OPC Unified Architecture

Specification

Part 12: Discovery and Global Services

Release 1.04

February 7, 2018

Specification Type	Industry Standard Specification	Comments:	
Title:	OPC Unified Architecture **Discovery and Global Services**	Date:	**February 7, 2018**
Version:	**Release 1.04**	Software Source:	MS-Word OPC UA Part 12 - Discovery and Global Services Release 1.04 Specification.docx
Author:	OPC Foundation	Status:	**Release**

CONTENTS

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

Revision 1.04 Highlights

The following table includes the Mantis issues resolved with this revision.

Mantis ID	Summary	Resolution
3046	There should be a subtype of "CertificateType" for user certificates.	Added 7.5.14.
3062	Add references Discovery Endpoint term defined in Part 4.	Add references to the new term in clauses 4 and 6.
3185	Not precise enough about the visibility of objects that have security related access restrictions.	Added restrictions to 7.7.2.
3343	Missing defaults for MaxTrustlistSize.	Added default to 7.7.3.
3501	7.6.4 clarification of Domain Names.	Added text to 7.6.4.
3502	7.6.4 RSA key length of 1024 is ok?.	Removed text from 7.6.4.
3534	ApplyChanges() - clarifications needed regarding private keys and existing/new connections.	Added text to 7.7.5.
3582	RegisterApplication and handling of duplications.	Added text to 6.3.6.
3584	The spec does not actually allow the client to use GDS to discover server's applicationUri and other info about the server.	Added QueryApplications Method in 6.3.10.
3627	Need a way to manage Broker credentials.	Added clause 8.
3648	Clarify who can use the Pull and Push Models.	Clarified text in 7.4.
3751	Need model to Request Tokens from Authorization Services.	Added clause 9.
3752	Clarify encoding of PEM private keys.	Add reference to RFC 5958 in 7.6.4.
3839	Change Part Name to Discovery and Global Services.	Changed Part Name.
3892	Required trust list update time should be indicated by GDS.	Updated 7.5.2, 7.5.9 and 7.5.10.
3898	LDS-ME must return IP addresses in order for the multi-subnet use case to work.	Added 6.2.5.
4081	CA certificates with CRLs with AddCertificate/RemoveCertificate.	Updated 7.5.5.

OPC UNIFIED ARCHITECTURE

Part 12: Discovery and Global Services

1 Scope

This part specifies how OPC Unified Architecture (OPC UA) *Clients* and *Servers* interact with *DiscoveryServers* when used in different scenarios. It specifies the requirements for the *LocalDiscoveryServer, LocalDiscoveryServer-ME and GlobalDiscoveryServer.* It also defines information models for *Certificate* management*, KeyCredential m*anagement and *Authorization Services*.

2 Normative references

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

Part 1: **OPC UA Specification: Part 1** – Overview and Concepts

http://www.opcfoundation.org/UA/Part1/

Part 2: **OPC UA Specification: Part 2** – Security Model

http://www.opcfoundation.org/UA/Part2/

Part 3: **OPC UA Specification: Part 3** – Address Space Model

http://www.opcfoundation.org/UA/Part3/

Part 4: **OPC UA Specification: Part 4** – Services

http://www.opcfoundation.org/UA/Part4/

Part 5: **OPC UA Specification: Part 5** – Information Model

http://www.opcfoundation.org/UA/Part5/

Part 6: **OPC UA Specification: Part 6** – Mappings

http://www.opcfoundation.org/UA/Part6/

Part 7: **OPC UA Specification: Part 7** – Profiles

http://www.opcfoundation.org/UA/Part7/

Part 9: **OPC UA Specification: Part 9** – Alarms and Conditions

http://www.opcfoundation.org/UA/Part9/

Part 14: **OPC UA Specification: Part 14** - PubSub

http://www.opcfoundation.org/UA/Part14/

Auto-IP: Dynamic Configuration of IPv4 Link-Local Addresses

http://www.ietf.org/rfc/rfc3927.txt

DNS-Name: Domain Names – Implementation and Specification

http://www.ietf.org/rfc/rfc1035.txt

DHCP: Dynamic Host Configuration Protocol

http://www.ietf.org/rfc/rfc2131.txt

mDNS: Multicast DNS

http://www.ietf.org/rfc/rfc6762.txt

DNS-SD: DNS Based Service Discovery

http://www.ietf.org/rfc/rfc6763.txt

RFC 5958: Asymmetric Key Packages

http://www.ietf.org/rfc/rfc5958.txt

PKCS #10: Certification Request Syntax Specification

http://www.ietf.org/rfc/rfc2986.txt

PKCS #12: Personal Information Exchange Syntax

http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11301-wp-pkcs-12v1-1-personal-information-exchange-syntax.pdf

RFC 7030: Enrollment over Secure Transport

http://www.ietf.org/rfc/rfc7030.txt

DI: OPC Unified Architecture for Devices (DI)

https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture-for-devices-di/

ADI: OPC Unified Architecture for Analyzer Devices (ADI)

https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture-for-analyzer-devices-adi/

PLCopen: OPC Unified Architecture / PLCopen Information Model

https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture-plcopen-information-model/

FDI: OPC Unified Architecture for FDI

https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture-for-fdi/

ISA-95: ISA-95 Common Object Model

https://opcfoundation.org/developer-tools/specifications-unified-architecture/isa-95-common-object-model/

X.500: ISO/IEC 9594-1:2017 – The Directory

https://www.iso.org/standard/72550.html

3 Terms, definitions, and conventions

3.1 Terms and definitions

For the purposes of this document the following terms and definitions as well as the terms and definitions given in Part 1, Part 2, Part 3, Part 4, Part 6 and Part 9 apply.

3.1.1
CertificateManagement Server
a software application that manages the *Certificates* used by *Applications* in an administrative domain.

3.1.2
Certificate Group
a context used to describe the *Trust List* and *Certificate(s)* associated with an *Application*.

3.1.3
Certificate Request
a PKCS #10 encoded structure used to request a new *Certificate* from a *Certificate Authority*.

3.1.4
KeyCredential

a unique identifier and a secret used to access a *Server*, an *Authorization Service* or a *Broker*.

Note 1 to entry: a user name and password is an example of a credential.

3.1.5
KeyCredentialService
a software application that provides *KeyCredentials* needed to access a *Server*, an *Authorization Service* or a *Broker*.

3.1.6
DirectoryService

a software application, or a set of applications, that stores and organizes information about resources such as computers or services.

3.1.7
DiscoveryServer
an *Application* that maintains a list of OPC UA *Servers* that are available on the network and provides mechanisms for Clients to obtain this list.

3.1.8
DiscoveryUrl
a URL for a network *Endpoint* that provides the information required to connect to a *Client* or *Server*.

3.1.9
GlobalDiscoveryServer (GDS)

a *DiscoveryServer* that maintains a list of OPC UA *Applications* available in an administrative domain.

Note 1 to entry: a GDS may also provide certificate management services.

3.1.10
IPAddress

a unique number assigned to a network interface that allows Internet Protocol (IP) requests to be routed to that interface.

Note 1 to entry: An *IPAddress* for a host may change over time.

3.1.11
LocalDiscoveryServer (LDS)

a *DiscoveryServer* that maintains a list of all *Servers* that have registered with it.

Note 1 to entry: *Servers* normally register with the LDS on the same host.

3.1.12
LocalDiscoveryServer-ME (LDS-ME)
a *LocalDiscoveryServer* that includes the MulticastExtension.

3.1.13
MulticastExtension
an extension to a *LocalDiscoveryServer* that adds support for the mDNS protocol.

3.1.14
MulticastSubnet

a network that allows multicast packets to be sent to all nodes connected to the network.

Note 1 to entry: a *MulticastSubnet* is not necessarily the same as a TCP/IP subnet.

3.1.15
Network Service

a secured resource on a network that provides functionality used by *Clients* and/or *Servers*.

Note 1 to entry: an *Authorization Service* (AS) is an example of a *Network Service*.

3.1.16
ServerCapabilityIdentifier

a short identifier which uniquely identifies a set of discoverable capabilities supported by a *Server*.

Note 1 to entry: the list of the currently defined *ServerCapabilityIdentifiers* is in Annex D.

3.2 Abbreviations and symbols

API	Application Programming Interface
CA	Certificate Authority
CRL	Certificate Revocation List
CSR	Certificate Signing Request
DER	Distinguished Encoding Rules
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EST	Enrolment over Secure Transport
GDS	Global Discovery Server
IANA	The Internet Assigned Numbers Authority
LDAP	Lightweight Directory Access Protocol
LDS	Local Discovery Server
LDS-ME	Local Discovery Server with the Multicast Extension
mDNS	Multicast Domain Name System
NAT	Network Address Translation
PEM	Privacy Enhanced Mail
PFX	Personal Information Exchange
PKCS	Public Key Cryptography Standards
SHA1	Secure Hash Algorithm
SSL	Secure Socket Layer
TLS	Transport Layer Security
UA	Unified Architecture
UDDI	Universal Description, Discovery and Integration

3.3 Conventions for Namespaces

This standard uses multiple namespaces to define *Nodes*. The following abbreviations are used in the definitions for these *Nodes*:

CORE	http://opcfoundation.org/UA/
GDS	http://opcfoundation.org/UA/GDS/

The default namespace for each *Node* is defined at the top of the table. All of the *BrowseNames* in the table use the default namespace unless the *BrowseName* is preceded by one of the above abbreviations.

The *NamespaceMetadataType Object* for the GDS namespace is defined Table 1.

Table 1 – GDS NamespaceMetadataType Object Definition

Attribute	**Value**		
BrowseName	http://opcfoundation.org/UA/GDS/		
Namespace	GDS (see 3.3)		
TypeDefinition	NamespaceMetadataType defined in Part 5.		
References	**NodeClass**	**BrowseName**	**Value**
HasProperty	Variable	NamespaceUri	http://opcfoundation.org/UA/GDS/
HasProperty	Variable	NamespaceVersion	1.04
HasProperty	Variable	NamespacePublicationDate	2016-12-31

4 The Discovery Process

4.1 Overview

The discovery process allows applications to find other applications on the network and then discover how to connect to them. Note that this discussion builds on the discovery related concepts defined in Part 4. Discoverable applications are generally *Servers*, however, some *Clients* will support reverse connections as described in Part 6 and want *Servers* to be able to discover them.

Clients and *Servers* can be on the same host, on different hosts in the same subnet, or even on completely different locations in an administrative domain. The following clauses describe the different configurations and how discovery can be accomplished.

The mechanisms for *Clients* to discover *Servers* are specified in 4.3.

The mechanisms for *Servers* to make themselves discoverable are specified in 4.2.

The *Discovery Services* are specified in Part 4. They are implemented by individual *Servers* and by dedicated *DiscoveryServers*. The following dedicated *DiscoveryServers* provide a way for applications to discover registered OPC UA applications in different situations:

- A *LocalDiscoveryServer* (LDS) maintains discovery information for all applications that have registered with it, usually all applications available on the host that it runs on.
- A *LocalDiscoveryServer* with the *MulticastExtension* (LDS-ME) maintains discovery information for all applications that have been announced on the local *MulticastSubnet.*
- A *GlobalDiscoveryServer* (GDS) maintains discovery information for applications available in an administrative domain.

LDS and LDS-ME are specified in Clause 5. The GDS is specified in Clause 6.

4.2 Registration and Announcement of Applications

4.2.1 Overview

The clause describes how an application registers itself so it can be discovered. Most *Applications* will want other applications to discover them. *Applications* that do not wish to be discovered openly should not register with a *DiscoveryServer*. In this case such *Applications* should only publish a *DiscoveryUrl* via some out-of-band mechanism to be discovered by specific *Applications*.

4.2.2 Hosts with a LocalDiscoveryServer

Applications register themselves with the LDS on the same host if they wish to be discovered. The registration ensures that the applications is visible for local discovery (see 4.3.4) and *MulticastSubnet* discovery if the LDS is a LDS-ME (see 4.3.5).

The OPC UA Standard (Part 4) defines a *RegisterServer2 Service* which provides additional registration information. *All Applications* and *LocalDiscoveryServer* the shall support the *RegisterServer2 Service* and, for backwards compatibility, the older *RegisterServer Service*. If an *Application* encounters an older LDS that returns a *Bad_ServiceUnsupported* error when calling *RegisterServer2 Service* it shall try again with *RegisterServer Service*.

The *RegisterServer2 Service* allows the *Application* to specify zero or more *ServerCapability Identifiers. ServerCapabilityIdentifiers* are short, string identifiers of well-known OPC UA features. *Applications* can use these identifiers as a filter during discovery.

The set of known *ServerCapabilityIdentifiers* is specified in Annex D and is limited to features which are considered to be important enough to report before an application makes a connection. For example, support for the GDS information model or the Alarms information model are *Server* capabilities that have a *ServerCapabilityIdentifier* defined.

Before an application registers with the LDS it should call the *GetEndpoints Service* and choose the most secure endpoint supported by the LDS and then call *RegisterServer2* or *RegisterServer*.

Registration with LDS or LDS-ME is illustrated in Figure 1.

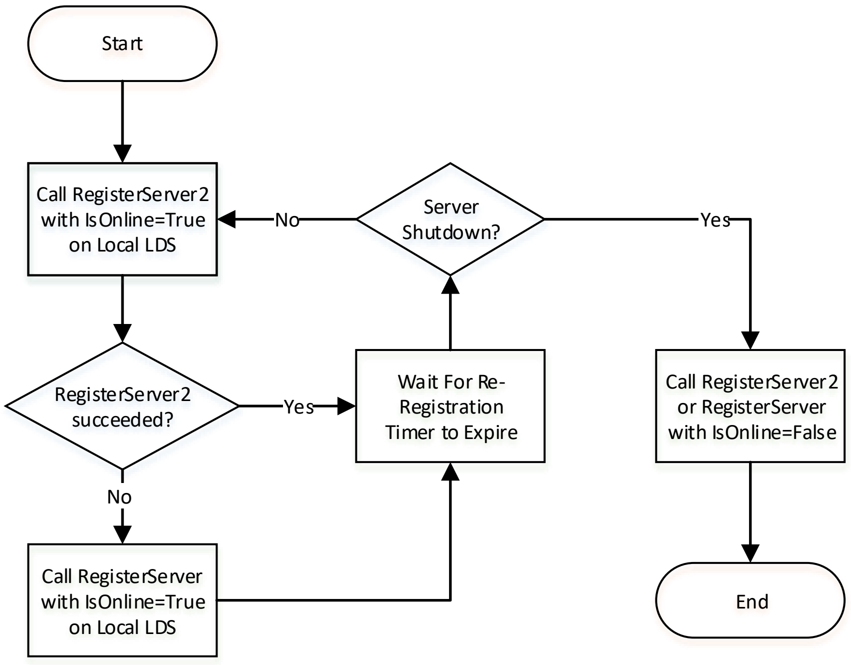
Start
Call RegisterServer2 with IsOnline=True on Local LDS
No
Server Shutdown?
Yes
RegisterServer2 succeeded?
Yes
Wait For Re-Registration Timer to Expire
Call RegisterServer2 or RegisterServer with IsOnline=False
No
Call RegisterServer with IsOnline=True on Local LDS
End

Figure 1 – The Registration Process with an LDS

See Part 4 for more information on the re-registration timer and the *IsOnline* flag.

4.2.3 Hosts without a LocalDiscoveryServer

Dedicated systems (usually embedded systems) with exactly one *Server* installed may not have a separate LDS. Such *Servers* shall become their own LDS or LDS-ME by implementing *FindServers* and *GetEndpoints Services* at the well-known address for an LDS. They should also announce themselves on the *MulticastSubnet* with a basic *MulticastExtension*. This requires a small subset of an mDNS Responder (see mDNS and Annex C) that announces the *Server* and responds to mDNS probes. The *Server* may not provide the caching and address resolution implemented by a full mDNS Responder.

4.3 The Discovery Process for Clients to Find Servers

4.3.1 Overview

The discovery process allows *Clients* to find *Servers* on the network and then discover how to connect to them. Once a *Client* has this information it can save it and use it to connect directly to the *Server* again without going through the discovery process. *Clients* that cannot connect with the saved connection information should assume the *Server* configuration has changed and therefore repeat the discovery process.

A *Client* has several choices for finding *Servers*:

- Out-of-band discovery (i.e. entry into a GUI) of a *DiscoveryUrl* for a *Server*;
- Calling *FindServers* on the LDS installed on the *Client* host;
- Calling *FindServers* on a remote LDS, where the *HostName* for the remote host is manually entered*;*
- Calling *FindServersOnNetwork* (see Part 4) on the LDS-ME installed on *Client* host;
- Supporting the LDS-ME functionality locally in the Client.
- Searching for *Servers* known to a *GlobalDiscoveryServer*.

The *DiscoveryUrl* provides all of the information a *Client* needs to connect to a *DiscoveryEndpoint* (see 4.3.3).

4.3.2 Security

Clients should be aware of rogue *DiscoveryServers* that might direct them to rogue *Servers*. *Clients* can use the SSL/TLS server certificate (if available) to verify that the *DiscoveryServer* is a server that they trust and/or ensure that they trust any *Server* provided by the *DiscoveryServer*. See Part 2 for a detailed discussion of these issues.

4.3.3 Simple Discovery with a DiscoveryUrl

Every *Server* has one or more *DiscoveryUrls* that allow access to its *Endpoints*. Once a *Client* obtains (e.g. via manual entry into a form) the *DiscoveryUrl* for the *Server,* it reads the *EndpointDescriptions* using the *GetEndpoints Service* defined in Part 4*.*

The discovery process for this scenario is illustrated in Figure 2.

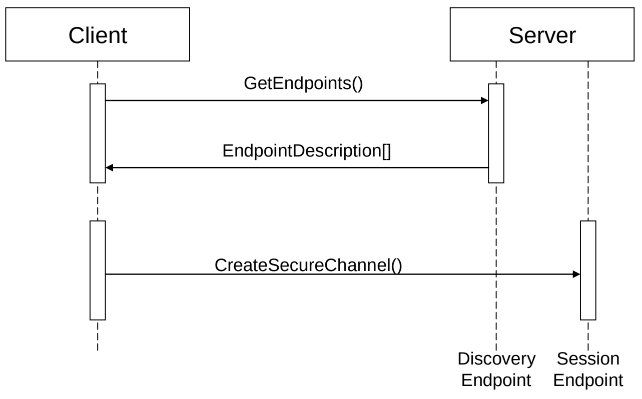

Figure 2 – The Simple Discovery Process

4.3.4 Local Discovery

In many cases *Clients* do not know which *Servers* exist but possibly know which hosts might have *Servers* on them. In this situation the *Client* will look for the *LocalDiscoveryServer* on a host by constructing a *DiscoveryUrl* using the Well-Known Addresses defined in Part 6.

If a *Client* finds a *LocalDiscoveryServer* then it will call the *FindServers Service* on the LDS to obtain a list of *Servers* and their *DiscoveryUrls*. The *Client* would then call the *GetEndpoints* service for one of the *Servers* returned. The discovery process for this scenario is illustrated in Figure 3.

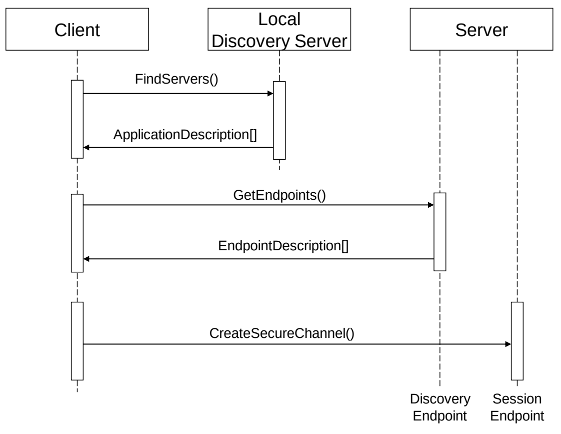

Figure 3 – The Local Discovery Process

4.3.5 MulticastSubnet Discovery

In some situations *Clients* will not know which hosts have *Servers*. In these situations the *Client* will look for a *LocalDiscoveryServer* with the *MulticastExtension* on its local host and requests a list of *DiscoveryUrls* for *Servers* and *DiscoveryServers* available on the *MulticastSubnet*.

The discovery process for this scenario is illustrated in Figure 4.

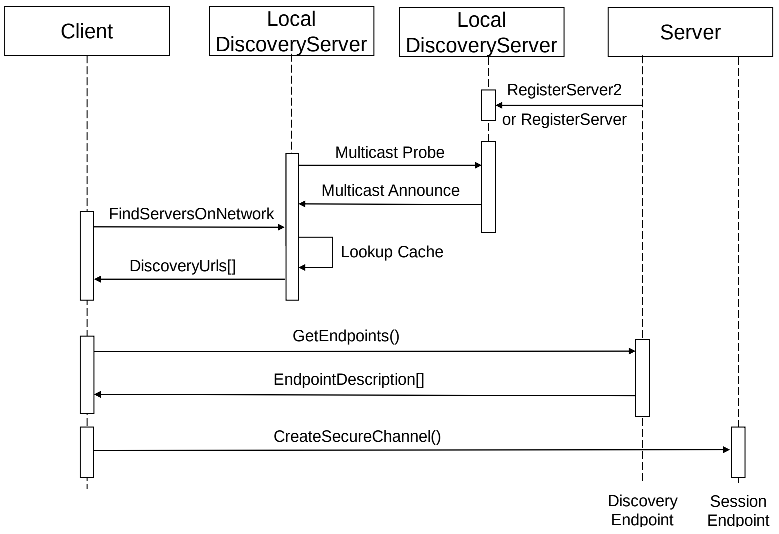

Figure 4 – The MulticastSubnet Discovery Process

In this scenario the *Server* uses the *RegisterServer2 Service* to tell a *LocalDiscoveryServer* to announce the *Server* on the *MulticastSubnet*. The *Client* will receive the *DiscoveryUrl* and *ServerCapabilityIdentifiers* for the Server when it calls *FindServersOnNetwork* and then connects directly to the *Server*. When a *Client* calls *FindServers* it only receives the *Servers* running on the same host as the LDS.

Clients running on embedded systems may not have a LDS-ME available on the system, These *Clients* can support an mDNS Responder which understands how OPC UA concepts are mapped to mDNS messages and maintains the same table of servers as maintained by the LDS-ME. This mapping is described in Annex C.

4.3.6 Global Discovery

A GDS is an OPC UA *Server* which allows *Clients* to search for *Servers* in the administrative domain. It may also provide Certificate Services (see Clause 7). It provides *Methods* that allow applications to search for other applications (See Clause 6). To access the GDS, the *Client* will create a *Session* with the *GDS* and use the *Call* service to invoke the *QueryApplications Method* (see 6.3.11). The *QueryServers Method* is similar to the *FindServers* service except that it provides more advanced search and filter criteria. The discovery process is illustrated in Figure 5.

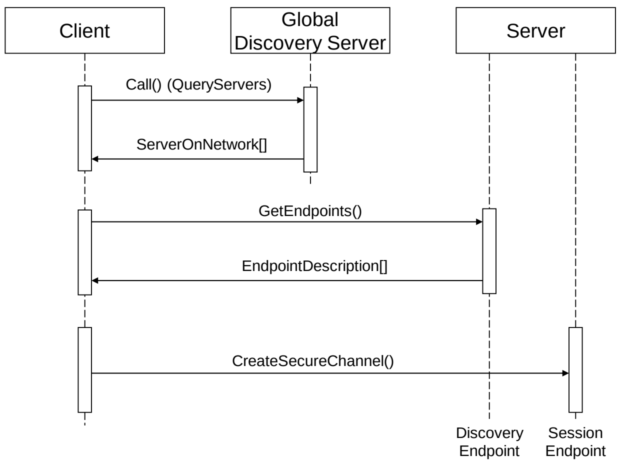

Figure 5 – The Global Discovery Process

The GDS may be coupled with any of the previous network architectures. For each *MulticastSubnet,* one or more LDSs may be registered with a GDS.

The *Client* can also be configured with the URL of the GDS using an out of band mechanism.

The complete discovery process is shown in Figure 6.

4.3.7 Combined Discovery Process for Clients

The use cases in the preceding clauses imply a number of choices that have to be made by *Clients* when a *Client* needs to connect to a *Server*. These choices are combined together in Figure 6.

