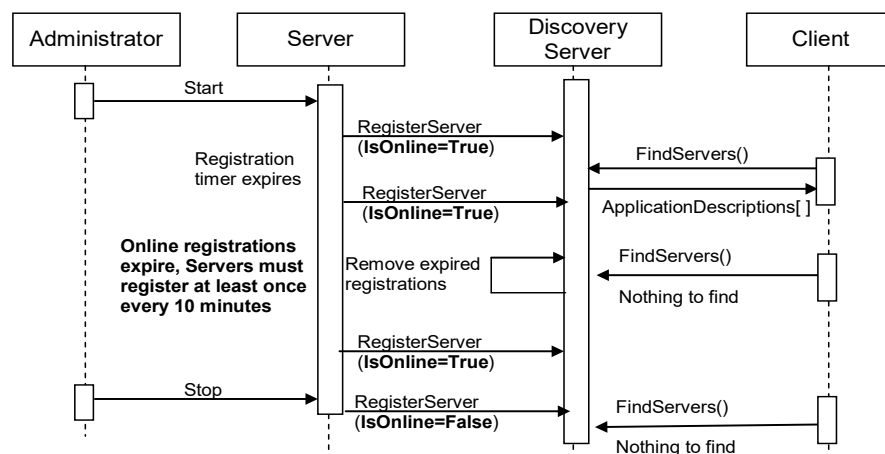


Figure 11 – The Registration Process – Manually Launched Servers

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

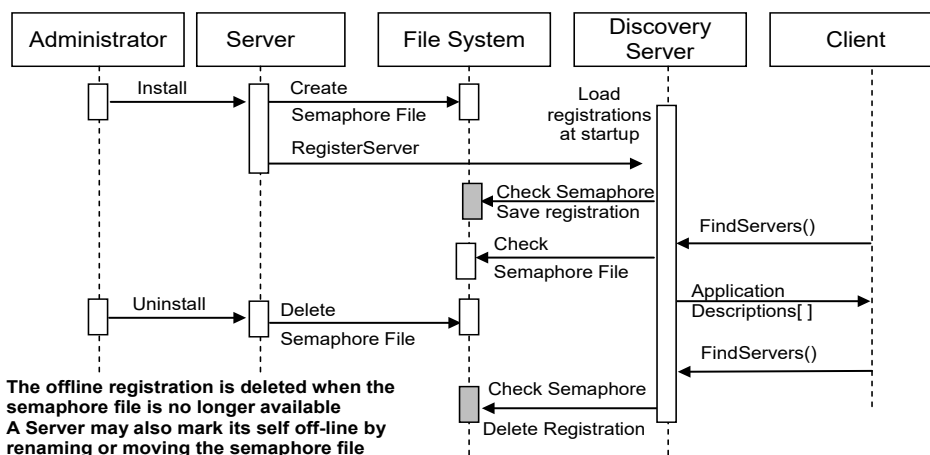


Figure 12 – The Registration Process – Automatically Launched Servers

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 7.33.
Server	RegisteredServer	The <i>Server</i> to register. The type <i>RegisteredServer</i> is defined in 7.32.
Response		
ResponseHeader	ResponseHeader	Common response parameters. The type <i>ResponseHeader</i> is defined in 7.34.

5.4.5.3 Service Results

Table 7 defines the *Service* results specific to this *Service*. Common *StatusCodes* are defined in Table 182.

Table 7 – RegisterServer Service Result Codes

Symbolic Id	Description
Bad_InvalidArgument	See Table 182 for the description of this result code.
Bad_ServerUriInvalid	See Table 182 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.

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 7.33.
Server	RegisteredServer	The <i>Server</i> to register. The type <i>RegisteredServer</i> is defined in 7.32.
discoveryConfiguration []	ExtensibleParameter DiscoveryConfiguration	Additional configuration settings for the <i>Server</i> to register. The <i>discoveryConfiguration</i> is an extensible parameter type defined in 7.13. 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 7.34.
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 182.