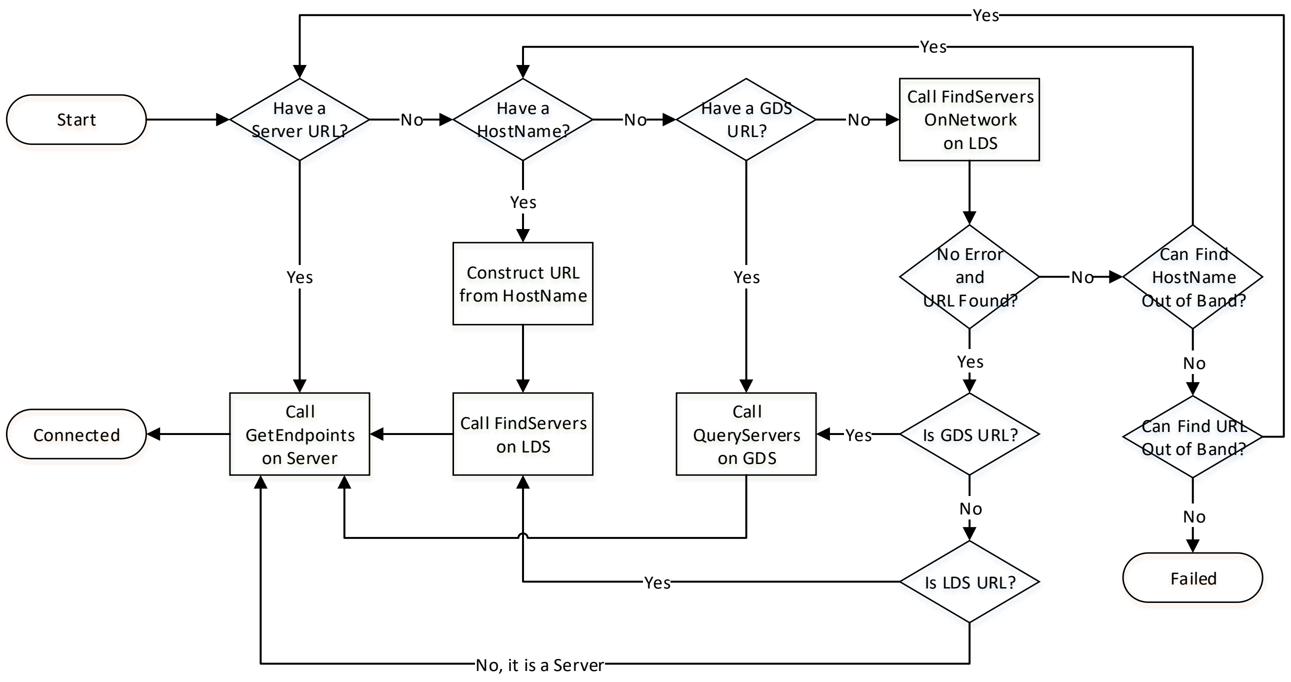

Figure 6 – The Discovery Process for Clients

FindServersOnNetwork can be called on the local LDS, however, It can also be called on a remote LDS which is part of a different *MulticastSubnet*.

An out-of-band mechanism is a way to find a URL or a *HostName* that is not described by this standard. For example, a user could manually enter a URL or use system specific APIs to browse the network neighbourhood.

A *Client* that goes through the discovery process can save the URL that was discovered. If the *Client* restarts later it can use that URL and bypass the discovery process. If reconnection fails the *Client* will have to go through the process again.

5 Local Discovery Server

5.1 Overview

Each host that could have multiple discoverable applications installed should have a standalone *LocalDiscoveryServer* installed. The *LocalDiscoveryServer* shall expose one or more *Endpoints* which support the *FindServers* and *GetEndpoints* services defined in Part 4 for all applications on the host. In addition, the *LocalDiscoveryServer* shall provide at least one *Endpoint* which implements the *RegisterServer* service for these applications .

In systems (usually embedded systems) with exactly one *Server* installed this *Server* may also be the LDS (see 4.2.3).

An LDS-ME will announce all applications that it knows about on the local *MulticastSubnet*. In order to support this, a *LocalDiscoveryServer* supports the *RegisterServer2 Service* defined in Part 4. For backward compatibility a *LocalDiscoveryServer* also supports the *RegisterServer Service* which is defined in Part 4.

Each host with OPC UA Applications (Clients and Servers) installed should have a *LocalDiscoveryServer* with a *MulticastExtension*.

The *MulticastExtension* incorporates the functionality of the mDNS Responder described in the Multicast DNS (mDNS) specification (see mDNS). In addition the *LocalDiscoveryServer* that supports the *MulticastExtension* supports the *FindServersOnNetwork Service* described in Part 4.

5.2 Security Considerations for Multicast DNS

The Multicast DNS (mDNS) specification is used for various commercial and consumer applications. This provides a benefit in that implementations exist, however, system administrators could choose to disable Multicast DNS operations. For this reason, *Applications* shall not rely on Multicast DNS capabilties.

Multicast DNS operations are insecure because of their nature; therefore they should be disabled in environments where an attacker could cause problems by impersonating another host. This risk is minimized if OPC UA security is enabled and all *Applications* use *Certificate TrustLists* to control access.

6 Global Discovery Server

6.1 Overview

The *LocalDiscoveryServer* is useful for networks where the host names can be discovered. However, this is typically not the case in large systems with multiple servers on multiple subnets. For this reason there is a need for an enterprise wide *DiscoveryServer* called a *GlobalDiscoveryServer*. The *GlobalDiscoveryServer* (GDS) is an OPC UA *Server* which allows *Clients* to search for *Servers* in the administrative domain. It provides methods that allow applications to register themselves and to search for other applications.

The essential element of a *GlobalDiscoveryServer* (GDS) is that it can provide the *Certificate* management services defined in Clause 7. These services can simplify *Certificate* management even in medium to small systems, therefore, a GDS can be deployed in smaller systems. Different implementations are expected. Some of them will likely provide a front-end to an existing *DirectoryService* such as LDAP (See Annex E). By standardizing on an OPC UA based interface, OPC UA *Clients* do not need to have knowledge of different *DirectoryServices.*

If an administrator registers a *LocalDiscoveryServer* with the GDS, then the GDS shall periodically update its database by calling *FindServersOnNetwork* or *FindServers* on the LDS. Figure 7 shows the relationship between a GDS and the LDS-ME or LDS.

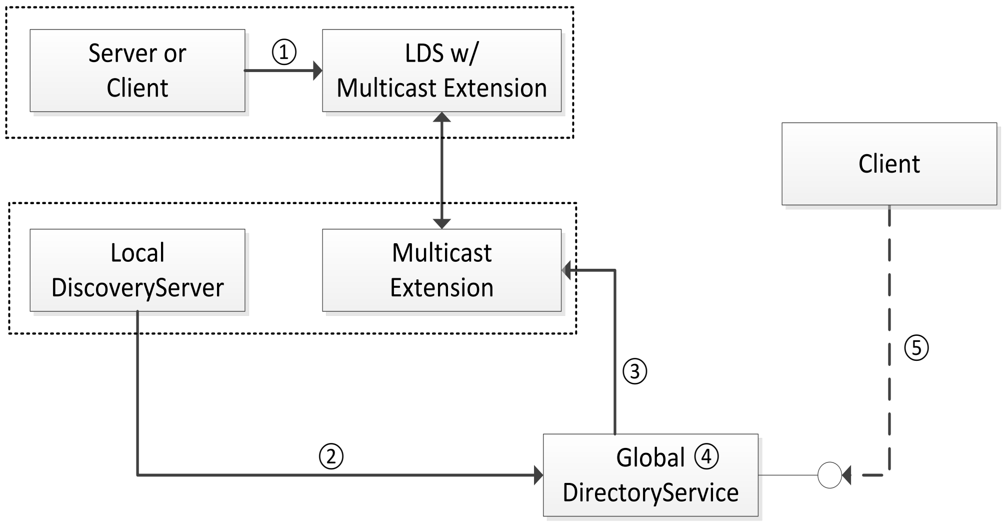

Figure 7 – The Relationship Between GDS and other components

The steps shown in Figure 7 are:

1	The Server calls *RegisterServer2* on the LDS running on the same machine.
2	The administrator registers LDS-ME installations with the GDS.
3	The GDS calls *FindServersOnNetwork* on the LDS-ME to find all applications on the same *MulticastSubnet*.
4	The GDS creates a record for each application returned by the LDS-ME. These records shall be approved before they are made available to *Clients* of the GDS. This approval can be obtained from an *Administrator or* the GDS can connect to the *Server* and verifies that it has a trusted *Certfiicate.*
5	The *Client* calls *QueryServers Method* on the GDS to discover applications.

The *Information Model* used for registration and discovery is shown in clause 6.2**.** Any *Client* shall be able to call the *QueryServers Method* to find applications known to GDS. The complete definitions for each of the types used are described in clause 7.5.

6.2 Network Architectures

6.2.1 Overview

The discovery mechanisms defined in this standard are expected to be used in many different network architectures. The following three architectures are Illustrated:

- Single *MulticastSubnet*;
- Multiple *MulticastSubnets*;
- No *MulticastSubnet* (or multiple *MulticastSubnets* with exactly one host each);

A *MulticastSubnet* is a network segment where all hosts on the segment can receive multicast packets from the other hosts on the segment. A physical LAN segment is typically a

MulticastSubnet unless the administrator has specifically disabled multicast communication. In some cases multiple physical LAN segments can be connected as a single *MulticastSubnet*

6.2.2 Single MulticastSubnet

The Single *MulticastSubnet* Architecture is shown in Figure 8.

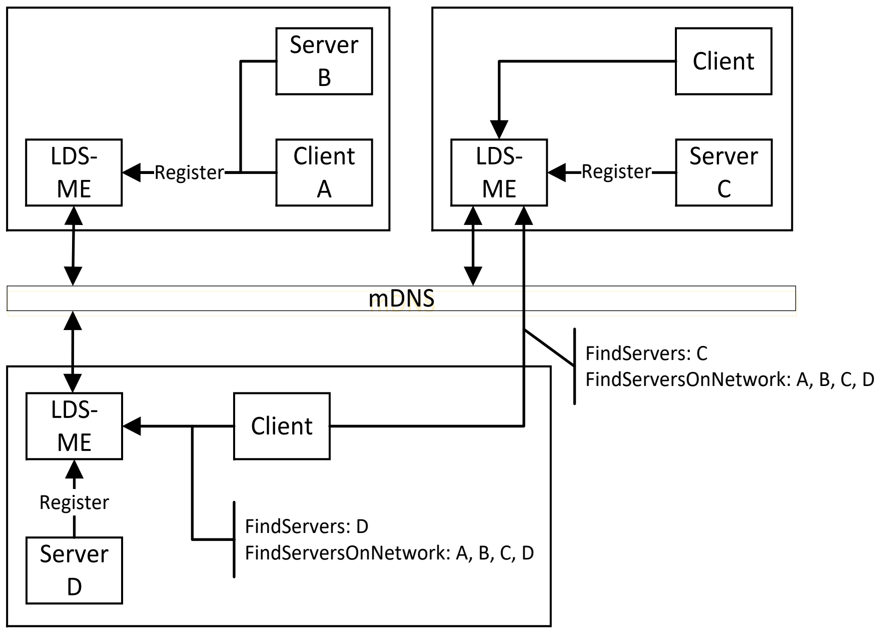

Figure 8 – The Single MulticastSubnet Architecture

In this architecture every host has an LDS-ME and uses mDNS to maintain a cache of the applications on the *MulticastSubnet*. A *Client* can call *FindServersOnNetwork* on any LDS-ME and receive the same set of applications. When a *Client* calls *FindServers* it only receives the applications running on the same host as the LDS.

6.2.3 Multiple MulticastSubnet

The Multiple *MulticastSubnet* Architecture is shown in Figure 9.

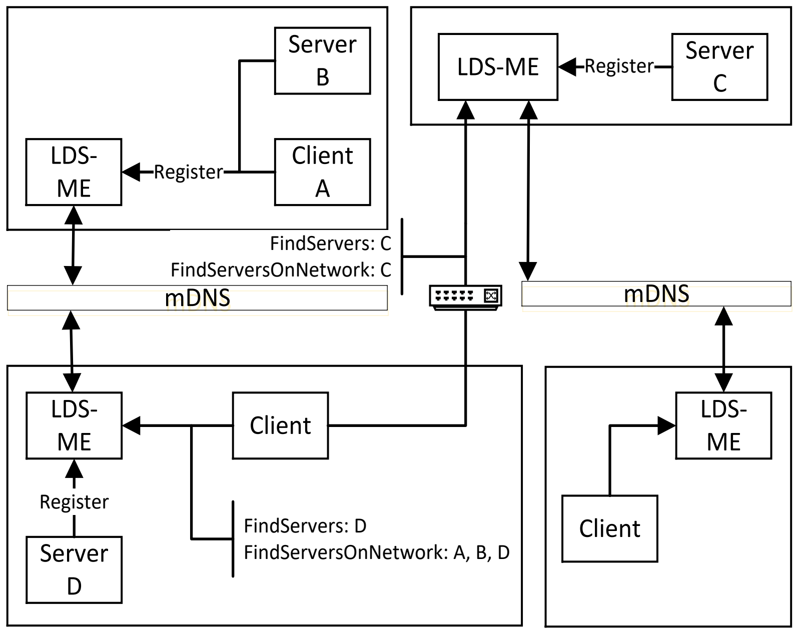

Figure 9 – The Multiple MulticastSubnet Architecture

This architecture is the same as the previous architecture except in this architecture the mDNS messages do not pass through routers connecting the *MulticastSubnets*. This means that a *Client* calling *FindServersOnNetwork* will only receive a list of applications running on the *MulticastSubnets* that the LDS-ME is connected to.

A *Client* that wants to connect to a remote *MulticastSubnet* shall use out of band discovery (i.e. manual entry) of a *HostName* or *DiscoveryUrl*. Once a *Client* finds an LDS-ME on a remote *MulticastSubnet* it can use *FindServersOnNetwork* to discover all applications on that *MulticastSubnet.*

6.2.4 No MulticastSubnet

The No *MulticastSubnet* Architecture is shown in Figure 10.

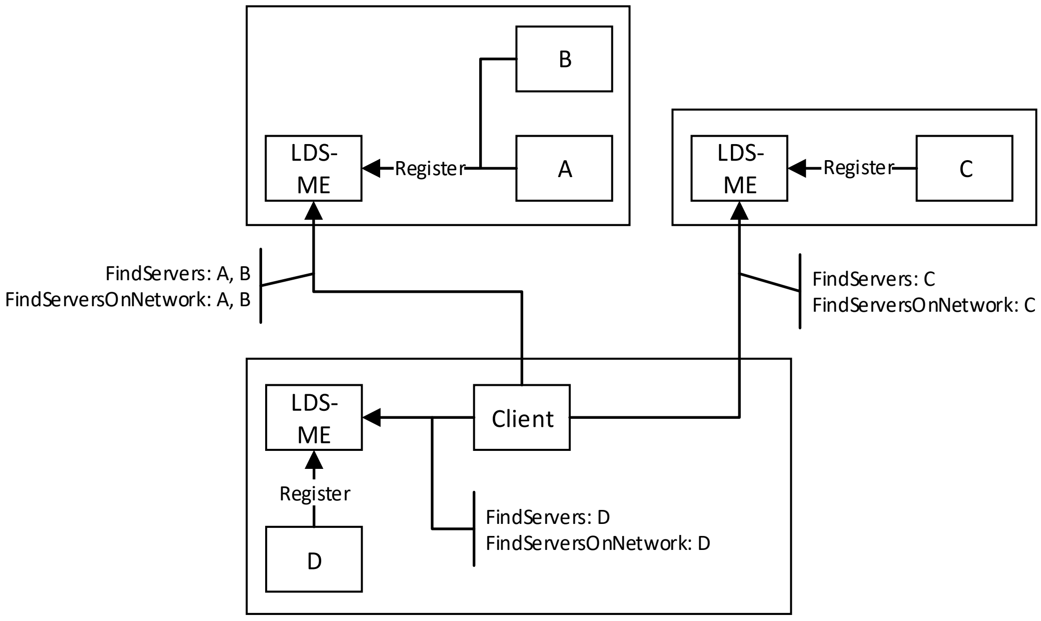

Figure 10 – The No MulticastSubnet Architecture

In this architecture the mDNS is not used at all because the Administrator has disabled multicast at a network level or by turning off multicast capabilities of each LDS-ME.

A *Client* that wants to discover a applications needs to use an out of band mechanism to find the *HostName* and call *FindServers* on the LDS of that host. *FindServersOnNetwork* may also work but it will never return more than what *FindServers* returns.

6.2.5 Domain Names and MulticastSubnets

The mDNS specification requires that fully qualified domain name be annouced on the network. If a *Server* is not configured with a fully qualified domain name then mDNS requires that the 'local' top level domain be appended to the domain names. The 'local' top level domain indicates that the domain can only be consided to be unique within the subnet where the domain name was used. This means *Clients* need to be be aware that URLs received from any LDS-ME other than the one on the *Client's* machine could contain 'local' domains which are not reachable or will connect to a different machine with the same domain name that happens to be on the same subnet as the *Client*. It is recommended that *Clients* ignore all URLs with the 'local' top level domain unless they are returned from the LDS-ME running on the same machine.

System administrators can eliminate this problem by configuring a normal DNS with the fully qualilfied domain names for all machines which need to be accessed by *Clients* outside the *MulticastSubnet*.

Servers configured with fully qualified domain names should specify the fully qualified domain name in its *ApplicationInstance Certificate*. *Servers* shall not specify domains with the 'local' top level domain in their *Certificate*. *Clients* using a URL returned from an LDS-ME shall ignore the 'local' top level domain when checking the domain against the *Server Certificate*.

Note that domain name validation is a necessary but not sufficient check against rogue *Servers* or man-in-the-middle attacks when *Server Certificates* do not contain fully qualified domain names. The *Certificate* trust relationship established by administrators is the primary mechanism used to protect against these risks.

6.3 Information Model

6.3.1 Overview

The *GlobalDiscoveryServer Information Model* used for *discovery* is shown in Figure 11. Most of the interactions between the *GlobalDiscoveryServer* and *Application* administrator or the *Client* will be via *Methods* defined on the *Directory* folder.

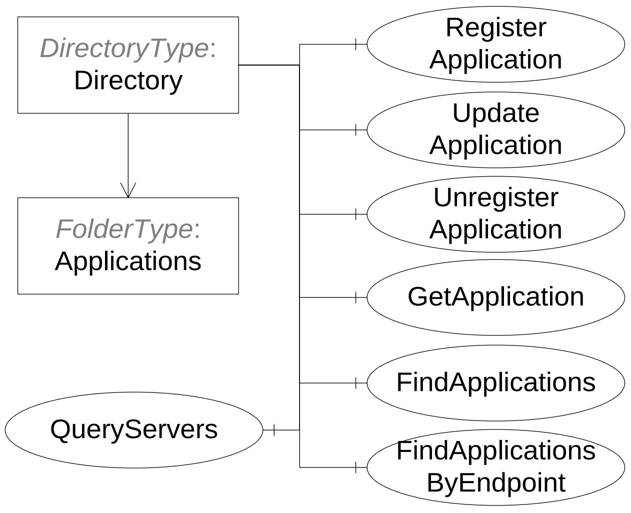

Figure 11 – The Address Space for the GDS

6.3.2 Directory

This *Object* is the root of the *GlobalDiscoveryServer AddressSpace* and it is the target of an *Organizes* reference from the *Objects* folder defined in Part 5. It organizes the information that can be accessed into subfolders. The implementation of a GDS can customize and organize the folders in any manner it desires. For example folders may exist for information models, or for optional services or for various locations in an administrative domain. It is defined in Table 2.

Table 2 – Directory Object Definition

Attribute	**Value**				
BrowseName	Directory				
Namespace	GDS (see 3.3)				
TypeDefinition	DirectoryType defined in 6.3.3.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**

6.3.3 DirectoryType

DirectoryType is the *ObjectType* for the root of the *GlobalDiscoveryServer AddressSpace*. It organizes the information that can be accessed into subfolders It also provides methods that allow applications to register or find applications. It is defined in Table 3.

Table 3 – DirectoryType Definition

Attribute	**Value**
BrowseName	DirectoryType

Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FolderType* defined in Part 5.					
Organizes	Object	Applications	-	FolderType	Mandatory
HasComponent	Method	FindApplications	Defined in 6.3.4.		Mandatory
HasComponent	Method	RegisterApplication	Defined in 6.3.6.		Mandatory
HasComponent	Method	UpdateApplication	Defined in 6.3.7.		Mandatory
HasComponent	Method	UnregisterApplication	Defined in 6.3.8.		Mandatory
HasComponent	Method	GetApplication	Defined in 6.3.9.		Mandatory
HasComponent	Method	QueryApplications	Defined in 6.3.10.		Mandatory
HasComponent	Method	QueryServers	Defined in 6.3.11.		Mandatory

The *Applications* folder may contain *Objects* representing the *Applications* known to the GDS. These *Objects* may be organized into subfolders as determined by the GDS. Some characteristics for organizing applications are the networks, the physical location, or the supported profiles. The *QueryServers Method* can be used to search for OPC UA *Applications* based on various criteria.

A GDS is not required to expose its *Applications* as browsable *Objects* in its *AddressSpace*, however, each *Application* shall have a unique *NodeId* which can be passed to *Methods* used to administer the GDS.

The *FindApplications Method* returns the *Applications* associated with an *ApplicationUri*. It can be called by any *Client* application.

The *RegisterApplication Method* is used to add a new *Application* to the GDS. It requires administrative privileges.

The *UpdateApplication Method* is used to update an existing *Application* in the GDS. It requires administrative privileges.

The *UnregisterApplication Method* is used to remove an *Application* from the GDS. It requires administrative privileges.

The *QueryApplications Method* is used to find *Client* or *Server* applications that meet the criteria provided. This *Method* replaces the *QueryServers Method.*

The *QueryServers Method* is used to find *Servers* that meet the criteria specified. It can be called by any *Client* application. This *Method* has been replaced by the *QueryApplications Method*

6.3.4 FindApplications

FindApplications is used to find the *ApplicationId* for an OPC UA *Application* known to the GDS. In normal situations the list of records returned will not have more than one entry, however, system configuration errors can create situations where the GDS has multiple entries for a single *ApplicationUri*. If this happens a human will likely have to look at records to determine which record is the true match for the *ApplicationUri*.

If the returned array is null or zero length then the GDS does not have an entry for the *ApplicationUri*.

Signature

```
FindApplications(
      [in]  String applicationUri
      [out] ApplicationRecordDataType[] applications
      );
```

Argument	**Description**
applicationUri	The *ApplicationUri* that identifies the *Application* of interest.
applications	A list of application records that match the *ApplicationUri.* The ApplicationRecordDataType is defined in 6.3.5.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 4 specifies the *AddressSpace* representation for the *FindApplications Method*.

Table 4 – FindApplications Method AddressSpace Definition

Attribute	**Value**				
BrowseName	FindApplications				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

6.3.5 ApplicationRecordDataType

This type defines a DataType which represents a record in the GDS.

Table 5 – ApplicationRecordDataType Definition

Name	Type	Value
applicationId	NodeId	The unique identifier assigned by the GDS to the record. This *NodeId* may be passed to other *Methods*.
applicationUri	String	The URI for the *Application* associated with the record.
applicationType	ApplicationType	The type of application. This type is defined in Part 4.
applicationNames	LocalizedText[]	One or more localized names for the application. The first element is the default *ApplicationName* for the application when a non-localized name is needed.
productUri	String	A globally unique URI for the product associated with the application. This URI is assigned by the vendor of the application.
discoveryUrls	String[]	The list of discovery URLs for an application. The first element is the default if a *Client* needs to choose one URL. The first HTTPS URL specifies the domain used as the Common Name of HTTPS *Certificates*. If the *ApplicationType* is *Client* then all of the URLs shall have the 'inv+' prefix which indicates they support reverse connect.
serverCapability Identifiers	String[]	The list of server capability identifiers for the application. The allowed values are defined in Annex D.

6.3.6 RegisterApplication

RegisterApplication is used to register a new *Application* Instance with a *GlobalDiscoveryServer*.

This *Method* shall only be invoked by authorized users.

Servers that support transparent redundancy shall register as a single application and pass the *DiscoveryUrls* for all available instances and/or network paths.

RegisterApplication will create duplicate records if the *ApplicationUri* already exists since misconfiguration of applications can result in different applications having the same *ApplicationUri*. Before calling this *Method* the *Client* shall call *FindApplications* to check if a record for the application it is using already exists. If records are found which appear to belong to different applications (e.g. the *DiscoveryUrls* are different) then the *Client* shall report a warning before continuing.

If registration was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChangedAuditEventType* (see 6.3.12).

Signature

```
RegisterApplication(
   [in]  ApplicationRecordDataType application
      [out] NodeId applicationId
);
```

Argument	Description

application	The application that is to be registered with the *GlobalDiscoveryServer*.
applicationId	A unique identifier for the registered *Application*. This identifier is persistent and is used in other *Methods* used to administer applications.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The application or one of the fields of the application record is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 6 specifies the *AddressSpace* representation for the *RegisterApplication Method*.

Table 6 – RegisterApplication Method AddressSpace Definition

Attribute	**Value**				
BrowseName	RegisterApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

6.3.7 UpdateApplication

UpdateApplication is used to update an existing *Application* in a *GlobalDiscoveryServer*.

This *Method* shall only be invoked by authorized users.

If the update was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChangedAuditEventType* (see 6.3.12).

Signature

```
UpdateApplication(
        [in]  ApplicationRecordDataType application
       );
```

Argument	Description
application	The application that is to be updated in the GDS database.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The applicationId is not known to the GDS.
Bad_InvalidArgument	The application or one of the fields of the application record is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 7 specifies the *AddressSpace* representation for the *UpdateApplication Method*.

Table 7 – UpdateApplication Method AddressSpace Definition

Attribute	**Value**				
BrowseName	UpdateApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.3.8 UnregisterApplication

UnregisterApplication is used to remove an *Application* from a *GlobalDiscoveryServer*.

This *Method* shall only be invoked by authorized users.

A *Server Application* that is unregistered may be automatically added again if the GDS is configured to populate itself by calling *FindServersOnNetwork* and the *Server Application* is still registering with its local LDS.

If un-registration was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChangedAuditEventType* (see 6.3.12).

Signature

```
UnregisterApplication(
      [in] NodeId applicationId
     );
```

Argument	Description
applicationId	The identifier assigned by the GDS to the *Application*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *ApplicationId* is not known to the GDS.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 8 specifies the *AddressSpace* representation for the *UnregisterApplication Method*.

Table 8 – UnregisterApplication Method AddressSpace Definition

Attribute	**Value**				
BrowseName	UnregisterApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.3.9 GetApplication

GetApplication is used to find an OPC UA *Application* known to the GDS.

Signature

```
GetApplication(
     [in]  NodeId applicationId
     [out] ApplicationRecordDataType application
     );
```

Argument	Description
applicationId	The *ApplicationId* that identifies the *Application* of interest.
application	The application record that matches the *ApplicationId.* The ApplicationRecordDataType is defined in6.3.5

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The no record found for the specified *ApplicationId*.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 9 specifies the *AddressSpace* representation for the *GetApplication Method*.

Table 9 – GetApplication Method AddressSpace Definition

Attribute	**Value**				
BrowseName	GetApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

6.3.10 QueryApplications

QueryApplications is used to find *Client* or *Server* applications that meet the specified filters. The only Clients returns are those that support the reverse connection capability described in Part 6.

QueryApplications returns *ApplicationDescriptions* instead of the *ServerOnNetwork Structures* returned by *QueryServers*. This is more useful to some *Clients* because it matches the return type of *FindServers*.

Any *Client* is able to call this *Method*, however, the set of results returned may be restricted based on the *Client's* user credentials.

The applications returned shall pass all of the filters provided (i.e. the filters are combined in an AND operation). The c*apabilities* parameter is an array and an application will pass this filter if it supports all of the specified capabilities.

Each time the GDS creates or updates an application record 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 *QueryApplications*. To support this the GDS shall return records in order starting from the lowest record identifier. The GDS shall also return the last time the counter was reset. If a *Client* detects that this time is more recent than the last time the *Client* called the *Method* it shall call the *Method* again with a *startingRecordId* of 0.

Signature

```
QueryApplications(
      [in]  UInt32 startingRecordId
      [in]  UInt32 maxRecordsToReturn
      [in]  String applicationName
      [in]  String applicationUri
      [in]  UInt32 applicationType
      [in]  String productUri
      [in]  String[] capabilities
      [out] DateTime lastCounterResetTime
      [out] UInt32 nextRecordId
      [out] ApplicationDescription[] applications
     );
```

Argument	Description
INPUTS	
startingRecordId	Only records with an identifier greater than this number will be returned. Specify 0 to start with the first record in the database.
maxRecordsToReturn	The maximum number of records to return in the response. 0 indicates that there is no limit.
applicationName	The *ApplicationName* of the applications to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified. The filter is only applied to the default *ApplicationName*.
applicationUri	The *ApplicationUri* of the applications to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified.
applicationType	A mask indicating what types of applications are returned. The mask values are: 0x1 – Servers; 0x2 – Clients; : If the mask is 0 then all applications are returned.
productUri	The *ProductUri* of the applications to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified.
capabilities	The capabilities supported by the applications returned. The applications returned shall support all of the capabilities specified. If no capabilities are provided this filter is not used.
OUTPUTS	
lastCounterResetTime	The last time the counters were reset.
nextRecordId	The identifier of the next record. It is passed as the *startingRecordId* in subsequent calls to *QueryApplications* to fetch the next batch of records. It is 0 if there are no more records to return.
applications	A list of *Applications* which meet the criteria. The *ApplicationDescription* structure is defined in Part 4.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 11 specifies the *AddressSpace* representation for the *QueryApplications Method*.

Table 10 – QueryApplications Method AddressSpace Definition

Attribute	**Value**				
BrowseName	QueryApplications				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

6.3.11 QueryServers (depreciated)

QueryServers is used to find *Server* applications that meet the specified filters.

Any *Client* is able to call this *Method*, however, the set of results returned may be restricted based on the *Client's* user credentials.

The applications returned shall pass all of the filters provided (i.e. the filters are combined in an AND operation). The *serverCapabilities* parameter is an array and an application will pass this filter if it supports all of the specified capabilities.

Each time the GDS creates or updates an application record 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 *QueryServers*. To support this the GDS shall return records in order starting from the lowest record identifier. The GDS shall also return the last time the counter was reset. If a *Client* detects that this time is more recent than the last time the *Client* called the *Method* it shall call the *Method* again with a *startingRecordId* of 0.

Signature

```
QueryServers(
    [in]  UInt32 startingRecordId
    [in]  UInt32 maxRecordsToReturn
    [in]  String applicationName
    [in]  String applicationUri
    [in]  String productUri
    [in]  String[] serverCapabilities
    [out] DateTime lastCounterResetTime
    [out] ServerOnNetwork[] servers
   );
```

Argument	**Description**
INPUTS	
startingRecordId	Only records with an identifier greater than this number will be returned. Specify 0 to start with the first record in the database.
maxRecordsToReturn	The maximum number of records to return in the response. 0 indicates that there is no limit.
applicationName	The *ApplicationName* of the *Applications* to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified. The filter is only applied to the default *ApplicationName*.
applicationUri	The *ApplicationUri* of the *Servers* to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified.
productUri	The *ProductUri* of the *Servers* to return. Supports the syntax used by the LIKE *FilterOperator* described in Part 4. Not used if an empty string is specified.
serverCapabilities	The applications returned shall support all of the server capabilities specified. If no server capabilities are provided this filter is not used.
OUTPUTS	
lastCounterResetTime	The last time the counters were reset.
servers	A list of *Servers* which meet the criteria. The *ServerOnNetwork* structure is defined in Part 4.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 11 specifies the *AddressSpace* representation for the *QueryServers Method*.

Table 11 – QueryServers Method AddressSpace Definition

Attribute	**Value**				
BrowseName	QueryServers				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

6.3.12 ApplicationRegistrationChangedAuditEventType

This event is raised when the *RegisterApplication*, *UpdateApplication* or *UnregisterApplication Methods* are called.

Its representation in the *AddressSpace* is formally defined in Table 12.

Table 12 – ApplicationRegistrationChangedAuditEventType Definition

Attribute	**Value**				
BrowseName	ApplicationRegistrationChangedAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantics are defined in Part 5.

7 Certificate Management Overview

7.1 Overview

Certificate management functions comprise the management and distribution of certificates and *Trust Lists* for OPC UA Applications. An application that provides the certificate management functions is called *CertificateManager*. GDS and *CertificateManager* will typically be combined in one application. The basic concepts regarding *Certificate* management are described in Part 2.

There are two primary models for *Certificate* management: pull and push management. In pull management, the application acts as a *Client* and uses the *Methods* on the *CertificateManager* to request and update *Certificates* and *Trust Lists*. The application is responsible for ensuring the *Certificates* and *Trust Lists* are kept up to date. In push management the application acts as a *Server* and exposes *Methods* which the *CertificateManager* can call to update the *Certificates* and *Trust Lists* as required.

The GDS is intended to work in conjunction with different Certificate Management services such as Active Directory. The GDS provides a standard OPC UA based information model that all OPC UA applications can support without needing to know the specifics of a particular Certificate Management system.

The *CertificateManager* shall support the following use cases:

- Provisioning (First time setup for a device/application);
- Renewal (Renewing expired or compromised certificates);
- Trust List Update (Updating the Trust Lists including the Revocation Lists);

- Revocation (Removing a device/application from the system).

Although it is generally assumed that Client applications will use the Pull model and Server applications will use the Push model, this is not required.

During provisioning, the *CertificateManager* shall be able to operate in a mode where any *Client* is allowed to connect securely with any valid *Certificate* and user credentials are used to determine the rights a *Client* has; this eliminates the need to configure *Trust Lists* before connecting to the *CertificateManager* for provisioning.

Application vendors may decide to build the interaction with the *CertificateManager* as a separate component, e.g. as part of an administration application with access to the OPC UA configuration of this *Application*. This is transparent for the *CertificateManager* and will not be considered further in the rest of this chapter.

This standard does not define how to administer a *CertificateManager* but a *CertificateManager* shall provide an integrated system that includes both push and pull management.

7.2 Pull Management

Pull Management is performed by using the *CertificateManager* information model – in particular the Methods - defined in 7.6. The interactions between *Application* and *CertificateManager* during Pull Management are illustrated in Figure 12.

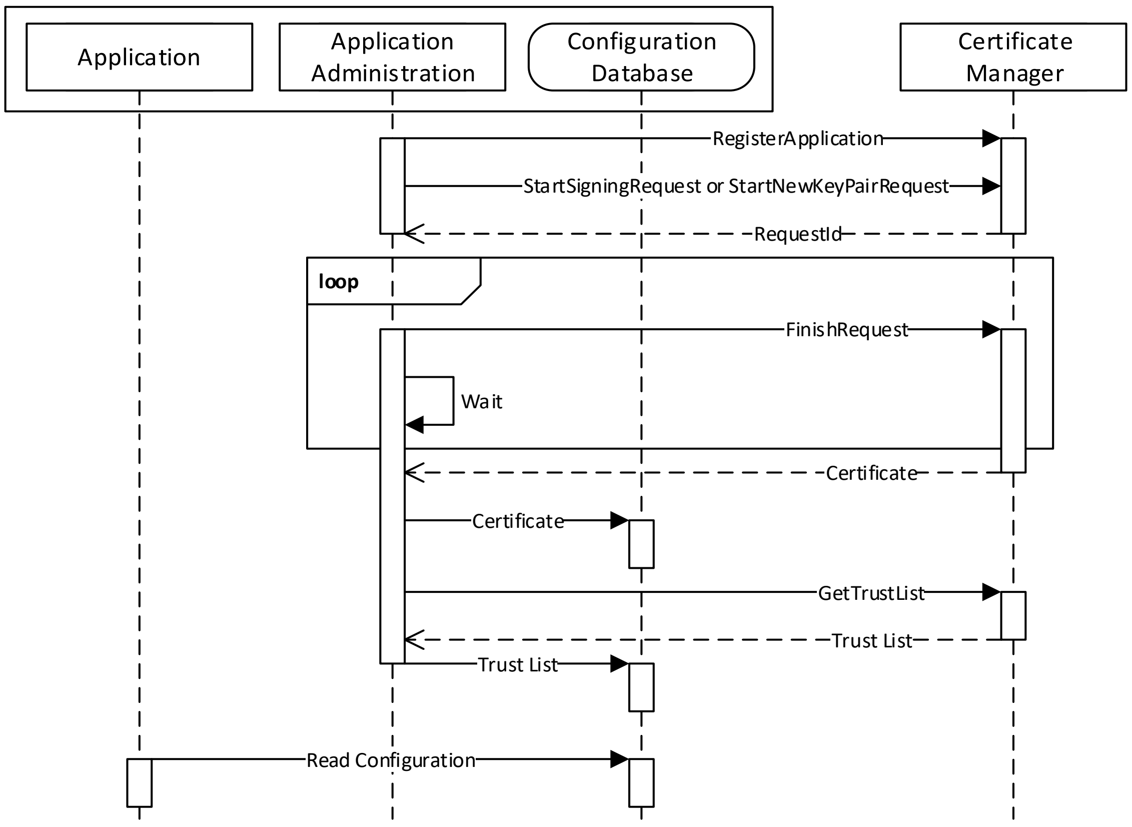

Figure 12 – The Pull Certificate Management Model

The Application Administration component may be part of the Application or a standalone utility that understands how the Application persists its configuration information in its Configuration Database.

A similar process is used to renew certificates or to periodically update *Trust List*.

Security in Pull management requires an encrypted channel and the use of *Administrator* credentials for the *CertificateManager* that ensure only authorized users can register new *Applications* and request an initial new *Certificate*. Once an *Application* has a *Certificate* it can use this *Certificate* to renew the *Certificate* or to update *Trust Lists* and *Revocation* lists. It is important that a *CertificateManager* does not provide certificate renewals except to the applications that already own the prior certificate.

7.3 Push Management

Push management is targeted at *Server* applications and relies on *Methods* defined in 7.7 to get a *Certificate Request* which can be passed onto the *CertificateManager.* After the *CertificateManager* signs the *Certificate* the new *Certificate* is pushed to the *Server* with the *UpdateCertificate Method*.

The interactions between a *Server Application* and *CertificateManager* during Push Management are illustrated in Figure 13.

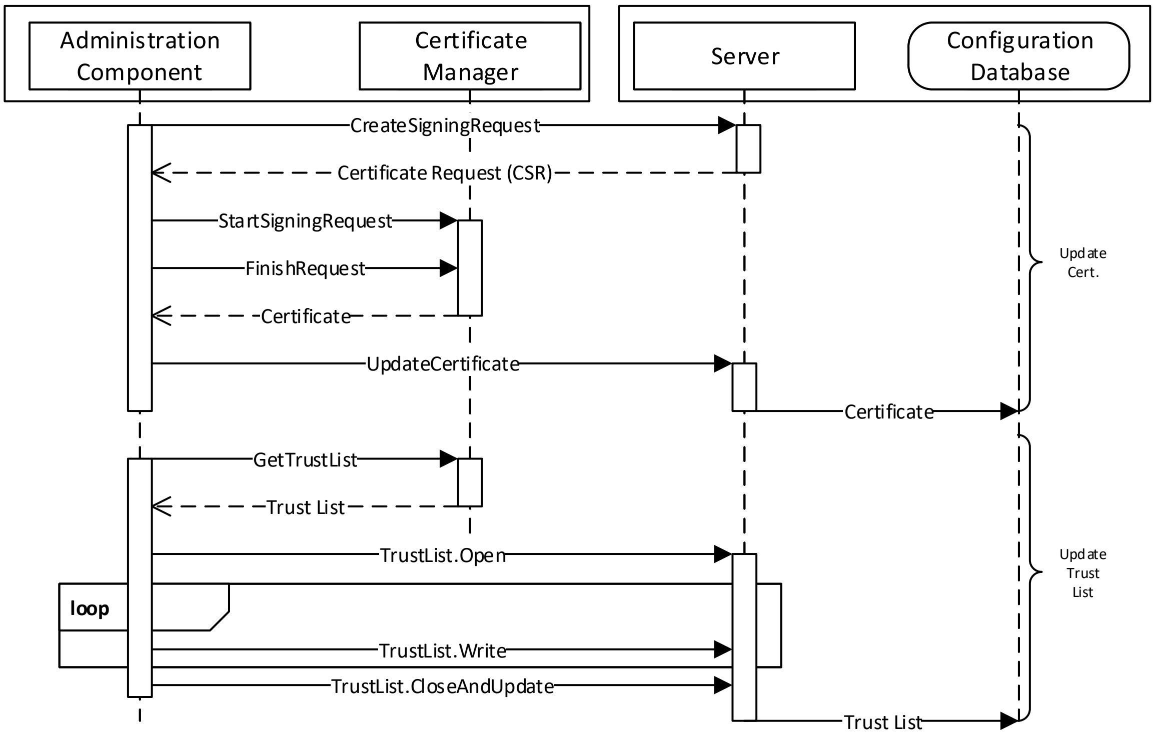

Figure 13 – The Push Certificate Management Model

The Administration Component may be part of the *CertificateManager* or a standalone utility that uses OPC UA to communicate with the *CertificateManager* (see 7.2 for a more complete description of the interactions required for this use case). The Configuration Database is used by the *Server* to persist its configuration information. The *RegisterApplication Method* (or internal equivalent) is assumed to have been called before the sequence in the diagram starts.

A similar process is used to renew certificates or to periodically update *Trust List*.

Security when using the Push Management Model requires an encrypted channel and the use of Administrator credentials for the *Server* that ensure only authorized users can update *Certificates* or *Trust Lists*. In addition, separate *Administrator* credentials are required for the

CertificateManager that ensure only authorized users can register new *Servers* and request new *Certificates*.

7.4 Provisioning

Provisioning is the initial installation of an OPC UA *Server* or *Client* into a system in which a GDS is available and managing all certificates. For applications using *Client* interface provisioning can be accomplished using a pull model. Applications using the *Server* interface can be provisioned using the push model.

OPC UA *Servers* will typically auto-generate a self-signed *Certificate* when they first start. They may also have a pre-configured *Trust List* with *Applications* that are allowed to provision the *Server*. For example, a device vendor may use a CA that is used to issue *Certificates* to *Applications* used by their field technicians.

For embedded devices, the *Server* should allow any *Client* that provides the proper *Administrator* credentials to create the secure connection needed for provisioning using push management. Once the device has been given its initial *Trust List* the *Server* should then restrict access to those *Clients* with *Certificates* in the *Trust List*. A vendor specific process for provisioning is required if a device does not allow any *Client* to connect securely for provisioning.

See G.1 for more specific examples of how to provision an application.

7.5 Common Information Model

7.5.1 Overview

The common information model defines types that are used in both the Push and the Pull Model.

7.5.2 TrustListType

This type defines a *FileType* that can be used to access a *Trust List*.

The *CertificateManager* uses this type to implement the Pull Model.

Servers use this type when implementing the Push Model.

An instance of a *TrustListType* shall restrict access to appropriate users or applications. This may be a *CertificateManager* administrative user that can change the contents of a *Trust List*, it may be an Administrative user that is reading a *Trust List* to deploy to an Application host or it may be an Application that can only access the Trust List assigned to it.

The *Trust List* file is a UA Binary encoded stream containing an instance of *TrustListDataType* (see 7.5.7).

The *Open Method* shall not support modes other than Read (0x01) and the Write + EraseExisting (0x06).

When a *Client* opens the file for writing the *Server* will not actually update the *Trust List* until the *CloseAndUpdate Method* is called. Simply calling *Close* will discard the updates. The bit masks in *TrustListDataType* structure allow the *Client* to only update part of the *Trust List*.

When the *CloseAndUpdate Method* is called the *Server* will validate all new *Certificates* and *CRLs.* If this validation fails the *Trust List* is not updated and the *Server* returns the appropriate *Certificate* error code (see Part 4).

Table 13 – TrustListType Definition

Attribute	**Value**				
BrowseName	TrustListType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FileType* defined in Part 5.					
HasProperty	Variable	LastUpdateTime	UtcTime	PropertyType	Mandatory

HasProperty	Variable	UpdateFrequency	Duration	PropertyType	Optional
HasComponent	Method	OpenWithMasks	Defined in 7.5.3.		Optional
HasComponent	Method	CloseAndUpdate	Defined in 7.5.4.		Optional
HasComponent	Method	AddCertificate	Defined in 7.5.5.		Optional
HasComponent	Method	RemoveCertificate	Defined in 7.5.6.		Optional

The *LastUpdateTime* indicates when the *Trust List* was last updated via *Trust List Object Methods*. This can be used to determine if a device has an up to date *Trust List* or to detect unexpected modifications. Out of band changes are not necessarily reported by this value.

The *UpdateFrequency Property* specifies how often the *Trust List* needs to be checked for changes. When the *CertificateManager* specifies this value, all *Clients* that read a copy of the *Trust List* should connect to the *CertificateManager* and check for updates to the *Trust List* within 2 times the *UpdateFrequency*. If the *Trust List Object* is contained within a *ServerConfiguration Object* then this value specifies how frequently the *Server* expects the *Trust List* to be updated.

If auditing is supported, the CertificateManager shall generate the *TrustListUpdatedAuditEventType* (see 7.5.18) if the *CloseAndUpdate*, *AddCertificate* or *RemoveCertificate Methods* are called.

7.5.3 OpenWithMasks

The *OpenWithMasks Method* allows a *Client* to read only the portion of the *Trust List.*

This *Method* can only be used to read the *Trust List*.

Signature

```
OpenWithMasks(
    [in]  UInt32 masks
    [out] UInt32 fileHandle
   );
```

Argument	Description
masks	The parts of the *Trust List* that are include in the file to read. The masks are defined in 7.5.8.
fileHandle	The handle of the newly opened file.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 14 specifies the *AddressSpace* representation for the *OpenWithMasks Method*.

Table 14 – OpenWithMasks Method AddressSpace Definition

Attribute	Value				
BrowseName	OpenWithMasks				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.5.4 CloseAndUpdate

The *CloseAndUpdate Method* closes the file and applies the changes to the *Trust List*. It can only be called if the file was opened for writing. If the *Close Method* is called any cached data is discarded and the *Trust List* is not changed.

The *Server* shall verify that every *Certificate* in the new *Trust List* is valid according to the mandatory rules defined in Part 4. If an invalid *Certificate* is found the *Server* shall return an error and shall not update the *Trust List*. If only part of the *Trust List* is being updated the

Server creates a temporary *Trust List* that includes the existing *Trust List* plus any updates and validates the temporary *Trust List*.

If the file cannot be processed this *Method* still closes the file and discards the data before returning an error. This *Method* is required if the *Server* supports updates to the *Trust List*.

The structure uploaded includes a mask (see 7.5.8) which specifies which fields are updated. If a bit is not set then the associated field is not changed.

Signature

```
CloseAndUpdate(
       [in]  UInt32 fileHandle
       [out] Boolean applyChangesRequired
      );
```

Argument	Description
fileHandle	The handle of the previously opened file.
applyChangesRequired	A flag indicating whether the *ApplyChanges* Method (see 7.7.5) shall be called before the new *Trust List* will be used by the *Server*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_CertificateInvalid	The Server could not validate all Certificates in the Trust List. The DiagnosticInfo shall specify which Certificate(s) are invalid and the specific error.

Table 15 specifies the *AddressSpace* representation for the *CloseAndUpdate Method*.

Table 15 – CloseAndUpdate Method AddressSpace Definition

Attribute	Value				
BrowseName	CloseAndUpdate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.5.5 AddCertificate

The *AddCertificate Method* allows a *Client* to add a single *Certificate* to the *Trust List*. The *Server* shall verify that the *Certificate* is valid according to the rules defined in Part 4. If an invalid *Certificate* is found the *Server* shall return an error and shall not update the *Trust List*.

If the *Certificate* is issued by a CA then the *Client* shall provide the entire chain in the *certificate* argument (see Part 6). After validating the *Certificate*, the *Server* shall add the CA *Certificates* to the *Issuers* list in the *Trust List*. The leaf *Certificate* is added to the list specified by the *isTrustedCertificate* argument.