Table 9 – RegisterServer2 Service Result Codes

Symbolic Id	Description
Bad_InvalidArgument	See Table 182 for the description of this result code.
Bad_ServerUriInvalid	See Table 182 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 182 for the description of this result code.

5.4.6.4 StatusCodes

Table 10 defines values for the operation level *configurationResults* parameters that are specific to this *Service*. Common *StatusCodes* are defined in Table 183.

Table 10 – RegisterServer2 Operation Level Result Codes

Symbolic Id	Description
Bad_NotSupported	See Table 183 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 62541-2.

The *SecureChannel Services* are unlike other *Services* because they are not implemented directly by the *OPC UA Application*. Instead, they are provided by the *Communication Stack* on which the *OPC UA Application* is built. For example, an OPC UA *Server* may be built on a 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 sign and encrypt *Messages* sent across the network to ensure *Confidentiality* and *Integrity*. 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 sign 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*.

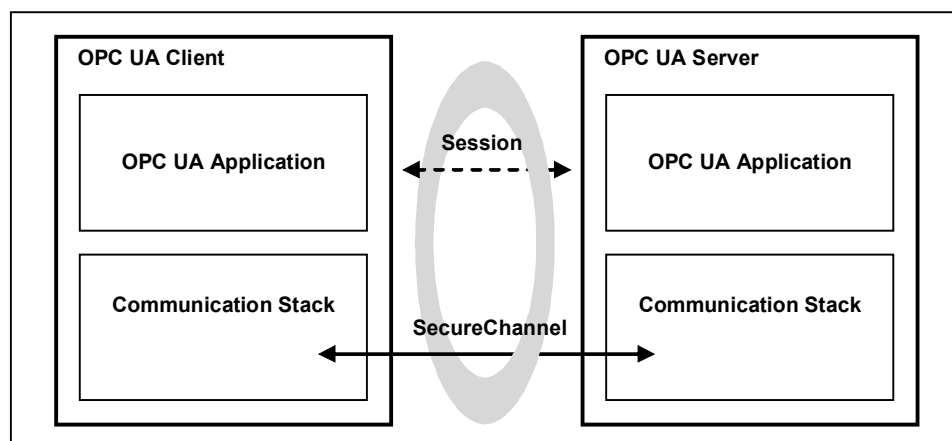


Figure 13 – SecureChannel and Session Services

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 unused *SecureChannel* that has no *Session* assigned before reaching the maximum number of supported *SecureChannels*. When *Session-less Service* invocation is done through a transport mapping that requires the *OpenSecureChannel Service*, the *Server* shall maintain a last used time for the *SecureChannel* to detect the oldest unused *SecureChannel*.

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.mycompany.com 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*. *Servers* shall not append the 'local' top level domain to any domains declared in their *Certificate*; an unqualified domain name is used if a more appropriate qualifier does not exist. *Clients* using a URL returned from an LDS-ME shall remove the 'local' top level domain when checking the domain against the *Server Certificate*.

Clients should be prepared to replace the *HostName* and port returned in the *EndpointDescription* with the *HostName* or the IP addresses and the port they used to call *GetEndpoints*. The *Client* shall still execute the *HostName* verification comparing the *HostName* used by the *Client* to create the *SecureChannel* with the *HostName* list in the *Server Certificate*. See Table 106 for more details.

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 7.33.
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 7.3. 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 creates a new <i>SecurityToken</i> for a new <i>SecureChannel</i> . RENEW 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 7.20. A <i>SecureChannel</i> may need to be created even if the <i>securityMode</i> is <i>NONE</i> . 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> shall 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 7.34 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. The revised lifetime shall be used by the <i>Client</i> to renew a <i>SecureChannel</i> before it expires even if the <i>MessageSecurityMode</i> is <i>NONE</i> .
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 182.