This method cannot be called if the file object is open.

```
AddCertificate(
       [in] ByteString certificate
       [in] Boolean isTrustedCertificate
      );
```

Argument	Description
Certificate	The DER encoded Certificate to add.
isTrustedCertificate	If TRUE the Certificate is added to the Trusted Certificates List. If FALSE the Certificate is added to the Issuer Certificates List.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_CertificateInvalid	The certificate to add is invalid.
Bad_InvalidState	The object is opened.

Table 16 specifies the *AddressSpace* representation for the *AddCertificate Method*.

Table 16 – AddCertificate Method AddressSpace Definition

Attribute	**Value**				
BrowseName	AddCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

7.5.6 RemoveCertificate

The *RemoveCertificate Method* allows a *Client* to remove a single *Certificate* from the *Trust List*. It returns *Bad_InvalidArgument* if the thumbprint does not match a Certificate in the *Trust List*.

If the *Certificate* is a CA *Certificate* with associated CRLs then all CRLs are removed as well.

This method cannot be called if the file object is open.

```
RemoveCertificate(
      [in] String thumbprint
      [in] Boolean isTrustedCertificate
     );
```

Argument	Description
Thumbprint	The SHA1 hash of the *Certificate* to remove.
isTrustedCertificate	If TRUE the Certificate is removed from the Trusted Certificates List. If FALSE the Certificate is removed from the Issuer Certificates List.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_InvalidArgument	The certificate to remove was not found.
Bad_InvalidState	The object is opened.

Table 17 specifies the *AddressSpace* representation for the *RemoveCertificate Method*.

Table 17 – RemoveCertificate Method AddressSpace Definition

Attribute	**Value**				
BrowseName	RemoveCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

7.5.7 TrustListDataType

This type defines a DataType which stores the Trust List of a *Server.* Its values are defined in Table 18.

Table 18 – TrustListDataType Definition

Name	Type	Value
TrustListDataType	structure	
specifiedLists	TrustListMasks	A bit mask which indicates which lists contain information. The *TrustListMasks* enumeration in 7.5.8 defines the allowed values.
trustedCertificates	ByteString[]	The list of *Application* and CA *Certificates* which are trusted.
trustedCrls	ByteString[]	The CRLs for the *Certificates* in the *trustedCertificates* list.
issuerCertificates	ByteString[]	The list of CA *Certificates* which are necessary to validate *Certificates*.
issuerCrls	ByteString[]	The CRLs for the CA *Certificates* in the *issuerCertificates* list.

7.5.8 TrustListMasks

This is a DataType that defines the values used for the SpecifiedLists field in the *TrustListDataType*. Its values are defined in Table 19.

Table 19 – TrustListMasks Values

Value	Value
None_0	No fields are provided.
TrustedCertificates_1	The TrustedCertificates are provided.
TrustedCrls_2	The TrustedCrls are provided.
IssuerCertificates_4	The IssuerCertificates are provided.
IssuerCrls_8	The IssuerCrls are provided.
All_15	All fields are provided.

7.5.9 TrustListOutOfDateAlarmType

This *SystemOffNormalAlarmType* is raised by the *Server* when the *UpdateFrequency* elapses and the *Trust List* has not been updated. This alarm automatically returns to normal when the *Trust List* is updated.

Table 20 – TrustListOutOfDateAlarmType definition

Attribute	Value				
BrowseName	TrustListOutOfDateAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the SystemOffNormalAlarmType defined in Part 9.					
HasProperty	Variable	TrustListId	NodeId	PropertyType	Mandatory
HasProperty	Variable	LastUpdateTime	UtcTime	PropertyType	Mandatory
HasProperty	Variable	UpdateFrequency	Duration	PropertyType	Mandatory

TrustListId Property specifies the *NodeId* of the out of date *Trust List Object*.

LastUpdateTime Property specifies when the *Trust List* was last updated.

UpdateFrequency Property specifies how frequently the *Trust List* needs to be updated.

7.5.10 CertificateGroupType

This type is used for *Objects* which represent *Certificate Groups* in the *AddressSpace*. A *Certificate Group* is a context that contains a *Trust List* and one or more *Certificates* that can be assigned to an *Application*. This type exists to allow an *Application* which has multiple *Trust Lists* and/or *Application Certificates* to express them in its *AddressSpace*. This type is defined in Table 21.

Table 21 – CertificateGroupType Definition

Attribute	Value				
BrowseName	CertificateGroupType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasComponent	Object	TrustList	-	TrustListType	Mandatory
HasProperty	Variable	CertificateTypes	NodeId[]	PropertyType	Mandatory
HasComponent	Object	CertificateExpired		CertificateExpirationAlarmType	Optional
HasComponent	Object	TrustListOutOfDate		TrustListOutOfDateAlarmType	Optional

The *TrustList Object* is the *Trust List* associated with the *Certificate Group*.

The *CertificateTypes Property* specifies the *NodeIds* of the *CertificateTypes* which may be assigned to *Applications* which belong to the *Certificate Group*. For example, a *Certificate*

Group with the *NodeId* of *RsaMinApplicationCertificateType* (see 7.5.15) and the *NodeId RsaSha256ApplicationCertificate* (see 7.5.16) specified allows an *Application* to have one *Application Instance Certificates* for each type. Abstract base types may be used in this value and indicate that any subtype is allowed. If this list is empty then the *Certificate Group* does not allow *Certificates* to be assigned to *Applications* (i.e. the *Certificate Group* exists to allow the associated *Trust List* to be read or updated). All *CertificateTypes* for a given *Certificate Group* shall be subtypes of a single common type which shall be either *ApplicationCertificateType* or *HttpsCertificateType*.

The *CertificateExpired Object* is an *Alarm* which is raised when the *Certificate* associated with the *CertificateGroup* is about to expire. The *CertificateExpirationAlarmType* is defined in Part 9.

The *TrustListOutOfDate Object* is an Alarm which is raised when the *Trust List* has not been updated within the period specified by the *UpdateFrequency* (see 7.5.2). The *TrustListOutOfDateAlarmType* is defined in 7.5.9.

7.5.11 CertificateType

This type is an abstract base type for types that describe the purpose of a *Certificate*. This type is defined in Table 22.

Table 22 – CertificateType Definition

Attribute	**Value**				
BrowseName	CertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasSubtype	ObjectType	ApplicationCertificateType	Defined in 7.5.12.		
HasSubtype	ObjectType	HttpsCertificateType	Defined in 7.5.13.		
HasSubtype	ObjectType	UserCredentialCertificateType	Defined in 7.5.14.		

7.5.12 ApplicationCertificateType

This type is an abstract base type for types that describe the purpose of an *ApplicationInstanceCertificate*. This type is defined in Table 23.

Table 23 – ApplicationCertificateType Definition

Attribute	**Value**				
BrowseName	ApplicationCertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *CertificateType* defined in 7.5.11.					
HasSubtype	ObjectType	RsaMinApplicationCertificateType	Defined in 7.5.15.		
HasSubtype	ObjectType	RsaSha256ApplicationCertificateType	Defined in 7.5.16.		

7.5.13 HttpsCertificateType

This type is used to describe Certificates that are intended for use as HTTPS *Certificates*. This type is defined in Table 24.

Table 24 – HttpsCertificateType Definition

Attribute	**Value**				
BrowseName	HttpsCertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *CertificateType* defined in 7.5.11.					

7.5.14 UserCredentialCertificateType

This type is used to describe Certificates that are intended for use as user credentials. This type is defined in Table 25.

Table 25 – UserCredentialCertificateType Definition

Attribute	Value				
BrowseName	UserCredentialCertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *CertificateType* defined in 7.5.11.					

7.5.15 RsaMinApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an RSA key size of 1024 or 2048 bits. All *Applications* which support the *Basic128Rsa15* and *Basic256* profiles (see Part 7) shall have a *Certificate* of this type. This type is defined in Table 26.

Table 26 – RsaMinApplicationCertificateType Definition

Attribute	Value				
BrowseName	RsaMinApplicationCertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *ApplicationCertificateType* defined in 7.5.12					

7.5.16 RsaSha256ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an RSA key size of 2048, 3072 or 4096 bits. All *Applications* which support the *Basic256Sha256* profile (see Part 7) shall have a *Certificate* of this type. This type is defined in Table 27.

Table 27 – RsaSha256ApplicationCertificateType Definition

Attribute	Value				
BrowseName	RsaSha256ApplicationCertificateType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *ApplicationCertificateType* defined in 7.5.12					

7.5.17 CertificateGroupFolderType

This type is used for *Folders* which organize *Certificate Groups* in the *AddressSpace*. This type is defined in Table 21.

Table 28 – CertificateGroupFolderType Definition

Attribute	Value				
BrowseName	CertificateGroupFolderType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FolderType* defined in Part 5.					
Organizes	Object	DefaultApplicationGroup		CertificateGroupType	Mandatory
Organizes	Object	DefaultHttpsGroup		CertificateGroupType	Optional
Organizes	Object	DefaultUserTokenGroup		CertificateGroupType	Optional
Organizes	Object	<AdditionalGroup>		CertificateGroupType	Optional Placeholder

The *DefaultApplicationGroup Object* represents the default *Certificate Group* for *Applications*. It is used to access the default *Application Trust List* and to define the *CertificateTypes* allowed for the *ApplicationInstanceCertificate*. This *Object* shall specify the *ApplicationCertificateType NodeId* (see 7.5.12) as a single entry in the *CertificateTypes* list or it shall specify one or more subtypes of *ApplicationCertificateType*.

The *DefaultHttpsGroup Object* represents the default *Certificate Group* for HTTPS communication. It is used to access the default HTTPS *Trust List* and to define the *CertificateTypes* allowed for the *HTTPS Certificate*. This *Object* shall specify the *HttpsCertificateType NodeId* (see 7.5.13) as a single entry in the *CertificateTypes* list or it shall specify one or more subtypes of *HttpsCertificateType*.

This *DefaultUserTokenGroup Object* represents the default *Certificate Group* for validating user credentials. It is used to access the default user credential *Trust List* and to define the *CertificateTypes* allowed for user credentials *Certificate*. This *Object* shall leave *CertificateTypes* list empty.

7.5.18 TrustListUpdatedAuditEventType

This event is raised when a *Trust List* is changed.

This is the result of a *CloseAndUpdate Method* on a *TrustListType Object* being called.

It shall also be raised when the *AddCertificate* or *RemoveCertificate* Method causes an update to the *Trust List*.

Its representation in the *AddressSpace* is formally defined in Table 29.

Table 29 – TrustListUpdatedAuditEventType Definition

Attribute	**Value**				
BrowseName	TrustListUpdatedAuditEventType				
Namespace	CORE (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

7.6 Information Model for Pull Certificate Management

7.6.1 Overview

The *GlobalDiscoveryServer AddressSpace* used for *Certificate* management is shown in Figure 14. Most of the interactions between the *GlobalDiscoveryServer* and *Application* administrator or the *Client* will be via *Methods* defined on the *Directory* folder.

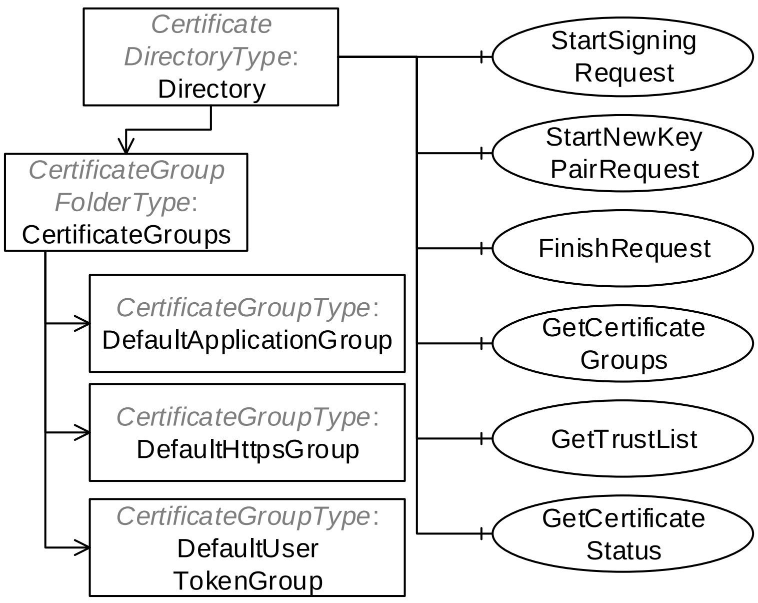
CertificateDirectoryType: Directory
CertificateGroupFolderType: CertificateGroups
CertificateGroupType: DefaultApplicationGroup
CertificateGroupType: DefaultHttpsGroup
CertificateGroupType: DefaultUserTokenGroup
StartSigningRequest
StartNewKeyPairRequest
FinishRequest
GetCertificateGroups
GetTrustList
GetCertificateStatus

Figure 14 – The Certificate Management AddressSpace for the GlobalDiscoveryServer

7.6.2 CertificateDirectoryType

This *ObjectType* is the *TypeDefinition* for the root of the *CertificateManager AddressSpace*. It provides additional *Methods* for *Certificate* management which are shown in Table 30.

Table 30 – CertificateDirectoryType ObjectType Definition

Attribute	**Value**				
BrowseName	CertificateDirectoryType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *DirectoryType* defined in 6.3.3.					
Organizes	Object	CertificateGroups		CertificateGroup FolderType	Mandatory
HasComponent	Method	StartSigningRequest	Defined in 7.6.3.		Mandatory
HasComponent	Method	StartNewKeyPairRequest	Defined in 7.6.4.		Mandatory
HasComponent	Method	FinishRequest	Defined in 7.6.5.		Mandatory
HasComponent	Method	GetCertificateGroups	Defined in 7.6.6.		Mandatory
HasComponent	Method	GetTrustList	Defined in 7.6.6.		Mandatory
HasComponent	Method	GetCertificateStatus	Defined in 7.6.8.		Mandatory

The *CertificateGroups Object* organizes the *Certificate Groups* supported by the *CertificateManager*. It is described in 7.5.17. *CertificateManagers* shall support the *DefaultApplicationGroup* and may support the *DefaultHttpsGroup* or the *DefaultUserTokenGroup*. *CertificateManagers* may support additional *Certificate Groups* depending on their requirements. For example, a *CertificateManager* with multiple Certificate Authorities would represent each as a *CertificateGroupType* Object organized by *CertificateGroups Folder. Clients* could then request *Certificates* issued by a specific CA by passing the appropriate *NodeId* to the *StartSigningRequest* or *StartNewKeyPairRequest Methods*.

The *StartSigningRequest Method* is used to request a new a *Certificate* that is signed by a CA managed by the *CertificateManager*. This *Method* is recommended when the caller already has a private key.

The *StartNewKeyPairRequest Method* is used to request a new *Certificate* that is signed by a CA managed by the *CertificateManager* along with a new private key. This *Method* is used only when the caller does not have a private key and cannot generate one.

The *FinishRequest Method* is used to check that a *Certificate* request has been approved by the *CertificateManager Administrator*. If successful the *Certificate* and *Private Key* (if requested) are returned.

The *GetCertificateGroups Method* returns a list of *NodeIds* for *CertificateGroupType Objects* that can be used to request *Certificates* or *Trust Lists* for an *Application*.

The *GetTrustList Method* returns a *NodeId* of a *TrustListType Object* that can be used to read a *Trust List* for an *Application*.

The *GetCertificateStatus Method* checks whether the *Application* needs to update its *Certificate*.

7.6.3 StartSigningRequest

StartSigningRequest is used to initiate a request to create a *Certificate* which uses the private key which the caller currently has. The new *Certificate* is returned in the *FinishRequest* response.

Signature

```
StartSigningRequest(
      [in]  NodeId applicationId
      [in]  NodeId certificateGroupId
      [in]  NodeId certificateTypeId
      [in]  ByteString certificateRequest
      [out] NodeId requestId
     );
```

Argument	Description
applicationId	The identifier assigned to the *Application* record by the *CertificateManager*.
certificateGroupId	The *NodeId* of the Certificate Group which provides the context for the new request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*.
certificateTypeId	The *NodeId* of the *CertificateType* for the new *Certificate*. If null the *CertificateManager* shall generate a *Certificate* based on the value of the certificateGroupId argument.
certificateRequest	A *CertificateRequest* used to prove possession of the *Private Key*. It is a PKCS #10 encoded blob in DER format. This blob shall include the *subjectAltName* extension that is in the *Certificate.*
requestId	The *NodeId* that represents the request. This value is passed to *FinishRequest* .

The call returns the *NodeId* that is passed to the *FinishRequest Method*.

The *certificateGroupId* parameter allows the caller to specify a *Certificate Group* that provides context for the request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*. The set of available *Certificate Groups* are found in the *CertificateGroups* folder described in 7.6.2. The *Certificate Groups* allowed for an *Application* are returned by the *GetCertificateGroups Method* (see 7.6.6).

The *certificateTypeId* parameter specifies the type of *Certificate* to return. The permitted values are specified by the CertificateTypes Property of the Object specified by the certificateGroupId parameter.

The *certificateRequest* parameter is a DER encoded *Certificate Request*. The subject name, subject alternative name and public key are copied into the new *Certificate*.

If the *certificateTypeId* is a subtype of *ApplicationCertificateType* the subject name shall have an organization (O=) or domain name (DC=) field. The public key length shall meet the length restrictions for the *CertificateType.* If the *certificateType* is a subtype of *HttpsCertificateType* the *Certificate* common name (CN=) shall be the same as a domain from a *DiscoveryUrl* which uses HTTPS and the subject name shall have an organization (O=) field. The public key length shall be greater than or equal to 1024 bits.

The *ApplicationUri* shall be specified in the CSR. The *CertificateManager* shall return *Bad_CertificateUriInvalid* if the stored *ApplicationUri* for the Application is different from what is in the CSR.

For *Servers*, the list of domain names shall be specified in the CSR. The domains shall include the domain(s) in the *DiscoveryUrls* known to the *CertificateManager*.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions. It can also be invoked by the *Application* that owns the private key used to sign the *CertificateRequest* (e.g. the private key shall be the private key used to create the *SecureChannel*).

If auditing is supported, the *CertificateManager* shall generate the *CertificateRequestedAuditEventType* (see 7.6.9) if this *Method* succeeds or fails.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The certificateGroupId, *certificateTypeId* or *certificateRequest* is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_RequestNotAllowed	The current configuration of the *CertificateManager* does not allow the request. The text associated with the error should indicate the exact reason.

Bad_CertificateUriInvalid	The ApplicationUri was not specified in the CSR or does not match the Application record.
Bad_NotSupported	The signing algorithm, public algorithm or public key size are not supported by the *CertificateManager.* The text associated with the error shall indicate the exact problem.

Table 31 specifies the *AddressSpace* representation for the *StartSigningRequest Method*.

Table 31 – StartSigningRequest Method AddressSpace Definition

Attribute	**Value**				
BrowseName	StartSigningRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.4 StartNewKeyPairRequest

This *Method* is used to start a request for a new *Certificate* and *Private Key*. The *Certificate* and private key are returned in the *FinishRequest* response.

Signature

```
StartNewKeyPairRequest(
       [in]  NodeId applicationId
       [in]  NodeId certificateGroupId
       [in]  NodeId certificateTypeId
       [in]  String subjectName
       [in]  String[] domainNames
       [in]  String privateKeyFormat
       [in]  String privateKeyPassword
       [out] NodeId requestId
      );
```

Argument	**Description**
applicationId	The identifier assigned to the *Application Instance* by the *CertificateManager*.
certificateGroupId	The *NodeId* of the Certificate Group which provides the context for the new request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*.
certificateTypeId	The *NodeId* of the *CertificateType* for the new *Certificate*. If null the *CertificateManager* shall generate a *Certificate* based on the value of the certificateGroupId argument.
subjectName	The subject name to use for the *Certificate*. If not specified the *ApplicationName* and/or *domainNames* are used to create a suitable default value. The format of the subject name is a sequence of name value pairs separated by a '/'. The name shall be one of 'CN', 'O', 'OU', 'DC', 'L', 'S' or 'C' and shall be followed by a '=' and then followed by the value. The value may be any printable character except for '"'. If the value contains a '/' or a '=' then it shall be enclosed in double quotes ('"').
domainNames	The domain names to include in the *Certificate*. If not specified the *DiscoveryUrls* are used to create suitable defaults.
privateKeyFormat	The format of the private key. The following values are always supported: PFX - PKCS #12 encoded PEM - Base64 encoded DER (see RFC 5958).
privateKeyPassword	The password to use for the private key.
requestId	The *NodeId* that represents the request. This value is passed to *FinishRequest*.

The call returns the *NodeId* that is passed to the *FinishRequest Method*.

The *certificateGroupId* parameter allows the caller to specify a *Certificate Group* that provides context for the request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*. The set of available *Certificate Groups* are found in the *CertificateGroups* folder described in 7.6.2. The *Certificate Groups* allowed for an *Application* are returned by the *GetCertificateGroups Method* (see 7.6.6).

The *certificateTypeId* parameter specifies the type of *Certificate* to return. The permitted values are specified by the *CertificateTypes Property* of the *Object* specified by the certificateGroupId parameter.

The *subjectName* parameter is a sequence of X.500 name value pairs separated by a '/'. For example: CN=ApplicationName/OU=Group/O=Company.

If the *certificateType* is a subtype of *ApplicationCertificateType* the *Certificate* subject name shall have an organization (O=) or domain name (DC=) field. The public key length shall meet the length restrictions for the *CertificateType.* The domain name field specified in the subject name is a logical domain used to qualify the subject name that may or may not be the same as a domain or IP address in the subjectAltName field of the *Certificate*.

If the *certificateType* is a subtype of *HttpsCertificateType* the *Certificate* common name (CN=) shall be the same as a domain from a *DiscoveryUrl* which uses HTTPS and the subject name shall have an organization (O=) field.

If the subjectName is blank or null the *CertificateManager* generates a suitable default.

The *domainNames* parameter is list of domains to be includes in the *Certificate*. If it is null or empty the GDS uses the *DiscoveryUrls* of the *Server* to create a list. For *Clients* the *domainNames* are omitted from the *Certificate* if they are not explicitly provided.

The *privateKeyFormat* specifies the format of the private key returned. All *CertificateManager* implementations shall support "PEM" and "PFX".

The *privateKeyPassword* specifies the password on the private key. The *CertificateManager* shall not persist this information and shall discard it once the new private key is generated.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions.

If auditing is supported, the *CertificateManager* shall generate the *CertificateRequested AuditEventType* (see 7.6.9) if this *Method* succeeds or fails.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NodeIdUnknown	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The certificateGroupId, *certificateTypeId*, *subjectName*, *domainNames or privateKeyFormat* parameter is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_RequestNotAllowed	The current configuration of the CertificateManager does not allow the request. The text associated with the error should indicate the exact reason.

Table 32 specifies the *AddressSpace* representation for the *StartNewKeyPairRequest Method*.

Table 32 – StartNewKeyPairRequest Method AddressSpace Definition

Attribute	**Value**				
BrowseName	StartNewKeyPairRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.5 FinishRequest

FinishRequest is used to finish a certificate request started with a call to *StartNewKeyPairRequest* or *StartSigningRequest*.

Signature

```
FinishRequest (
    [in]  NodeId applicationId
    [in]  NodeId requestId
    [out] ByteString certificate
    [out] ByteString privateKey
    [out] ByteString[] issuerCertificates
   );
```

Argument	Description
applicationId	The identifier assigned to the *Application Instance* by the GDS.
requestId	The *NodeId* returned by *StartNewKeyPairRequest* or *StartSigningRequest*.
certificate	The DER encoded *Certificate*.
privateKey	The private key encoded in the format requested. If a password was supplied the blob is protected with it. This field is null if no private key was requested.
issuerCertificates	The *Certificates* required to validate the new *Certificate*.

This call is passes the *NodeId* returned by a previous call to *StartNewKeyPairRequest* or *StartSigningRequest*.

It is expected that a *Client* will periodically call this *Method* until the GDS has approved the request.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions. It can also be invoked by the *Application* that owns the *Certificate* (e.g. the private key used to create the channel shall be the same as the private key used to sign the request passed to *StartSigningRequest*).

The *Method* shall only be called via a *SecureChannel* with encryption enabled.

If auditing is supported, the GDS shall generate the *CertificateDeliveredAuditEventType* (see 7.6.10) if this *Method* succeeds or if it fails with anything but *Bad_NothingToDo*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The *requestId* is does not reference to a valid request for the *Application*.
Bad_NothingToDo	There is nothing to do because request has not yet completed.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_RequestNotAllowed	The *CertificateManager* rejected the request. The text associated with the error should indicate the exact reason.

Table 33 specifies the *AddressSpace* representation for the *FinishRequest Method*.

Table 33 – FinishRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	FinishRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.6 GetCertificateGroups

GetCertificateGroups returns the Certificate Groups assigned to *Application*.

Signature

```
GetCertificateGroups(
    [in]  NodeId   applicationId
    [out] NodeId[] certificateGroupIds
   );
```

Argument	Description
applicationId	The identifier assigned to the *Application* by the GDS.
certificateGroupIds	An identifier for the Certificate Groups assigned to the Application.

A *Certificate Group* provides a *Trust List* and one or more *CertificateTypes* which may be assigned to an *Application*. The values returned by this Method are passed to the *GetTrustList* (see 7.6.7), *StartSigningRequest* (see 7.6.3) or *StartNewKeyPairRequest* (see 7.6.4) *Methods*.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions. It can also be invoked by the *Application* identified by the *applicationId* (e.g. the private key used to create the channel shall be private key associated with the *Certificate* assigned to the *Application*).

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 35 specifies the *AddressSpace* representation for the *GetCertificateGroups Method*.

Table 34 – GetCertificateGroups Method AddressSpace Definition

Attribute	**Value**				
BrowseName	GetCertificateGroups				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.7 GetTrustList

GetTrustList is used to retrieve the *NodeId* of a *Trust List* assigned to an *Application*.

Signature

```
GetTrustList(
    [in]  NodeId applicationId
    [in]  NodeId certificateGroupId
    [out] NodeId trustListId
   );
```

Argument	Description
applicationId	The identifier assigned to the *Application* by the GDS.
certificateGroupId	An identifier for a *Certificate Group* that the *Application* belongs to.
trustListId	The *NodeId* for a *Trust List Object* that can be used to download the *Trust List* assigned to the *Application*.

Access permissions also apply to the *Trust List Objects* which are returned by this *Method*. This *Trust List* includes any *Certificate Revocation Lists* (CRLs) associated with issuer *Certificates* in the *Trust List*.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions. It can also be invoked by the *Application* identified by the *applicationId* (e.g. the private key used to create the channel shall be private key associated with the *Certificate* assigned to the *Application*).

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The certificateGroupId parameter is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 35 specifies the *AddressSpace* representation for the *GetTrustList Method*.

Table 35 – GetTrustList Method AddressSpace Definition

Attribute	Value				
BrowseName	GetTrustList				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.8 GetCertificateStatus

GetCertificateStatus is used to check if an *Application* needs to update its *Certificate*.

Signature

```
GetCertificateStatus(
      [in]  NodeId applicationId
      [in]  NodeId certificateGroupId

      [in]  NodeId certificateTypeId
      [out] Boolean updateRequired
     );
```

Argument	Description
applicationId	The identifier assigned to the *Application Instance* by the GDS.
certificateGroupId	The *NodeId* of the Certificate Group which provides the context. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*.
certificateTypeId	The *NodeId* of the *CertificateType* for the *Certificate*. If null the *CertificateManager* shall select a *Certificate* based on the value of the certificateGroupId argument.
updateRequired	TRUE if the *Application* needs to request a new *Certificate* from the GDS. FALSE if the *Application* can keep using the existing *Certificate*.

Access permissions that apply to *CreateSigningRequest Method* shall apply to this *Method*.

This *Method* can be invoked by a configuration tool which has provided user credentials with necessary access permissions. It can also be invoked by the *Application* identified by the *applicationId* (e.g. the private key used to create the channel shall be private key associated with the *Certificate* assigned to the *Application*).

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The *certificateGroupId* or *certificateTypeId* parameter is not valid. The text associated with the error shall indicate the exact problem.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 36 specifies the *AddressSpace* representation for the *GetCertificateStatus Method*.

Table 36 – GetCertificateStatus Method AddressSpace Definition

Attribute	Value				
BrowseName	GetCertificateStatus				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.6.9 CertificateRequestedAuditEventType

This event is raised when a new certificate request has been accepted or rejected by the GDS.

This can be the result of a *StartNewKeyPairRequest* or *StartSigningRequest Method* calls.

Its representation in the *AddressSpace* is formally defined in Table 37.

Table 37 – CertificateRequestedAuditEventType Definition

Attribute	**Value**				
BrowseName	CertificateRequestedAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					
HasProperty	Variable	CertificateGroup	NodeId	PropertyType	Mandatory
HasProperty	Variable	CertificateType	NodeId	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

The *CertificateGroup Property* specifies the *Certificate Group* that was affected by the update.

The *CertificateType Property* specifies the type of *Certificate* that was updated.

7.6.10 CertificateDeliveredAuditEventType

This event is raised when a certificate is delivered by the GDS to a *Client*.

This is the result of a *FinishRequest Method* completing successfully.

Its representation in the *AddressSpace* is formally defined in Table 38.

Table 38 – CertificateDeliveredAuditEventType Definition

Attribute	**Value**				
BrowseName	CertificateDeliveredAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					
HasProperty	Variable	CertificateGroup	NodeId	PropertyType	Mandatory
HasProperty	Variable	CertificateType	NodeId	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

The *CertificateGroup Property* specifies the *Certificate Group* that was affected by the update.

The *CertificateType Property* specifies the type of *Certificate* that was updated.

7.7 Information Model for Push Certificate Management

7.7.1 Overview

If a *Server* supports Push Management it is required to support an information model as part of its address space. It shall support the *ServerConfiguration Object* shown in Figure 15. This *Object* shall only be visible and accessible to administrators and/or the GDS.

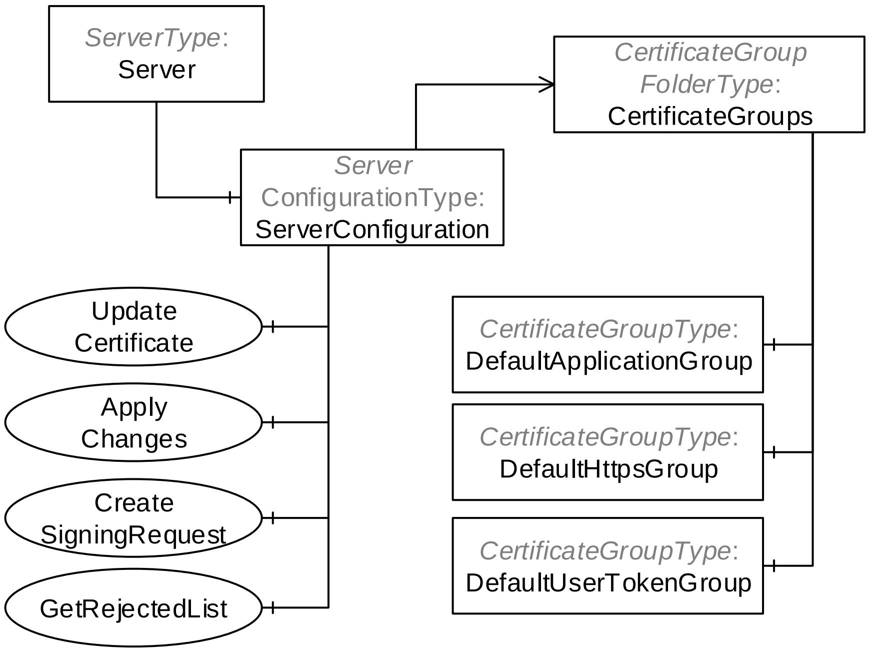

Figure 15 – The AddressSpace for the Server that supports Push Management

All access to *Methods* defined on the *ServerConfiguration Object* shall be over an encrypted channel. In addition, Servers should have user credentials with administrator privileges.

7.7.2 ServerConfiguration

This *Object* allows access to the *Server's* configuration and it is the target of an *HasComponent* reference from the *Server Object* defined in Part 5.

This *Object* and its immediate children shall be visible (i.e. browse access is available) to users who can access the *Server Object.* The children of the *CertificateGroups Object* should only be visible to authorized administrators.

Its representation in the *AddressSpace* is formally defined in Table 39.

Table 39 – ServerConfiguration Object Definition

Attribute	**Value**				
BrowseName	ServerConfiguration				
Namespace	CORE (see 3.3)				
TypeDefinition	ServerConfigurationType defined in 7.7.3.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**

7.7.3 ServerConfigurationType

This type defines an *ObjectType* which represents the configuration of a *Server* which supports Push Management . Its values are defined in Table 40. There is always exactly one instance in the *Server AddressSpace*.

Table 40 – ServerConfigurationType Definition

Attribute	**Value**				
BrowseName	ServerConfigurationType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**Type Definition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					

HasComponent	Object	CertificateGroups		CertificateGroup FolderType	Mandatory
HasProperty	Variable	ServerCapabilities	String[]	PropertyType	Mandatory
HasProperty	Variable	SupportedPrivateKeyFormats	String[]	PropertyType	Mandatory
HasProperty	Variable	MaxTrustListSize	UInt32	PropertyType	Mandatory
HasProperty	Variable	MulticastDnsEnabled	Boolean	PropertyType	Mandatory
HasComponent	Method	UpdateCertificate	See 7.7.4		Mandatory
HasComponent	Method	ApplyChanges	See 7.7.5.		Optional
HasComponent	Method	CreateSigningRequest	See 7.7.6.		Mandatory
HasComponent	Method	GetRejectedList	See 7.7.7.		Mandatory

The *CertificateGroups Object* organizes the *Certificate Groups* supported by the *Server*. It is described in 7.5.17. *Servers* shall support the *DefaultApplicationGroup* and may support the *DefaultHttpsGroup* or the *DefaultUserTokenGroup*. *Servers* may support additional *Certificate Groups* depending on their requirements. For example, a *Server* with two network interfaces should have a different *Trust List* for each interface. The second *Trust List* would be represented as a new *CertificateGroupType* Object organized by *CertificateGroups Folder.*

The *ServerCapabilities Property* specifies the capabilities from Annex D which the *Server* supports. The value is the same as the value reported to the *LocalDiscoveryServer* when the *Server* calls the *RegisterServer2 Service*.

The *SupportedPrivateKeyFormats* specifies the *PrivateKey* formats supported by the Server. Possible values include "PEM" (see RFC 5958) or "PFX" (see PKCS #12). The array is empty if the *Server* does not allow external *Clients* to update the *PrivateKey*.

The *MaxTrustListSize* is the maximum size of the *Trust List* in bytes. 0 means no limit. The default is 65 535 bytes.

If *MulticastDnsEnabled* is TRUE then the *Server* announces itself using multicast DNS. It can be changed by writing to the *Variable*.

The *GetRejectedList Method* returns the list of *Certificates* which have been rejected by the *Server*. It can be used to track activity or allow administrators to move a rejected *Certificate* into the *Trust List*.

The *UpdateCertificate Method* is used to update a *Certificate*.

The *ApplyChanges Method* is used to apply any security related changes if the *Server* sets the *applyChangesRequired* flag when another *Method* is called. *Servers* should minimize the impact of applying the new configuration, however, it could require that all existing *Sessions* be closed and re-opened by the *Clients*.

The *CreateSigningRequest Method* asks the *Server* to create a PKCS #10 encoded *Certificate Request* that is signed with the *Server's* private key.

7.7.4 UpdateCertificate

UpdateCertificate is used to update a *Certificate* for a *Server*.

There are the following three use cases for this *Method*:

- The new *Certificate* was created based on a signing request created with the *Method CreateSigningRequest* defined in 7.7.6. In this case there is no *privateKey* provided.
- A new *privateKey* and *Certificate* was created outside the *Server* and both are updated with this *Method*.
- A new *Certificate* was created and signed with the information from the old *Certificate*. In this case there is no *privateKey* provided.

The *Server* shall do all normal integrity checks on the *Certificate* and all of the issuer *Certificates*. If errors occur the *Bad_SecurityChecksFailed* error is returned.

The *Server* shall report an error if the public key does not match the existing *Certificate* and the privateKey was not provided.

If the *Server* returns *applyChangesRequired*=FALSE then it is indicating that it is able to satisfy the requirements specified for the *ApplyChanges Method*.

This *Method* requires an encrypted channel and that the Client provides credentials with administrative rights on the Server.

Signature

```
UpdateCertificate(
          [in] NodeId certificateGroupId
          [in] NodeId certificateTypeId
          [in] ByteString certificate
          [in] ByteString[] issuerCertificates
          [in] String privateKeyFormat
          [in] ByteString privateKey
          [out] Boolean applyChangesRequired
     );
```

Argument	Description
certificateGroupId	The NodeId of the *Certificate Group Object* which is affected by the update. If null the *DefaultApplicationGroup* is used.
certificateTypeId	The type of *Certificate* being updated. The set of permitted types is specified by the *CertificateTypes Property* belonging to the *Certificate Group*.
certificate	The DER encoded *Certificate* which replaces the existing Certificate.
issuerCertificates	The issuer *Certificates* needed to verify the signature on the new *Certificate*.
privateKeyFormat	The format of the *Private Key* (PEM or PFX). If the *privateKey* is not specified the *privateKeyFormat* is null or empty.
privateKey	The *Private Key* encoded in the *privateKeyFormat*.
applyChangesRequired	Indicates that the *ApplyChanges Method* shall be called before the new *Certificate* will be used.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The certificateTypeId or certificateGroupId is not valid.
Bad_CertificateInvalid	The *Certificate* is invalid or the format is not supported.
Bad_NotSupported	The *PrivateKey* is invalid or the format is not supported.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityChecksFailed	Some failure occurred verifying the integrity of the *Certificate*.

Table 41 specifies the *AddressSpace* representation for the *UpdateCertificate Method*.

Table 41 – UpdateCertificate Method AddressSpace Definition

Attribute	Value				
BrowseName	UpdateCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.7.5 ApplyChanges

ApplyChanges is used to tell the *Server* to apply any security changes.

This *Method* should only be called if a previous call to a *Method* that changed the configuration returns applyChangesRequired=true (see 7.7.4).

If the *Server Certificate* has changed, *Secure Channels* using the old *Certificate* will eventually be interrupted. The only leeway the *Server* has is with the timing. In the best case, the *Server* can close the *TransportConnections* for the affected *Endpoints* and leave any *Subscriptions* intact. This should appear no different than a network interruption from the perspective of the *Client*. The *Client* should be prepared to deal with *Certificate* changes during its reconnect logic. In the worst case, a full shutdown which affects all connected *Clients* will be necessary. In the latter case, the *Server* shall advertise its intent to interrupt connections by setting the *SecondsTillShutdown* and *ShutdownReason Properties* in the *ServerStatus Variable*.

If the *Secure Channel* being used to call this *Method* will be affected by the *Certificate* change then the *Server* shall introduce a delay long enough to allow the caller to receive a reply.

This *Method* requires an encrypted channel and that the *Client* provide credentials with administrative rights on the *Server*.

Signature

```
ApplyChanges();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 42 specifies the *AddressSpace* representation for the *ApplyChanges Method*.

Table 42 – ApplyChanges Method AddressSpace Definition

Attribute	Value				
BrowseName	ApplyChanges				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

7.7.6 CreateSigningRequest

CreateSigningRequest Method asks the *Server* to create a PKCS #10 DER encoded *Certificate Request* that is signed with the *Server's* private key. This request can be then used to request a *Certificate* from a CA that expects requests in this format.

This *Method* requires an encrypted channel and that the *Client* provide credentials with administrative rights on the *Server*.

Signature

```
CreateSigningRequest(
      [in]  NodeId certificateGroupId,
   [in]     NodeId certificateTypeId,
      [in]  String subjectName,
      [in]  Boolean regeneratePrivateKey,
   [in]     ByteString nonce,
      [out] ByteString certificateRequest
      );
```

Argument	Description
certificateGroupId	The NodeId of the *Certificate Group Object* which is affected by the request. If null the *DefaultApplicationGroup* is used.
certificateTypeId	The type of *Certificate* being requested. The set of permitted types is specified by the *CertificateTypes Property* belonging to the *Certificate Group*.
subjectName	The subject name to use in the *Certificate Request*. If not specified the *SubjectName* from the current *Certificate* is used. The format of the *subjectName* is defined in 7.6.4.
regeneratePrivateKey	If TRUE the *Server* shall create a new *Private Key* which it stores until the matching signed *Certificate* is uploaded with the *UpdateCertificate Method*. Previously created *Private Keys* may be discarded if *UpdateCertificate* was not called before calling this method again. If FALSE the *Server* uses its existing *Private Key.*
nonce	Additional entropy which the caller shall provide if regeneratePrivateKey is TRUE. It shall be at least 32 bytes long.
certificateRequest	The PKCS #10 DER encoded *Certificate Request.*

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The *certificateTypeId, certificateGroupId* or *subjectName* is not valid.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 43 specifies the *AddressSpace* representation for the *CreateSigningRequest Method*.

Table 43 – CreateSigningRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	CreateSigningRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.7.7 GetRejectedList

GetRejectedList Method returns the list of *Certificates* that have been rejected by the *Server*.

No rules are defined for how the *Server* updates this list or how long a *Certificate* is kept in the list. It is recommended that every valid but untrusted *Certificate* be added to the rejected list as long as storage is available. *Servers* should omit older entries from the list returned if the maximum message size is not large enough to allow the entire list to be returned.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights on the *Server*.

Signature

```
GetRejectedList(
      [out] ByteString[] certificates
      );
```

Argument	Description
certificates	The DER encoded form of the Certificates rejected by the Server.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 44 specifies the *AddressSpace* representation for the *GetRejectedList Method*.

Table 44 – GetRejectedList Method AddressSpace Definition

Attribute	Value				
BrowseName	GetRejectedList				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.7.8 CertificateUpdatedAuditEventType

This event is raised when the *Application Certificate* is changed.

This is the result of a *UpdateCertificate Method* completing successfully or failing.

Its representation in the *AddressSpace* is formally defined in Table 45.

Table 45 – CertificateUpdatedAuditEventType Definition

Attribute	Value				
BrowseName	CertificateUpdatedAuditEventType				
Namespace	CORE (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					
HasProperty	Variable	CertificateGroup	NodeId	PropertyType	Mandatory
HasProperty	Variable	CertificateType	NodeId	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

The *CertificateGroup Property* specifies the *Certificate Group* that was affected by the update.

The *CertificateType Property* specifies the type of *Certificate* that was updated.

8 KeyCredential Management

8.1 Overview

KeyCredential management functions allow the management and distribution of *KeyCredentials* which *OPC UA Applications* use to access *Authorization Services* and/or *Brokers*. An application that provides the *KeyCredential* management functions is called a *KeyCredentialService* and is typically combined with the GDS into a single application.

There are two primary models for *KeyCredential* management: pull and push management. In pull management, the application acts as a *Client* and uses the *Methods* on the *KeyCredentialService* to request and update *KeyCredentials*. The application is responsible for ensuring the *KeyCredentials* are kept up to date. In push management the application acts as a *Server* and exposes *Methods* which the *KeyCredentialService* can call to update the *KeyCredentials* as required.

A *KeyCredentialService* can directly manage the *KeyCredentials* it supplies or it may act as an intermediary between a *Client* and a system that does not support OPC UA such as Azure AD or LDAP.

Note that *KeyCredentials* are secrets that are directly passed to *Authorization Services* and/or *Brokers* and are not *Certificates* with private keys. *Certificate* distribution is managed by the *Certificate* management model described in 7. For example, *Authorization Service*s that support OAuth2 often require the client to provide a client_id and client_secret parameter with any request. The *KeyCredentials* are the values that the application shall place in these parameters.

8.2 Pull Management

Pull management is performed by using a *KeyCredentialManagement Object* (see 8.4.3). It allows *Clients* to request credentials for *Authorization Services* or *Brokers* which are supported by the *KeyCredentialService*. The interactions between the *Client* and the *KeyCredentialService* during pull management are illustrated in Figure 16.

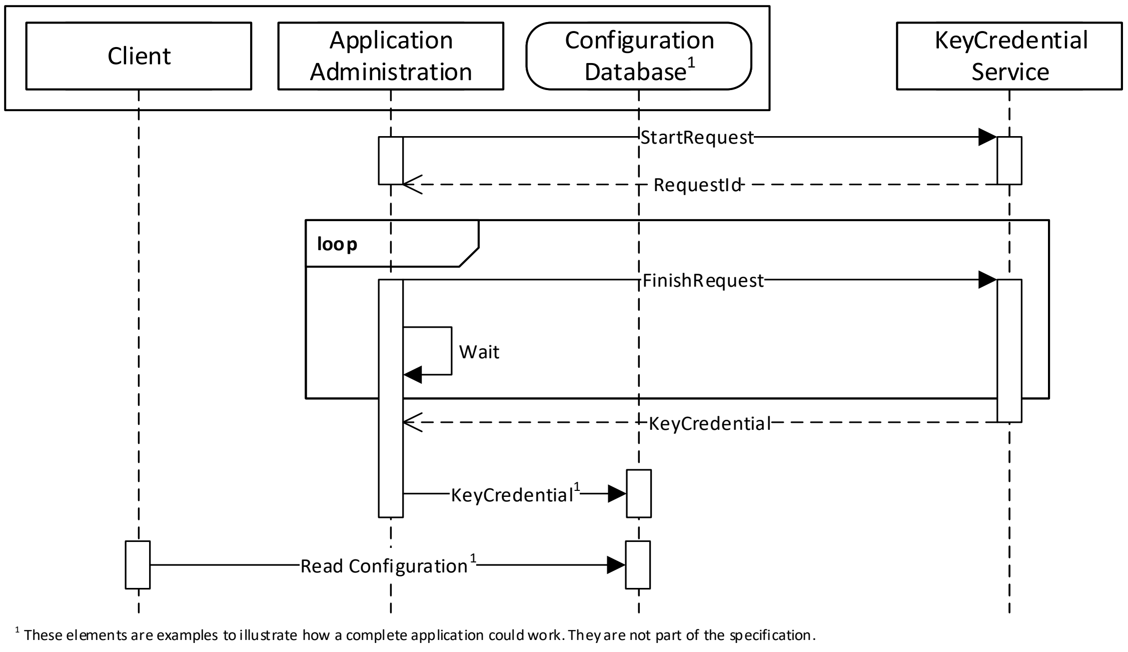
Client
Application Administration
Configuration Database1
KeyCredential Service
StartRequest
RequestId
loop
FinishRequest
Wait
KeyCredential
KeyCredential1
Read Configuration1
1 These elements are examples to illustrate how a complete application could work. They are not part of the specification.

Figure 16 – The Pull Model for KeyCredential Management

The Application Administration component may be part of the *Client* or a standalone utility that understands how the *Client* persists its configuration information in its Configuration Database. The administration and database components are examples to illustrate how an application could be built and are not a requirement.

Requesting credentials is a two stage process because some *KeyCredentialServices* require a human to review and approve requests. The calls to the *FinishKeyCredentialRequest Method* may not be periodic and could be initiated by events such as a user starting up the application or interacting with a UI element such as a button.

KeyCredentials can only be requested for *Clients* which are trusted by the *KeyCredentialService*.

Security in pull management requires an encrypted channel and the use of administrator credentials for the *KeyCredentialService* that ensure only authorized users can request *KeyCredentials*.

8.3 Push Management

Push management is performed by using a *KeyCredentialConfiguration Object* (see 8.5.2) which is a component of the *KeyCredentialManagement Folder* which is component of the *ServerConfiguration Object* in a *Server*. The interactions between the Administration application and the *KeyCredentialService* during push management are illustrated in Figure 17.

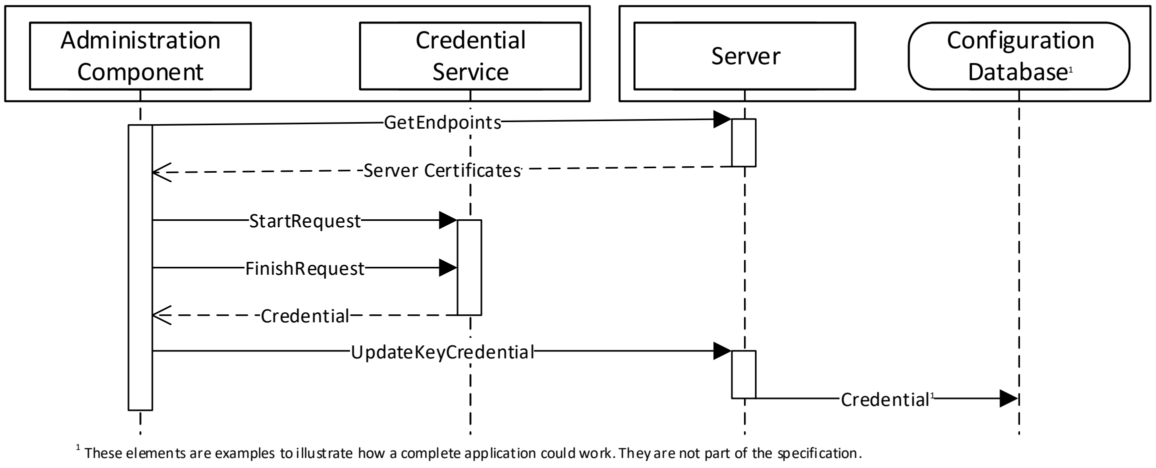
Administration Component
Credential Service
Server
Configuration Database[1]
GetEndpoints
Server Certificates
StartRequest
FinishRequest
Credential
UpdateKeyCredential
Credential[1]
[1] These elements are examples to illustrate how a complete application could work. They are not part of the specification.

Figure 17 – The Push Model for KeyCredential Management

The Administration Component may use internal APIs to manage *KeyCredentials* or it could be a standalone utility that uses OPC UA to communicate with a *Server* which supports the pull model (see 8.2). The Configuration Database is used by the *Server* to persist its configuration information. The administration and database components are examples to illustrate how an application could be built and are not a requirement.

To ensure security of the *KeyCredentials,* the *KeyCredentialService* component can require that secrets be encrypted with a key only known to the intended recipient of the *KeyCredentials*. For this reason, the Administration Component uses the *GetEndpoints Service* to read the *Certificate* from the *Server* before initiating the credential request on behalf of the *Server*.

Security, when using the push management model, requires an encrypted channel and the use of administrator credentials for the *Server* that ensure only authorized users can update *KeyCredentials*. If the KeyCredentialService component is separate from the Administration Component then different administrator credentials are required for the *Server* that exposes the that ensure only authorized users can request new *KeyCredentials* on behalf of *Servers.*

8.4 Information Model for Pull Management

8.4.1 Overview

The *AddressSpace* used for pull management is shown in Figure 18. *Clients* interact with the *Nodes* defined in this model when they need to request or revoke *KeyCredentials* for themselves or for another application. The *KeyCredentialManagement Folder* is a well-known *Object* that appears in the *AddressSpace* of any *Server* which supports *KeyCredential* management.

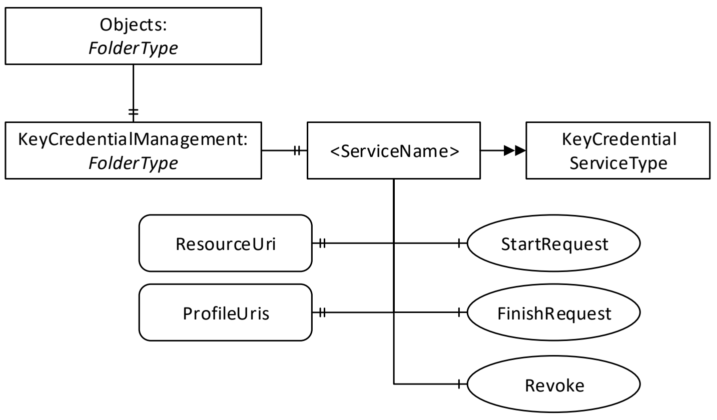

Figure 18 – The Address Space used for Pull KeyCredential Management

8.4.2 KeyCredentialManagement

This *Object* is an instance of *FolderType.* It contains the *KeyCredentialService Objects* which may be accessed via the *Server*. It is the target of an *Organizes* reference from the *Objects Folder* defined in Part 5. It is defined in Table 46.

Table 46 – KeyCredentialManagement Object Definition

Attribute	**Value**			
BrowseName	KeyCredentialManagement			
Namespace	GDS (see 3.3)			
TypeDefinition	FolderType defined in Part 5.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	<ServiceName>	KeyCredentialServiceType	OptionalPlaceholder

8.4.3 KeyCredentialServiceType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows the management of *KeyCredentials*. It is defined in Table 47.

Table 47 – KeyCredentialServiceType Definition

Attribute	**Value**				
BrowseName	KeyCredentialServiceType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasProperty	Variable	ResourceUri	String	PropertyType	Mandatory
HasProperty	Variable	ProfileUris	String[]	PropertyType	Mandatory
HasComponent	Method	StartRequest		Defined in 8.4.4.	Mandatory
HasComponent	Method	FinishRequest		Defined in 8.4.5.	Mandatory
HasComponent	Method	Revoke		Defined in 8.4.6.	Optional

The *ResourceUri Property* uniquely identifies the resource that accepts the *KeyCredentials* provided by the *KeyCredentialService Object*.

The *ProfileUris Property* specifies URIs assigned in Part 7 to the authentication mechanism used to communicate with the resource that accepts *KeyCredentials* provided by the *Object*. For example, it could specify that the resource returns JWTs using OAuth2 HTTP based APIs. As another example, it could specify an MQTT broker that expects a username/password.

The *StartRequest Method* is used to initiate a request for new *KeyCredentials* for an application. This request may complete immediately or it can require offline approval by an administrator.

The *FinishRequest Method* is used to complete a request created by calling *StartRequest* . If the *KeyCredential* is available it is returned. If request is not yet completed it returns *Bad_NothingToDo*.

The *Revoke Method* is used to revoke a previously issued *KeyCredential*.

8.4.4 StartRequest

StartRequest is used to request a new *KeyCredential*.

The *KeyCredential* secret may be encrypted with the public key of the *Certificate* supplied in the request. The *SecurityPolicyUri* specifies the security profile used for the encryption.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights for the application requesting the credentials.

Signature