Table 12 – OpenSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecurityChecksFailed	See Table 182 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 182 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 182 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 182 for the description of this result code.
Bad_CertificateUriInvalid	See Table 182 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 182 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 182 for the description of this result code.
Bad_CertificateUntrusted	See Table 182 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 182 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 182 for the description of this result code.
Bad_CertificateRevoked	See Table 182 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 182 for the description of this result code.
Bad_RequestTypeInvalid	The <i>SecurityToken</i> request type is not valid.
Bad_SecurityModeRejected	The security mode does not meet the requirements set by the server.
Bad_SecurityPolicyRejected	The security policy does not meet the requirements set by the <i>Server</i> .
Bad_SecureChannelIdInvalid	See Table 182 for the description of this result code.
Bad_NonceInvalid	See Table 182 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 <i>RENEW</i> .

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 7.33.
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 7.34 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 182.

Table 14 – CloseSecureChannel Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 182 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 *ActivateSession*. For the *N Sessions* supported by a *Server*, the *Server* shall support *N+1 SecureChannels*. A *Server* may support multiple *Sessions* for one *SecureChannel*.

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 7.36 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 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 organizations 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. When the communication connection fails, the *Client* should try to create a new communication connection and call *ActivateSession* again. See 6.7 for more details.

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 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 document. 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.

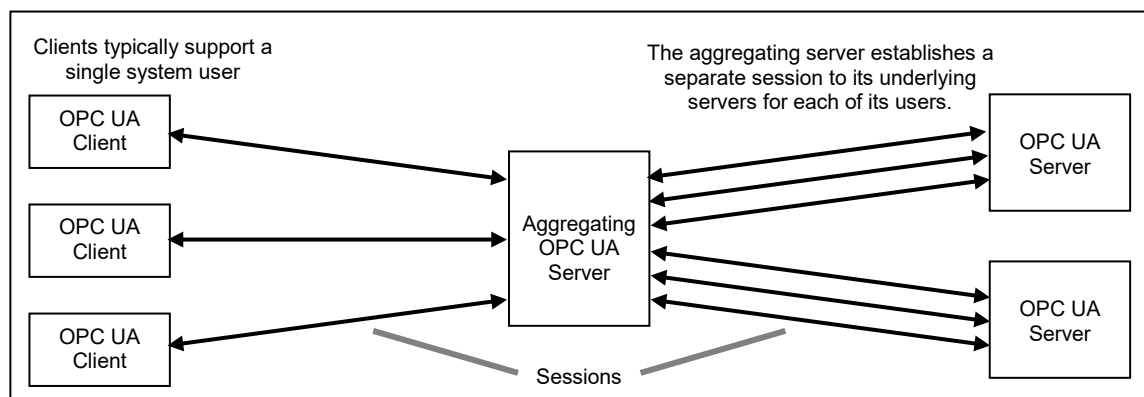


Figure 14 – Multiplexing users on a Session

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 7.33.
clientDescription	Application Description	Information that describes the <i>Client</i> application. The type <i>ApplicationDescription</i> is defined in 7.2.
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 7.14.
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 7.3). 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 null or empty 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 7.3. If the <i>securityPolicyUri</i> is None, the <i>Server</i> shall ignore the <i>ApplicationInstanceCertificate</i> . A <i>Client</i> shall prove possession by using the private key to sign the <i>Nonce</i> provided by the <i>Server</i> in the response. For <i>SecureChannels</i> that use the <i>Application Instance Certificate</i> the <i>Server</i> shall verify that this <i>Certificate</i> is the same as the one it used to create the <i>SecureChannel</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 7.34 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 7.36.
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. For <i>SecureChannels</i> that use the <i>Application Instance Certificate</i> 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> .

Name	Type	Description
		The <i>ApplicationInstanceCertificate</i> type is defined in 7.3. If the <i>securityPolicyUri</i> is <i>None</i> 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. All <i>Endpoints</i> are returned if the <i>serverUri</i> is null or empty. The <i>EndpointDescription</i> type is defined in 7.14. The <i>Client</i> 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 or empty. Only the recommended parameters shall be verified by the <i>Client</i> .
serverSoftwareCertificates []	SignedSoftwareCertificate	This parameter is no longer used 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 7.37.
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 182.