```
StartRequest (
       [in]  String        applicationUri,
       [in]  ByteString certificate,
       [in]  String        securityPolicyUri,
       [in]  NodeId[]   requestedRoles,
       [out] NodeId        requestId
      );
```

Argument	Description
applicationUri	The *applicationUri* of the application receiving the *KeyCredentials*. The request is rejected *applicationUri* does not uniquely identify an application known to the GDS (see 6.3.6). If the requestor is not the same as the application used to create the *Secure Channel* then a *Certificate* should be provided.
certificate	The *Certificate* containing the key used to encrypt the returned *KeyCredential* secret. This is the DER encoded form of an X.509 v3 *Certificate* as described in Part 6. Not specified if no encryption is required. If the *securityPolicyUri* is provided this field shall be provided.
securityPolicyUri	The *SecurityPolicy* used to encrypt the secret. If the *certificate* is provided this field shall be provided.
requestedRoles	A list of *Roles* which should be assigned to the *KeyCredential*. If not provided the *Server* chooses suitable defaults. The *Server* ignores *Roles* which it does not recognize or if the caller is not authorized to request access to the *Role*.
requestId	A unique identifier for the request. This identifier shall be passed to the *FinishRequest* (see 8.4.5).

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationUri* is not known to the GDS.
Bad_ConfigurationError	The *applicationUri* is used by multiple records in the GDS.
Bad_CertificateInvalid	The *Certificate* is invalid.
Bad_SecurityPolicyRejected	The *SecurityPolicy* is unrecognized or not allowed or does not match the *Certificate*.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 48 specifies the *AddressSpace* representation for the *StartRequest Method*.

Table 48 – StartRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	StartRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.4.5 FinishRequest

FinishRequest is used to retrieve a *KeyCredential*.

If a *Certificate* was provided in the request then the *KeyCredential* secret is encrypted using an asymmetric encryption algorithm specified by the *SecurityPolicyUri* provided in the request.

The format of the signed and encrypted *credentialSecret* is the same as the Version 2 Token Secret Format defined in Part 4. When used for the *credentialSecret*, the signature is provided by the source of the *KeyCredential* which can be the GDS *Application Instance Certificate*. The *serverNonce* is a random number generated by the GDS.

If the return code is *Bad_RequestNotComplete* then the request has not been processed and the *Client* should call again. The recommended time between calls depends on the GDS.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights for the application requesting the credentials.

Signature

```
FinishRequest (
    [in]  NodeId     requestId,
    [in]  Boolean    cancelRequest,
    [out] String     credentialId,
    [out] ByteString credentialSecret,
    [out] NodeId[]   grantedRoles
   );
```

Argument	Description
requestId	The identifier returned from a previous call to *StartRequest.*
cancelRequest	If TRUE the request is cancelled and no *KeyCredentials* are returned. If FALSE the normal processing proceeds.
credentialId	The unique identifier for the *KeyCredential*.
credentialSecret	The secret associated with the *KeyCredential*.
certificateThumbprint	The thumbprint of the *Certificate* containing the key used to encrypt the secret. Not specified if the secret is not encrypted.
securityPolicyUri	The *SecurityPolicy* used to encrypt the secret. If not specified the secret is not encrypted.
grantedRoles	A list of *Roles* which have been granted to *KeyCredential*. If empty then the information is not relevant or not available.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The *requestId* is does not reference to a valid request for the *Application*.
Bad_RequestNotComplete	The request has not been processed by the *Server* yet..
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_RequestNotAllowed	The *KeyCredential* manager rejected the request. The text associated with the error should indicate the exact reason.

Table 49 specifies the *AddressSpace* representation for the *FinishRequest Method*.

Table 49 – FinishRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	FinishRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.4.6 Revoke

Revoke is used to revoke a *KeyCredential* used by a *Server*.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights for the application which is having the credentials revoked.

Signature

```
Revoke (
     [in]  String credentialId
    );
```

Argument	Description
credentialId	The unique identifier for the *KeyCredential*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The *credentialId* is does not reference a valid *KeyCredential*.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 50 specifies the *AddressSpace* representation for the *RevokeKeyCredential Method*.

Table 50 – Revoke Method AddressSpace Definition

Attribute	Value				
BrowseName	Revoke				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

8.4.7 KeyCredentialAuditEventType

This abstract event is raised when an operation affecting *KeyCredentials* occur

This *Event* and it subtypes are security related and *Servers* shall only report them to users authorized to view security related audit events.

Its representation in the *AddressSpace* is formally defined in Table 52.

Table 51 – KeyCredentialAuditEventType Definition

Attribute	Value				
BrowseName	KeyCredentialAuditEventType				
Namespace	CORE (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *AuditUpdateMethodEventType* defined in Part 5.					
HasProperty	Variable	ResourceUri	String	PropertyType	Mandatory
HasSubtype	ObjectType	KeyCredentialRequestedAuditEventType		Defined in 8.4.8.	
HasSubtype	ObjectType	KeyCredentialDeliveredAuditEventType		Defined in 8.4.9.	
HasSubtype	ObjectType	KeyCredentialRevokedAuditEventType		Defined in 8.4.10.	
HasSubtype	ObjectType	KeyCredentialUpdatedAuditEventType		Defined in 8.5.5.	
HasSubtype	ObjectType	KeyCredentialDeletedAuditEventType		Defined in 8.5.6.	

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

The *ResourceUri Property* specifies the URI for the resource which accepts the *KeyCredential.*

8.4.8 KeyCredentialRequestedAuditEventType

This event is raised when a new *KeyCredential* request has been accepted or rejected by the *Server*.

This can be the result of a *StartKeyCredentialRequest Method* call.

Its representation in the *AddressSpace* is formally defined in Table 52.

Table 52 – KeyCredentialRequestedAuditEventType Definition

Attribute	**Value**				
BrowseName	KeyCredentialRequestedAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *KeyCredentialAuditEventType* defined in 8.4.7.					

This *EventType* inherits all *Properties* of the *KeyCredentialAuditEventType*.

8.4.9 KeyCredentialDeliveredAuditEventType

This event is raised when a *KeyCredential* is delivered by the *Server* to an application.

This is the result of a *FinishKeyCredentialRequest Method* completing.

Its representation in the *AddressSpace* is formally defined in Table 53.

Table 53 – KeyCredentialDeliveredAuditEventType Definition

Attribute	**Value**				
BrowseName	KeyCredentialDeliveredAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *KeyCredentialAuditEventType* defined in 8.4.7.					

This *EventType* inherits all *Properties* of the *KeyCredentialAuditEventType*.

8.4.10 KeyCredentialRevokedAuditEventType

This event is raised when a *KeyCredential* is revoked.

This is the result of a *RevokeKeyCredential Method* completing.

Its representation in the *AddressSpace* is formally defined in Table 54.

Table 54 – KeyCredentialRevokedAuditEventType Definition

Attribute	**Value**				
BrowseName	KeyCredentialRevokedAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *KeyCredentialAuditEventType* defined in 8.4.7.					

This *EventType* inherits all *Properties* of the *KeyCredentialAuditEventType*.

8.5 Information Model for Push Management

The *AddressSpace* used for push management is shown in Figure 19. *Clients* interact with the *Nodes* defined in this model when they need update the *KeyCredentials* used by a *Server* to access resources such as *Brokers* or *Authorization Servers*. The *NetworkResources Folder* is a well-known *Object* that appears in the *AddressSpace* of any *Server* which supports *KeyCredential* management.

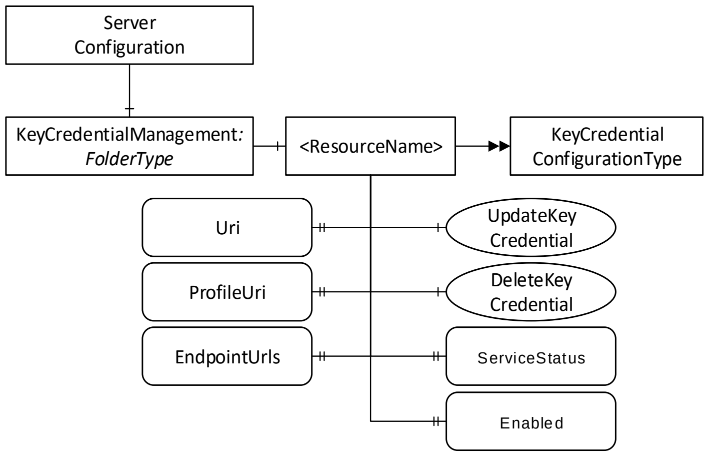

Figure 19 – The Address Space used for Push KeyCredential Management

8.5.1 KeyCredentialConfiguration

This *Object* is an instance of *FolderType.* It contains The *Objects* which make be accessed via the *Server*. It is the target of an *HasComponent* reference from the *ServerConfiguration Object* defined in 7.7.2. It is defined in Table 46.

Table 55 – KeyCredentialConfiguration Object Definition

Attribute	**Value**				
BrowseName	KeyCredentialConfiguration				
Namespace	CORE (see 3.3)				
TypeDefinition	FolderType defined in Part 5.				
References	**NodeClass**	**BrowseName**	**TypeDefinition**		**Modelling Rule**
HasComponent	Object	<ServiceName>	KeyCredentialConfigurationType		OptionalPlaceholder

8.5.2 KeyCredentialConfigurationType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows the configuration of *KeyCredentials* used by the *Server*. It also includes basic status information which report problems accessing the resource that might be related to bad *KeyCredentials*. It is defined in Table 56.

Table 56 – KeyCredentialConfigurationType Definition

Attribute	**Value**				
BrowseName	KeyCredentialConfigurationType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasProperty	Variable	ResourceUri	String	PropertyType	Mandatory
HasProperty	Variable	ProfileUri	String	PropertyType	Mandatory
HasProperty	Variable	EndpointUrls	String[]	PropertyType	Optional
HasProperty	Variable	ServiceStatus	StatusCode	PropertyType	Optional
HasComponent	Method	UpdateCredential		Defined in 8.5.3.	Optional
HasComponent	Method	DeleteCredential		Defined in 8.5.4.	Optional

The *ResourceUri Property* uniquely identifies the resource that accepts the *KeyCredentials*.

The *ProfileUri Property* specifies the protocol used to access the resource.

The *EndpointUrls Property* specifies the URLs that the *Server* uses to access the resource.

The *ServiceStatus Property* indicates the result of the last attempt to communicate with the resource. The following common error values are defined:

ServiceStatus	Description
Bad_OutOfService	Communication was not attempted by the *Server* because *Enabled* is FALSE.
Bad_IdentityTokenRejected	Communication failed because the *KeyCredentials* are not valid.
Bad_NoCommunication	Communication failed because the endpoint is not reachable. Where possible a more specific error code should be used. See Part 4 for a complete list of standard *StatusCodes*.

The *UpdateKeyCredential Method* is used to change the *KeyCredentials* used by the *Server*.

The *DeleteKeyCredential Method* is used to delete the *KeyCredentials* stored by the *Server*.

8.5.3 UpdateCredential

UpdateCredential is used to update a *KeyCredential* used by a *Server*.

The *KeyCredential* secret may be encrypted with the public key of the *Server's Certificate*. The *SecurityPolicyUri* species the algorithm used for encryption. The format of the encrypted data is described in 8.4.5.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights on the *Server*.

Signature