Table 16 – CreateSession Service Result Codes

Symbolic Id	Description
Bad_SecureChannelIdInvalid	See Table 182 for the description of this result code.
Bad_NonceInvalid	See Table 182 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 182 for the description of this result code.
Bad_CertificateTimeInvalid	See Table 182 for the description of this result code.
Bad_CertificateIssuerTimeInvalid	See Table 182 for the description of this result code.
Bad_CertificateHostNameInvalid	See Table 182 for the description of this result code.
Bad_CertificateUriInvalid	See Table 182 for the description of this result code.
Bad_CertificateUseNotAllowed	See Table 182 for the description of this result code.
Bad_CertificateIssuerUseNotAllowed	See Table 182 for the description of this result code.
Bad_CertificateUntrusted	See Table 182 for the description of this result code.
Bad_CertificateRevocationUnknown	See Table 182 for the description of this result code.
Bad_CertificateIssuerRevocationUnknown	See Table 182 for the description of this result code.
Bad_CertificateRevoked	See Table 182 for the description of this result code.
Bad_CertificateIssuerRevoked	See Table 182 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 182 for the description of this result code.
Bad_SecurityPolicyRejected	See Table 182 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 *UserNamelIdentityToken* 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 need to tear down connections to an underlying system and re-establish them using the new credentials. A *Server* shall re-evaluate the permissions of all *MonitoredItems* in *Subscriptions* assigned to the *Session* after a user identity change.

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* shall 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 7.33.
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 7.37.
clientSoftwareCertificates []	SignedSoftwareCertificate	Reserved for future use. The <i>SignedSoftwareCertificate</i> type is defined in 7.38.
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 null or empty 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 7.41. 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 or empty. The <i>SignatureAlgorithm</i> depends on the identity token type. The <i>SignatureData</i> type is defined in 7.37.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.34 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 7.39 for <i>StatusCode</i> definition).
diagnosticInfos []	DiagnosticInfo	List of diagnostic information associated with <i>SoftwareCertificate</i> validation errors (see 7.12 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 182.

Table 18 – ActivateSession Service Result Codes

Symbolic Id	Description
Bad_IdentityTokenInvalid	See Table 182 for the description of this result code.
Bad_IdentityTokenRejected	See Table 182 for the description of this result code.
Bad_UserAccessDenied	See Table 182 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.
Bad_SecurityPolicyRejected	See Table 182 for the description of this result code.
Good_PasswordChangeRequired	The log-on for the user succeeded but the user is required to change the password. The activated <i>Session</i> has limited rights and is mainly available to change the password. The detailed definitions for <i>UserManagement</i> , restrictions for <i>Sessions</i> and the <i>Method ChangePassword</i> used to set a new password are defined in IEC 62541-18. This result code is only used by <i>Servers</i> that support the <i>Method ChangePassword</i> .

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:

- u) It stops accepting requests for the *Session*. All subsequent requests received for the *Session* are discarded.
- v) 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.
- w) 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 7.33 for <i>RequestHeader</i> definition).
deleteSubscriptions	Boolean	If the value is TRUE, the <i>Server</i> deletes all <i>Subscriptions</i> associated with the <i>Session</i> . If the value is FALSE, the <i>Server</i> keeps the <i>Subscriptions</i> associated with the <i>Session</i> until they timeout based on their own lifetime.
Response		
responseHeader	ResponseHeader	Common response parameters (see 7.34 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 182.

Table 20 – CloseSession Service Result Codes

Symbolic Id	Description
Bad_SessionIdInvalid	See Table 182 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 7.33 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 7.34 for <i>ResponseHeader</i> definition).
cancelCount	UInt32	Number of cancelled requests.

5.6.5.3 Service results

Common *StatusCodes* are defined in Table 182.

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*.

Calls to *NodeManagement Services* may result in changes to the *AddressSpace* in addition to the requested change. The actual behaviour is *Server* specific.

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 7.33 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 7.16.
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 7.24. 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 7.34 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 7.39 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 7.12 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 182.

Table 23 – AddNodes Service Result Codes

Symbolic Id	Description
Bad_NothingToDo	See Table 182 for the description of this result code.
Bad_TooManyOperations	See Table 182 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 183.