```
UpdateCredential(
    [in] String       credentialId,
    [in] ByteString credentialSecret,
    [in] String       certificateThumbprint,
    [in] String       securityPolicyUri
   );
```

Argument	Description
credentialId	The unique identifier associated with the *KeyCredential*.
credentialSecret	The secret associated with the *KeyCredential*.
certificateThumbprint	The thumbprint of the *Certificate* used to encrypt the secret. This shall be one of the *Application Instance Certificates* assigned to the *Server*. Not specified if the secret is not encrypted.
securityPolicyUri	The *SecurityPolicy* used to encrypt the secret. If not specified the secret is not encrypted.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The credentialId or credentialSecret is not valid.
Bad_CertificateInvalid	The *Certificate* is invalid or it is not one of the *Server's Certificates*.
Bad_SecurityPolicyRejected	The *SecurityPolicy* is unrecognized or not allowed.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 57 specifies the *AddressSpace* representation for the *UpdateKeyCredential Method*.

Table 57 – UpdateCredential Method AddressSpace Definition

Attribute	Value				
BrowseName	UpdateCredential				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.5.4 DeleteCredential

DeleteCredential is used to delete a *KeyCredential* used by a *Server*.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights on the *Server*.

Signature

DeleteCredential()

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.

Table 57 specifies the *AddressSpace* representation for the *DeleteKeyCredential Method*.

Table 58 – DeleteCredential Method AddressSpace Definition

Attribute	Value				
BrowseName	DeleteCredential				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

8.5.5 KeyCredentialUpdatedAuditEventType

This event is raised when a *KeyCredential* is updated.

This *Event* and its subtypes report sensitive security related information. Servers shall only report these *Events* to Clients which are authorized to view such information.

This is the result of a *UpdateCredential Method* completing.

Its representation in the *AddressSpace* is formally defined in Table 59.

Table 59 – KeyCredentialUpdatedAuditEventType Definition

Attribute	Value				
BrowseName	KeyCredentialUpdatedAuditEventType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *KeyCredentialAuditEventType* defined in 8.4.7.					
HasProperty	Variable	ResourceUri	String	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *KeyCredentialAuditEventType*.

8.5.6 KeyCredentialDeletedAuditEventType

This event is raised when a *KeyCredential* is updated.

This is the result of a *DeleteCredential Method* completing.

Its representation in the *AddressSpace* is formally defined in Table 60.

Table 60 – KeyCredentialUpdatedAuditEventType Definition

Attribute	Value				
BrowseName	KeyCredentialDeletedAuditEventType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *KeyCredentialAuditEventType* defined in 8.4.7.					
HasProperty	Variable	ResourceUri	String	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *KeyCredentialAuditEventType*.

9 Authorization Services

9.1 Overview

Authorization Services provide *Access Tokens* to *Clients* that may use them to access resources. A *Server*, such as a GDS, with *Authorization Service* capabilities may support one or more *AuthorizationService Objects* (see 9.5.2) which may represent an internal

Authorization Service or be an API to an external *Authorization Service*. The *Authorization Service* is best used in conjunction with the *Role* model defined in Part 5. In this scenario, the mapping rules assigned to the *Roles* known to the *Server* are used to populate an *Access Token* with the *Roles* associated with the *UserIdentity* provided when the *Client* submits the request. This scenario is illustrated in Figure 20.

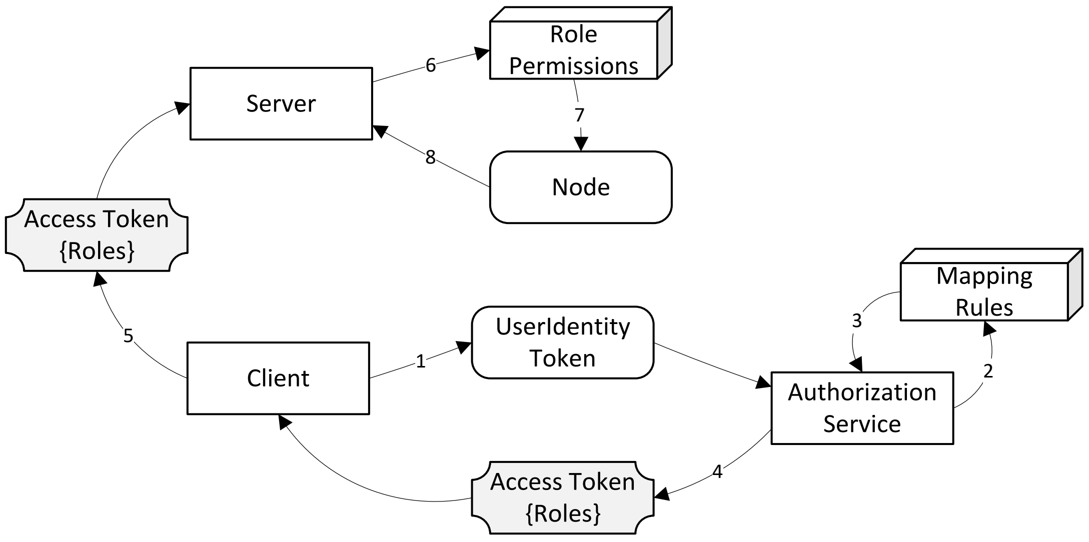

Figure 20 – Roles and Authorization Services

When requesting *Access Tokens* from an *AuthorizationService Object* there are three primary use cases based on where the *UserIdentityToken* comes from: Implicit, Explicit and Chained. These use cases are discussed below. The Implicit and Explicit use cases are implementations of the ‘Indirect’ model for *Authorization Services* described in Part 4. The Chained use case is an implementation of the ‘Direct’ model.

9.2 Implicit

The implicit use case means the *Client’s Application Certificate* and any *UserIdentityToken* associated with the *Session* is used to determine whether an *Access Token* is permitted and what claims are available. This use case is illustrated in Figure 21.

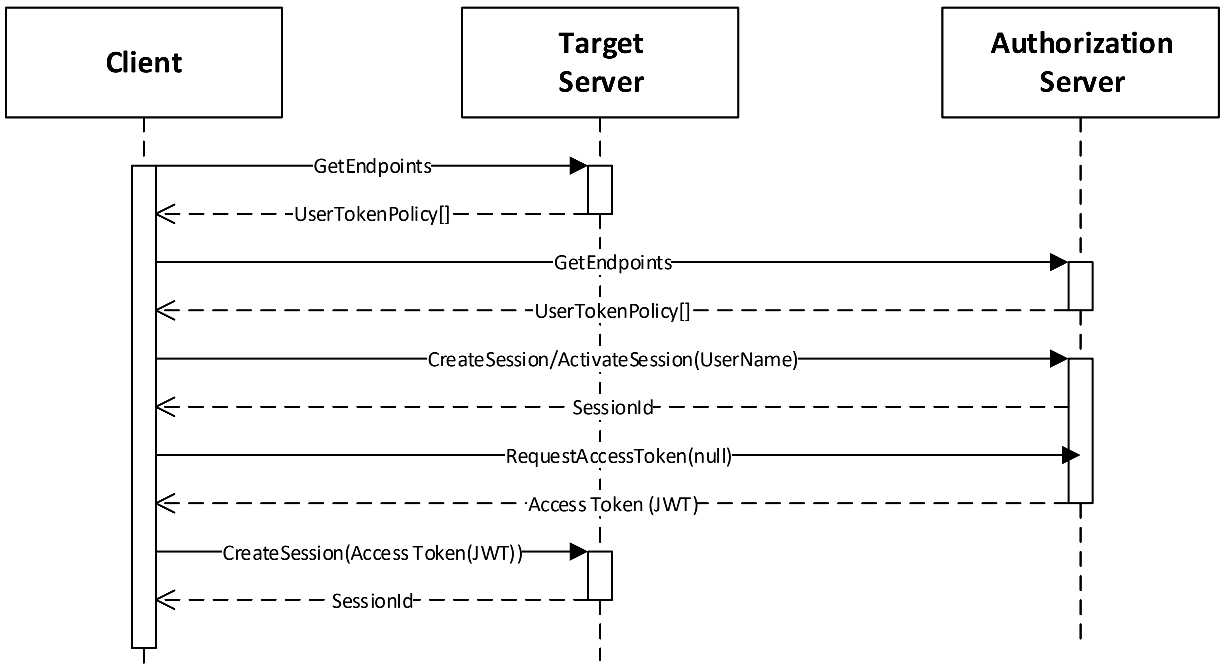

Figure 21 – Implicit Authorization

The Target *Server* is the *Server* that the *Client* wishes to access. It publishes a *UserTokenPolicy* that indicates that it accepts *Access Tokens* from an Authorization Server at a URL specified in the policy. The policy also contains the *NodeId* of the *AuthorizationService Object* which the is used to request the *Access Token*.

The *Client* needs to be trusted by the Authorization Server and this could require the *Client* to present user credentials. These credentials can be provided to the *Client* out-of-band (e.g. an administrator specified them in the *Client* configuration file).

The *Session* may be created explicitly with a call to *CreateSession* or it can be implicit via a *Session*-less *Method Call*.

After creating the *Session,* the *Client* calls the *RequestAccessToken Method* on the *AuthorizationService Object*. The Authorization Server determines if the *Client* is permitted to receive an *Access Token* and populates it with any claims granted to the *Client*. This claims may include *Roles* granted to the *Session* by applying the mapping rules for the Roles (see Part 3).

Once the *Client* has the *Access Token,* it passes the *Access Token* to the Target *Server* which validates the *Access Token*, as described in Part 4. The Target *Server* is configured out-of-band with the *Certificate* needed to validate the *Access Tokens* issued by the Authorization *Server*.

9.3 Explicit

The explicit use case means the *Client* provides the *UserIdentityToken* used to determine whether an *Access Token* is permitted and what claims are available in the call to *RequestAccessToken*. This use case is illustrated in Figure 22.

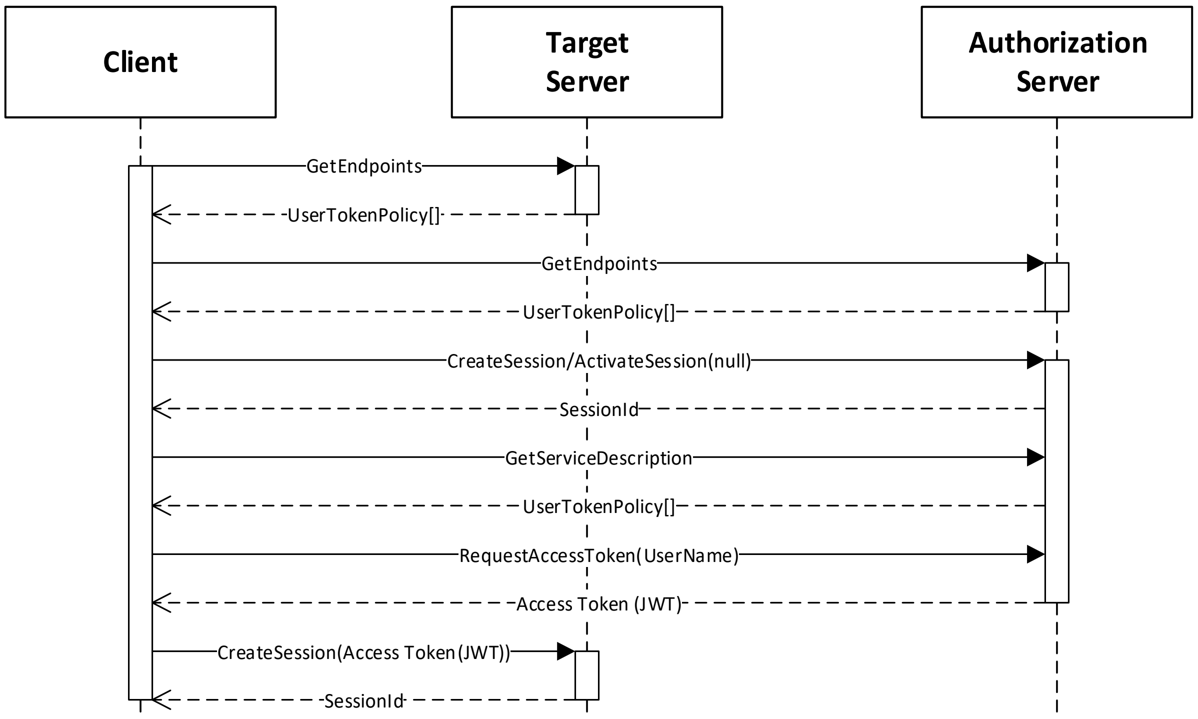

Figure 22 – Explicit Authorization

The Target *Server* is the *Server* that the *Client* wishes to access. The initial interactions are the same as with the Implicit use case described in 9.2.

The *Session* may be created explicitly with a call to *CreateSession* or it can be implicit via a *Session*-less *Method Call*.

After creating the *Session*, the *Client* reads the available *UserTokenPolicies* from the *AuthorizationService Object* if it has not previously cached the information. It then chooses one that matches credentials that it has been provided out-of-band. The *Client* then calls the *RequestAccessToken Method* on the *AuthorizationService Object*.

The Authorization *Server* determines if the *Client* is permitted to receive an *Access Token*. The rest of the interactions are the same as described in 9.2.

9.4 Chained

The chained use case means the *Client* provides an *Access Token* issued by another *Authorization Service* acting as an *Identity Provider*. This use case is illustrated in Figure 23.

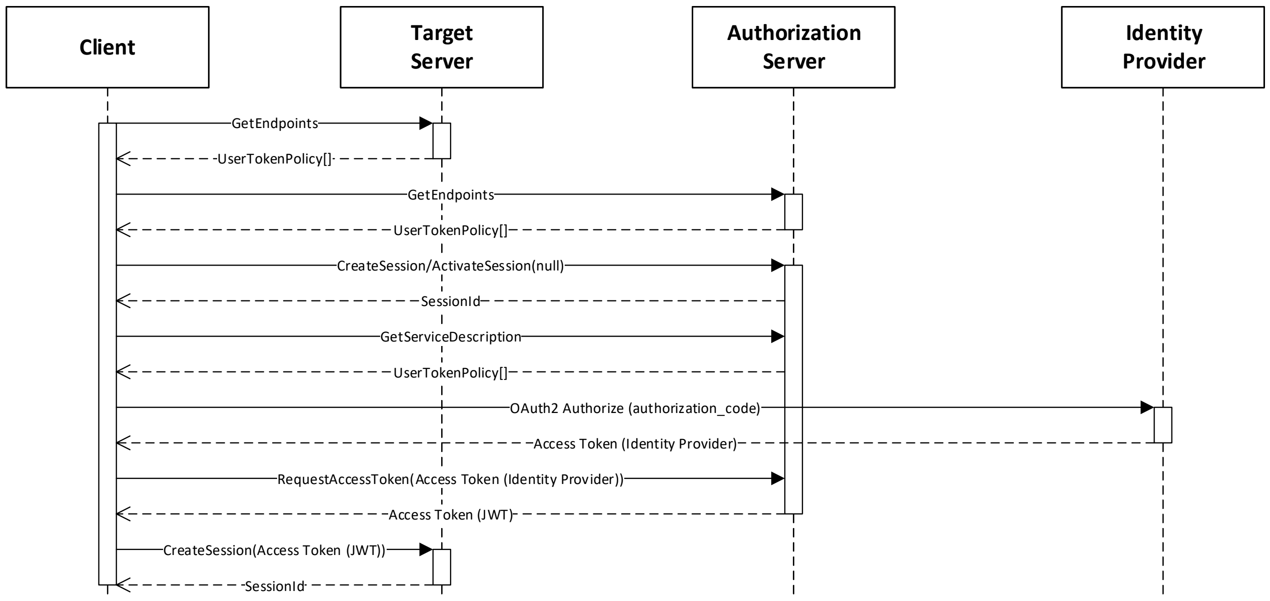
Client
Target Server
Authorization Server
Identity Provider
GetEndpoints
UserTokenPolicy[]
GetEndpoints
UserTokenPolicy[]
CreateSession/ActivateSession(null)
SessionId
GetServiceDescription
UserTokenPolicy[]
OAuth2 Authorize (authorization_code)
Access Token (Identity Provider)
RequestAccessToken(Access Token (Identity Provider))
Access Token (JWT)
CreateSession(Access Token (JWT))
SessionId

Figure 23 – Chained Authorization

The Target *Server* is the *Server* that the *Client* wishes to access. The initial interactions are the same as with the Implicit use case described in 9.2.

The *Session* may be created explicitly with a call to *CreateSession* or it can be implicit via a *Session*-less *Method Call*.

After creating the *Session*, the *Client* reads the available *UserTokenPolicies* from the *AuthorizationService Object* if it has not previously cached the information. It then chooses one that references an *Identity Provider* for the user identities that it has available. The user identities may be provided out-of-band or they may be provided by an interactive user. The *Client* then requests an *Access Token* from the *Identity Provider*.

The *Client* then calls the *RequestAccessToken Method* on the *AuthorizationService Object* and passes the *Access Token* from the *Identity Provider*.

The Authorization *Server* determines if the *Client* is permitted to receive an *Access Token* based on the claims granted by the *Identity Provider*. The rest of the interactions are the same as described in 9.2.

9.5 Information Model for Requesting Access Tokens

9.5.1 Overview

The information model for *Authorization Services* which allow *Clients* to request *Access Tokens* from a *Server* is shown in Figure 24.

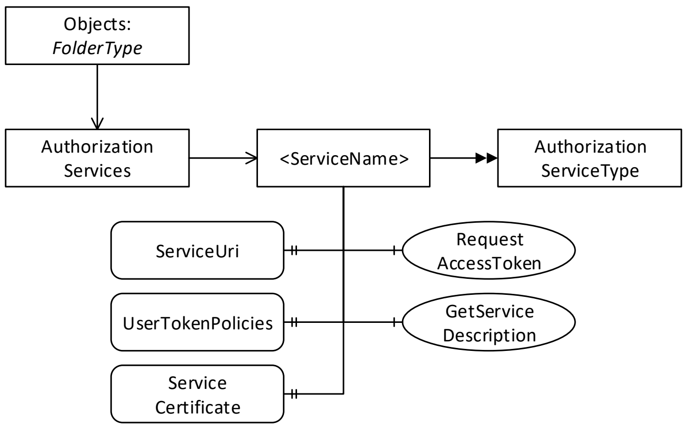

Figure 24 – The Model for Requesting Access Tokens from Authorization Services

9.5.2 AuthorizationServices

This *Object* is an instance of *FolderType.* It contains The *AuthorizationService Objects* which may be accessed via the GDS. It is the target of an *Organizes* reference from the *Objects Folder* defined in Part 5. It is defined in Table 61.

Table 61 – AuthorizationServices Object Definition

Attribute	Value			
BrowseName	AuthorizationServices			
Namespace	GDS (see 3.3)			
TypeDefinition	*FolderType* defined in Part 5.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	<ServiceName>	AuthorizationServiceType	OptionalPlaceholder

9.5.3 AuthorizationServiceType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows access to an *Authorization Service*. It is defined in Table 62.

Table 62 – AuthorizationServiceType Definition

Attribute	Value				
BrowseName	AuthorizationServiceType				
Namespace	GDS (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasProperty	Variable	ServiceUri	String	PropertyType	Mandatory
HasProperty	Variable	ServiceCertificate	ByteString	PropertyType	Mandatory
HasProperty	Variable	UserTokenPolicies	UserTokenPolicy []	PropertyType	Optional
HasComponent	Method	GetServiceDescription		Defined in 9.5.5.	Mandatory
HasComponent	Method	RequestAccessToken		Defined in 9.5.4.	Optional

The *ServiceUri* is a globally unique identifier that allows a *Client* to correlate an instance of *AuthorizationServiceType* with instances of *AuthorizationServiceConfigurationType* (see 9.6.3).

The *ServiceCertificate* is the complete chain of *Certificates* needed to validate the *Access Tokens* (see Part 6 for information on encoding chains).

The *UserTokenPolicies Property* specifies the *UserIdentityTokens* which are accepted by the *RequestAccessToken Method.*

The *GetServiceDescription Method* is used read the metadata needed to request *Access Tokens.*

The *RequestAccessToken Method* is used to request an *Access Token* from the *Authorization Service.*

9.5.4 RequestAccessToken

RequestAccessToken is used to request an *Access Token* from an *Authorization Service*. The scenarios where this this *Method* is used are described fully in 9.2, 9.3 and 9.4.

The *PolicyId* and *UserTokenType* of the *identityToken* shall match one of the elements of the *UserTokenPolicies Property*. If the *identityToken* is not provided the *Server* should use the *ApplicationInstanceCertificate* and/or the *UserIdentityToken* provided for the *Session* (or the request if using a *Session*-less *Method Call*) to determine privileges.

If the associated *UserTokenPolicy* provides a *SecurityPolicyUri*, then the *identityToken* is encrypted and digitally signed using the format defined for *UserIdentityToken* secrets in Part 4.

For *UserNameIdentityTokens* the secret is the password and the signature is created with the *Client ApplicationInstanceCertificate*. The signed and encrypted secret is passed in the *password* field.

For *X.509 v3IdentityTokens* the secret is null and signature is created with the key associated with user *Certificate*. The signed and encrypted secret is passed in the *certificateData* field.

For *IssuedIdentityTokens* the secret is the token and the signature is created with the key associated a user *Certificate* or the *Client ApplicationInstanceCertificate.* The signed and encrypted secret is passed in the *tokenData* field.

The *Server* shall check the *signingTime* in against the current system clock. The *Server* shall reject the request if the *signingTime* is outside of a configurable range. A suitable default value is 5 minutes. The permitted clock skew is a *Server* configuration parameter.

This *Method* requires an encrypted channel and that the *Client* provides credentials with administrative rights for the application which is having the credentials revoked.

Signature

```
RequestAccessToken (
      [in]  UserIdentityToken identityToken,
      [in]  String resourceId,
      [out] String accessToken
     );
```

Argument	Description
identityToken	The identity used to authorize the *Access Token* request.
resourceId	The identifier for the Resource that the *Access Token* is used to access. This is usually the *ApplicationUri* for a *Server*.
accessToken	The *Access Token* granted to the application.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_IdentityTokenInvalid	The *identityToken* does not match one of the allowed *UserTokenPolicies*.
Bad_IdentityTokenRejected	The *identityToken* was rejected.
Bad_NotFound	The *resourceId* is not known to the *Server*.
Bad_UserAccessDenied	The current user does not have the rights required.

Table 63 specifies the *AddressSpace* representation for the *RequestAccessToken Method*.

Table 63 – RequestAccessToken Method AddressSpace Definition

Attribute	**Value**				
BrowseName	RequestAccessToken				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

9.5.5 GetServiceDescription

GetServiceDescription is used to read the metadata needed to request *Access Tokens* from the *Authorization Service*.

Signature

```
GetServiceDescription (
       [out] String serviceUri
       [out] ByteString serviceCertificate
       [out] UserTokenPolicy[] policies
      );
```

Argument	**Description**
serviceUri	A globally unique identifier for the *Authorization Service*.
serviceCertificate	The complete chain of *Certificates* needed to validate the *Access Tokens* provided by the *Authorization Service.*
policies	The *UserIdentityTokens* accepted by the *Authorization Service.*

Method Result Codes (defined in Call Service)

Result Code	**Description**
Bad_UserAccessDenied	The current user does not have the rights required.

Table 64 specifies the *AddressSpace* representation for the *GetServiceDescription Method*.

Table 64 – GetServiceDescription Method AddressSpace Definition

Attribute	**Value**				
BrowseName	GetServiceDescription				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

9.5.6 AccessTokenIssuedAuditEventType

This event is raised when a *AccessToken* is issued.

This is the result of a *RequestAccessToken Method* completing.

This *Event* and it subtypes are security related and *Servers* shall only report them to users authorized to view security related audit events.

Its representation in the *AddressSpace* is formally defined in Table 65.

Table 65 – AccessTokenIssuedAuditEventType Definition

Attribute	**Value**				
BrowseName	AccessTokenIssuedAuditEventType				
Namespace	GDS (see 3.3)				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AuditUpdateMethodEventType defined in Part 5.					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in Part 5.

9.6 Information Model for Configuring Servers

9.6.1 Overview

The information model used to provide *Servers* with the information needed to accept *Access Tokens* from *Authorization Services* in Figure 24.

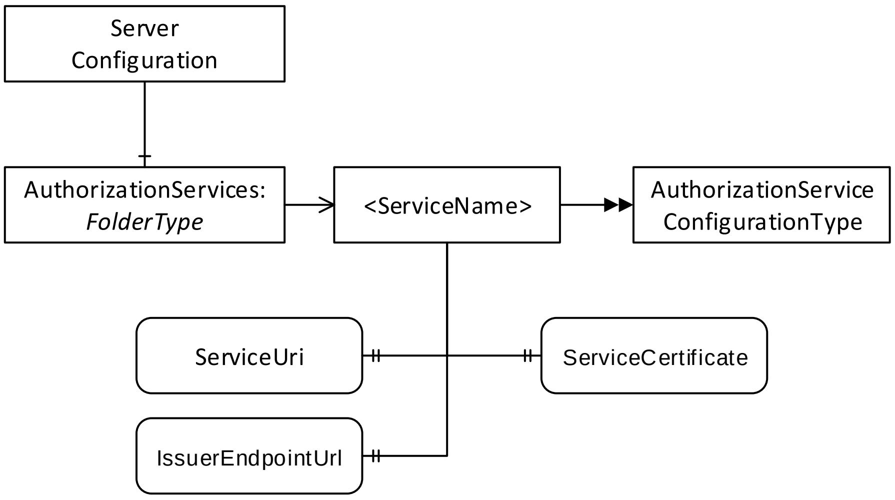

Figure 25 – The Model for Configuring Servers to use Authorization Services

If a *Server* is also a *Client* that needs to access the *Authorization Service*, the necessary *KeyCredentials* can be provided with the push configuration management model (see 8.3).

9.6.2 AuthorizationServices

This *Object* is an instance of *FolderType.* It contains The *AuthorizationServiceConfiguration Objects* which may be accessed via the *Server*. It is the target of an *HasComponent* reference from the *ServerConfiguration Object* defined in 7.7.2. It is defined in Table 61.

Table 66 – AuthorizationServices Object Definition

Attribute	**Value**			
BrowseName	AuthorizationServices			
Namespace	CORE (see 3.3)			
TypeDefinition	*FolderType* defined in Part 5.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	<ServiceName>	AuthorizationServiceConfiguration Type	OptionalPlaceholder

9.6.3 AuthorizationServiceConfigurationType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows the configuration of an *Authorization Service* used by a *Server*. It is defined in Table 67.

Table 67 – AuthorizationServiceConfigurationType Definition

Attribute	**Value**				
BrowseName	AuthorizationServiceConfigurationType				
Namespace	CORE (see 3.3)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in Part 5.					
HasProperty	Variable	ServiceUri	String	PropertyType	Mandatory
HasProperty	Variable	ServiceCertificate	ByteString	PropertyType	Mandatory
HasProperty	Variable	IssuerEndpointUrl	String	PropertyType	Mandatory

The *ServiceUri Property* uniquely identifies the *Authorization Service*.

The *ServiceCertificate Property* has the *Certificate(s)* needed to verify *Access Tokens* issued by the *Authorization Service*. The value is the complete chain of Certificate needed for verification (see Part 6 for information on encoding chains).

The *IssuerEndpointUrl* is the value of the *IssuerEndpointUrl* in *UserTokenPolicies* which require the use of the Authorization Service. This contents of the field depend on the Authorization Service and are described in Part 6.

Annex A
(informative)

Deployment and Configuration

A.1 Firewalls and Discovery

Many systems will have multiple networks that are isolated by firewalls. These firewalls will frequently hide the network addresses of the hosts behind them unless the Administrator has specifically configured the firewall to allow external access. In some networks the Administrator will place hosts with externally available *Servers* outside the firewall as shown in Figure 26.

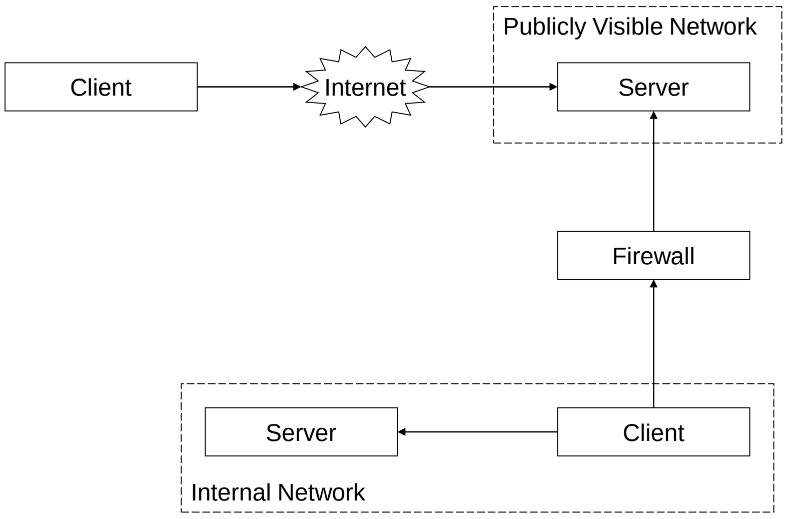

Figure 26 – Discovering Servers Outside a Firewall

In this configuration *Servers* running on the publicly visible network will have the same network address from the perspective of all *Clients* which means the URLs returned by *DiscoveryServer*s are not affected by the location of the *Client*.

In other networks the Administrator will configure the firewall to allow access to selected *Servers.* An example is shown in Figure 27.

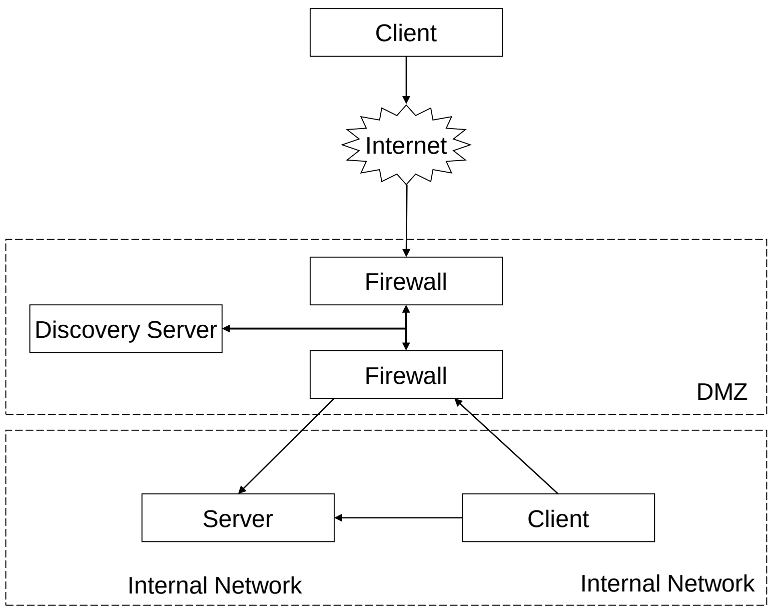

Figure 27 – Discovering Servers Behind a Firewall

In this configuration the address of the *Server* that the Internet *Client* sees will be different from the address that the Internet *Client* sees. This means that the *Server's DiscoveryEndpoint* would return incorrect URLs to the Internet *Client* (assuming it was configured to provide the internal URLs).

Administrators can correct this problem by configuring the *Server* to use multiple *HostNames*. A *Server* that has multiple *HostNames* shall look at the *EndpointUrl* passed to the *GetEndpoints* or *CreateSession* services and return *EndpointDescriptions* with URLs that use the same *HostName*. A Server with multiple *HostNames* shall also return an *Application Instance Certificate* that specifies the *HostName* used in the URL it returns. An Administrator may create a single *Certificate* with multiple HostNames or assign different *Certificates for each HostName* that the *Server* supports.

Note that *Servers* may not be aware of all *HostNames* which can be used to access the *Server* (i.e. a NAT firewall) so *Clients* need to handle the case where the URL used to access the *Server* is different from the *HostNames* in the *Certificate*. This is discussed in more detail in Part 4.

Administrators may also wish to set up a *DiscoveryServer* that is configured with the *ApplicationDescriptions* for *Servers* that are accessible to external *Clients*. This *DiscoveryServer* would have to substitute its own *Endpoint* for the *DiscoveryUrls* in all *ApplicationDescriptions* that it returns when a *Client* calls *FindServers*. This would tell the *Client* to call the *DiscoveryServer* back when it wishes to connect to the *Server*. The *DiscoveryServer* would then request the *EndpointDescriptions* from the actual Server as shown in Figure 28. At this point the *Client* would have all the information it needs to establish a secure channel with the *Server* behind the firewall.

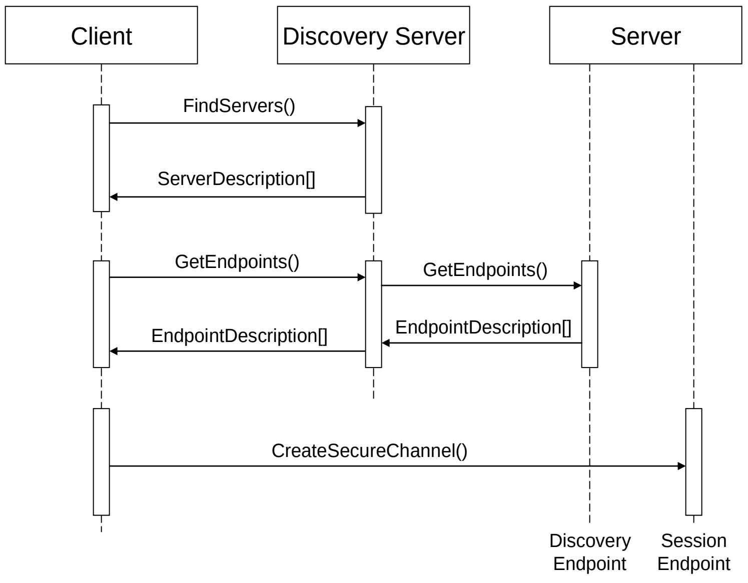

Figure 28 – Using a Discovery Server with a Firewall

In this example, the *DiscoveryServer* outside of the firewall allows the *Administrator* to close off the *Server's DiscoveryEndpoints* to every Client other than the *DiscoveryServer*. The Administrator could eliminate that hole as well if it stored the *EndpointDescriptions* on the *DiscoveryServer*. This allows an Administrator to configure a system in which no public access is allowed to any application behind the firewall. The only access behind the firewall is via a secure connection.

The *DiscoveryServer* could also be replaced with a *DirectoryService* that stores the *ApplicationDescriptions* and/or the *EndpointDescriptions* for the *Servers* behind the firewalls.

A.2 Resolving References to Remote Servers

The UA *AddressSpace* supports references between *Nodes* that exist in different *Server AddressSpace* spaces. These references are specified with a *ExpandedNodeId* that includes the URI of the *Server* which owns the *Node*. A *Client* that wishes to follow a reference to an external *Node* should map the *ApplicationUri* onto an *EndpointUrl* that it can use. A *Client* can do this by using the *GlobalDiscoveryServer* that knows about the *Server*. The process of connecting to a *Server* containing a remote *Node* is illustrated in Figure 29.

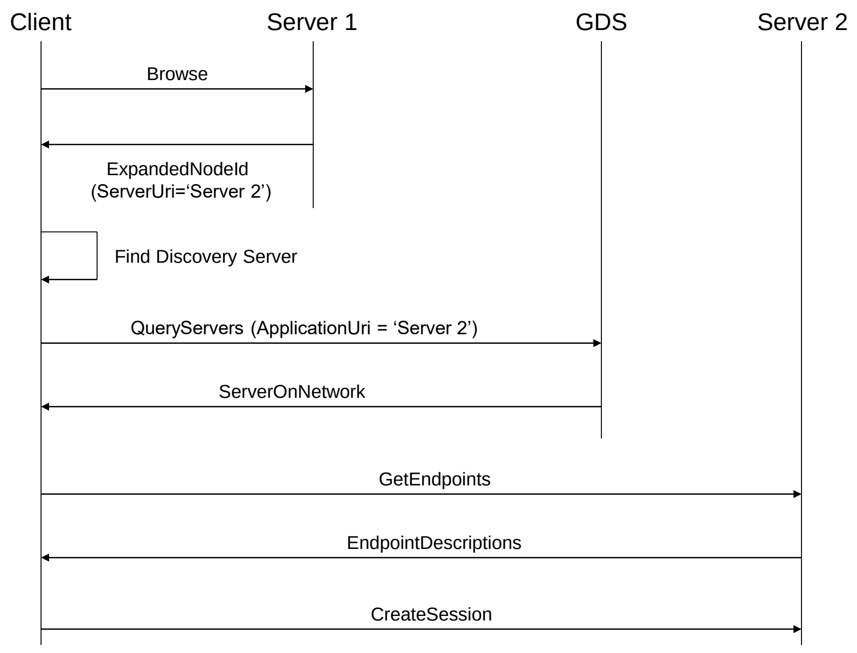

Figure 29 – Following References to Remote Servers

If a GDS not available *Client* may use other strategies to find the *Server* associated with the URI.

Annex B
(normative)

Constants

B.1 Numeric Node Ids

This document defines *Nodes* which are part of the base OPC UA Specification. The numeric identifiers for these *Nodes* are part of the complete list of identifiers defined in Part 6.

In addition, this document defines *Nodes* which are only used by *GlobalDiscoveryServers*.

The *NamespaceUri* for any GDS specific *NodeIds* is http://opcfoundation.org/UA/GDS/

The CSV released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.04/Opc.Ua.Gds.NodeIds.csv

NOTE The latest CSV that is compatible with this version of the standard can be found here:

http://www.opcfoundation.org/UA/schemas//Opc.Ua.Gds.NodeIds.csv

Annex C
(normative)

OPC UA Mapping to mDNS

C.1 DNS Server (SRV) Record Syntax

Annex Cdescribes the OPC UA specific requirements which are above and beyond the more general requirements of the mDNS specification.

mDNS uses DNS SRV records to advertise the services (a.k.a. the *DiscoveryUrls* for the *Servers*) available on the network.

An SRV record has the form:

```
_service._proto.name TTL class SRV priority weight port target
```

service: the symbolic name of the desired service. For OPC UA this field shall be one of service names for OPC UA which are defined in Table 68.

Table 68 – Allowed mDNS Service Names

Service Name	Description
_opcua-tcp	The *DiscoveryUrl* supports the OPC UA TCP mapping (see Part 6). This name is assigned by IANA.
_opcua-tls	The *DiscoveryUrl* supports the OPC UA WebSockets mapping (see Part 6). Note that WebSockets mapping supports multiple encodings. If a *Client* supports more than one encoding it should attempt to use the alternate encodings if an error occurs during connect. This name is assigned by IANA.

proto: the transport protocol of the desired service; For OPC UA this field shall be '_tcp'.

The other fields have no OPC UA specific requirements.

An example SRV record in textual form that might be found in a zone file might be the following:

```
_opcua-tcp._tcp.example.com. 86400 IN SRV 0 5 4840 uaserver.example.com.
```

This points to a server named `uaserver.example.com` listening on TCP port 4840 for OPC UA TCP requests. The priority given here is 0, and the weight is 5 (the priority and weights are not important for OPC UA). The mDNS specification describes the rest of the fields in detail.

C.2 DNS Text (TXT) Record Syntax

The SRV record has a TXT record associated with it that provides additional information about the *DiscoveryUrl*. The format of this record is a sequence of strings prefixed by a length. This specifications adopts the key-value syntax for TXT records described in DNS-SD.

Table 69 defines the syntax for strings that may in the TXT record.

Table 69 – DNS TXT Record String Format

Key-Value Format	Description
path=/<path>	Specifies the text that appears after the port number when constructing a URL. This text always starts with a forward slash (/).

caps=<capability1>,<capability2>	Specifies the capabilities supported by the *Server*. These are short (<=8 character) strings which are published by the OPC Foundation (see Annex D). The number of capabilities supported by a Server should be less than 10.

The *MulticastExtension* shall convert *DiscoveryUrls* to and from these SRV records.

C.3 DiscoveryUrl Mapping

An *DiscoveryUrl* has the form:

```
scheme://hostname:port/path
```

scheme: the protocol used to establish a connection.

hostname: the domain name or *IPAddress* of the host where the *Server* is running.

port: the TCP port on which the *Server* is to be found.

path: additional data used to identify a specific *Server*.

Table 70 – DiscoveryUrl to DNS SRV and TXT Record Mapping

URL Field	Mapping
scheme	The scheme maps onto SRV record service field. The following mappings are defined at this time: opc.tcp _opcua-tcp._tcp. opc.wss _opcua-tls._tcp. https _opcua-https._tcp. The first two are OPC UA service names assigned by IANA. Additional service names may be added in the future. The endpoint shall support the default transport profile for the scheme.
hostname	The hostname maps onto the SRV record target field. If the hostname is an *IPAddress* then it shall be converted to a domain name. If this cannot be done then LDS shall report an error.
port	The port maps onto the SRV record port field.
path	The path maps onto the path string in the TXT record (see Table 69).

Suitable default values should be chosen for fields in a SRV record that do not have a mapping specified in Table 70. e.g. TTL=86400, class=IN, priority=0, weight=5

Annex D
(normative)

Server Capability Identifiers

Clients benefit if they have more information about a *Server* before they connect, however, providing this information imposes a burden on the mechanisms used to discover *Servers*. The challenge is to find the right balance between the two objectives.

ServerCapabilityIdentifiers are the way this specification achieves the balance. These identifiers are short and map onto a subset of OPC UA features which are likely to be useful during the discovery process. The identifiers are short because of length restrictions for fields used in the mDNS specification. Table 71 is a non-normative list of possible identifiers.

Table 71 – Examples of *ServerCapabilityIdentifiers*

Identifier	Description
NA	No capability information is available. Cannot be used in combination with any other capability.
DA	Provides current data.
HD	Provides historical data.
AC	Provides alarms and conditions that may require operator interaction.
HE	Provides historical alarms and events.
GDS	Supports the Global Discovery Server information model.
LDS	Only supports the Discovery Services. Cannot be used in combination with any other capability.
DI	Supports the Device Integration (DI) information model (see DI).
ADI	Supports the Analyser Device Integration (ADI) information model (see ADI).
FDI	Supports the Field Device Integration (FDI) information model (see FDI).
FDIC	Supports the Field Device Integration (FDI) Communication Server information model (see FDI).
PLC	Supports the PLCopen information model (see PLCopen).
S95	Supports the ISA95 information model (see ISA-95).
RCP	Supports the reverse connect capabilities defined in Part 6.
PUB	Supports the *Publisher* capabilities defined in Part 14.
SUB	Supports the *Subscriber* capabilities defined in Part 14.

The normative set of *ServerCapabilityIdentifiers* can be found here:

http://www.opcfoundation.org/UA/schemas/1.04/ServerCapabilities.csv

Application developers shall always use the linked CSV.

Client applications that support the PUB or SUB capability can send or receive PubSub Messages but do not support the PubSub information model.

Client applications that support the RCP capability allow *Servers* to connect, however, they do not support *GetEndpoints Service.*

Annex E
(normative)

DirectoryServices

E.1 Global Discovery via Other Directory Services

Many organizations will deploy *DirectoryService*s such as LDAP or UDDI to manage resources available on their network. A *Client* can use these services as a way to find *Servers* by using APIs specific to *DirectoryService* to query for UA *Servers* or UA *DiscoveryServers* available on the network. The *Client* would then use the URLs for *DiscoveryEndpoints* stored in the *DirectoryService* to request the *EndpointDescriptions* necessary to connect to an individual servers

Some implementations of a *GlobalDiscoveryServer* will be a front-end for a standard *Directory Service*. In these cases, the *QueryServers* method will return the same information as the *DirectoryService* API. The discovery process for this scenario is illustrated in Figure 30 .

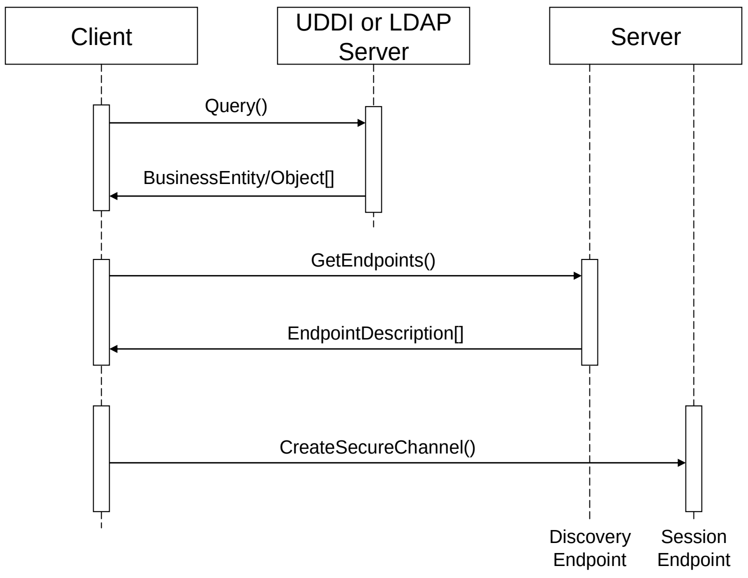

Figure 30 – The UDDI or LDAP Discovery Process

E.2 UDDI

UDDI registries contain *businessEntities* which provide one or more *businessServices*. The *businessServices* have one or more *bindingTemplates*. *bindingTemplates* specify a physical address and a *Server* Interface (called a tModel). Figure 31 illustrates the relationships between the UDDI registry elements.

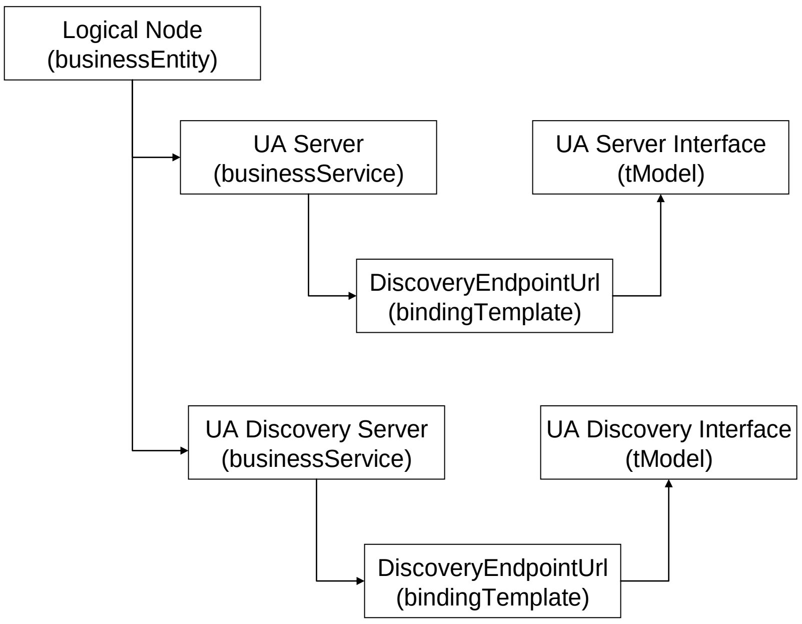

Figure 31 – UDDI Registry Structure

This specification defines standard tModels which shall be referenced by businessServices that support UA. The standard UA tModels shown in Table 72.

Table 72 – UDDI tModels

Name	domainKey	uuidKey
Server	uddi:server.ua.opcfoundation.org	uddi:AA206B41-EC9E-49a4-B789-4478C74120B5
DiscoveryServer	uddi:discoveryserver.ua.opcfoundation.org	uddi:AA206B42-EC9E-49a4-B789-4478C74120B5

The name of the businessService elements should be the same as the *ApplicationName* for the UA application. The serviceKey shall be the *ApplicationUri*. At least one bindingTemplate shall be present and the accessPoint shall be the URL of the *DiscoveryEndpoint* for the UA server identified by the serviceKey. Servers with multiple *DiscoveryEndpoints* would have multiple bindingTemplates

A UDDI registry will generally only contain UA servers, however, there are situations where the administrators cannot know what *Servers* are available at any given time and will find it more convenient to place a *DiscoveryServer* in the registry instead.

E.3 LDAP

LDAP servers contain *objects* organized into hierarchies. Each object has an *objectClass* which specifies a number of *attributes*. *Attributes* have values which describe an *object*. Figure 32 illustrates a sample LDAP hierarchy which contains entries describing UA servers.

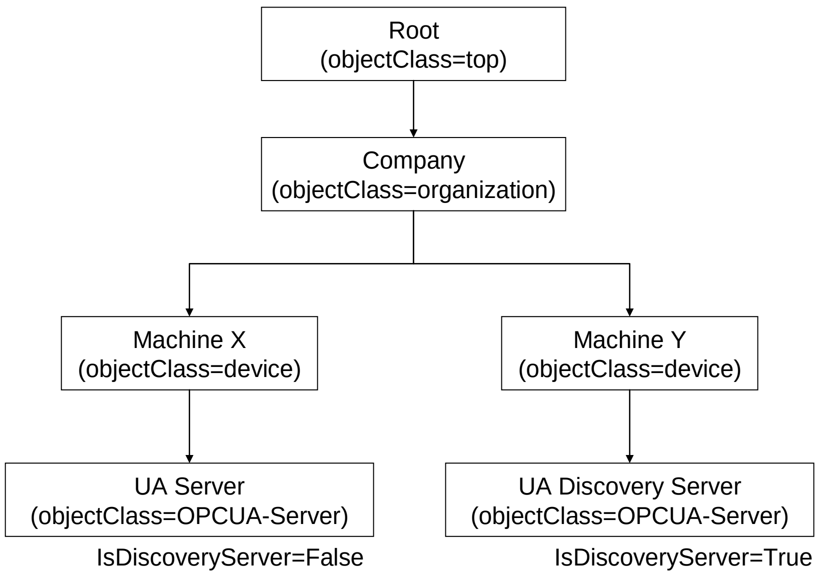

Figure 32 – Sample LDAP Hierarchy

UA applications are stored in LDAP servers as entries with the UA defined objectClasses associated with them. The schema for the objectClasses defined for UA are shown in Table 73.

Table 73 – LDAP Object Class Schema

Name	LDAP Name	Type	OID
Application	opcuaApplication	Structural	1.2.840.113556.1.8000.2264.1.12.1
ApplicationName	cn	String (Required)	Built-in
HostName	dNSName	String	Built-in
ApplicationUri	opcuaApplicationUri	Name	1.2.840.113556.1.8000.2264.1.12.1.1
ApplicationType	opcuaApplicationType	Boolean	1.2.840.113556.1.8000.2264.1.12.1.3
DiscoveryUrl	opcuaDiscoveryUrl	String, Multi-valued	1.2.840.113556.1.8000.2264.1.12.1.4

This OID is globally unique and can use used with any LDAP implementation.

Administrators may extend the LDAP schema by adding new attributes.

Annex F
(normative)

Local Discovery Server

F.1 Certificate Store Directory Layout

A recommended directory layout for *Applications* that store their *Certificates* on a file system is shown in Table 74. The Local Discovery Server shall use this structure.

This structure is based on the rules defined in Part 6.

Table 74 – Application Certificate Store Directory Layout

Path	Description
<root>	A descriptive name for the trust list.
<root>/own	The *Certificate* store which contains private keys used by the application.
<root>/own/certs	Contains the X.509 v3 *Certificates* associated with the private keys in the ./private directory.
<root>/own/private	Contains the private keys used by the application.
<root>/trusted	The *Certificate* store which contains trusted *Certificates*.
<root>/trusted/certs	Contains the X.509 v3 *Certificates* which are trusted.
<root>/trusted/crl	Contains the X.509 v3 CRLs for any *Certificates* in the ./certs directory.
<root>/issuer	The *Certificate* store which contains the CA *Certificates* needed for validation.
<root>/issuer/certs	Contains the X.509 v3 *Certificates* which are needed for validation.
<root>/issuer/crl	Contains the X.509 v3 CRLs for any *Certificates* in the ./certs directory.
<root>/rejected	The *Certificate* store which contains certificates which have been rejected.
<root>/rejected/certs	Contains the X.509 v3 *Certificates* which have been rejected.

All X.509 v3 certificates are stored in DER format and have a '.der' extension on the file name.

All CRLs are stored in DER format and have a '.crl' extension on the file name.

Private keys should be in PKCS #12 format with a '.pfx' extension or in the OpenSSL PEM format. The OpenSSL PEM format is not formally defined and should only be used by applications which use the OpenSSL libraries to implement security. Other private key formats may exist.

The base name of the Private Key file shall be the same as the base file name for the matching Certificate file stored in the ./certs directory.

A recommended naming convention is:

<CommonName> [<Thumbprint>].(der | pem | pfx)

Where the CommonName is the CommonName of the Certificate and the Thumbprint is the SHA1 hash of the certificate formatted as a hexadecimal string.

F.2 Installation Directories on Windows

The *LocalDiscoveryServer* executable shall be installed in the following location:

```
%CommonProgamFiles%\OPC Foundation\UA\Discovery
```

where %CommonProgamFiles% is the value of the *CommonProgamFiles* environment variable on 32-bit systems. On 64-bit systems the value of the *CommonProgamFiles(x86)* environment variable is used instead.

The configuration files used by the *LocalDiscoveryServer* executable shall be installed in the following location:

```
%CommonApplicationData%\OPC Foundation\UA\Discovery
```

where %CommonApplicationData% is the location of the application data folder shared by all users. The exact location depends on the operating system, however, under Windows 7 or later the common application data folder is usually C:\ProgramData.

The certificates stores used by the *LocalDiscoveryServer* shall be organized as described in F.1. The root of the certificates stores shall be in the following location:

```
%CommonApplicationData%\OPC Foundation\UA\pki
```

Annex G
(normative)

Application Installation Process

G.1 Provisioning with Pull Management

Applications that use Pull Management (see 7.2) to initialize their configuration need to know the location of the *CertificateManager* which they can use to request *Certificates* and download Trust Lists. This location may be auto-discovered via mDNS by looking for servers with the GDS capability (see Annex D) or by providing a URL via and out of band mechanism such as e-mail or a web page.

Once the location is known the *Application* can connect to the *CertificateManager* and establish a secure channel. This will require that the Application trust the *Certificate* provided by the *CertificateManager* even if it is not in the Application's *Trust List*. If there is an interactive user the Application should warn the user before proceeding. If there is no interactive user the Application should ensure the domain in the URL matches one of the domains in the *Certificate*.

In some cases, the Application distributor or installer will know the CA used to sign the *Certificate* used by the *CertificateManager* and can add this CA to the Application's *Trust List* during installation. If practical, this approach provides the best protection against accidental registration with rogue *CertificateManagers.*

After establishing a secure channel with the *CertificateManager*, the Application shall provide user credentials which allow it to register new applications and request new *Certificates.* The credentials may be provided by prompting a user or they may be one time use credentials delivered via some out of band mechanism such as a web site during the installation process.

For embedded systems it can be impractical to enter user credentials. As an alternative, a unique *ApplicationInstance Certificate* can be provided during manufacture and the *Certificate* or a unique identifier for the *Certificate* should be provided to the device installer. The installer would then register the unique identifier or *Certificate* with the *CertificateManager* which would allow the device to request a new *Certificate* by creating a Secure Channel with the manufacturer's *Certificate*.

Once an Application has received its first *Certificate* then the *Certificate* can be used in lieu of user credentials when the Application needs to renew its *Certificate* or update its Trust List.

G.2 Provisioning with the Push Management

Servers that use Push Management (see 7.3) to initialize their configuration shall have a default *Certificate* assigned before the Push Management process can start.

In addition, Servers shall go into a "provisioning state" that makes it possible for remote clients to update the security configuration via the *ServerConfiguration Object* (see 7.7.2). When a *Server* is in the "provisioning state" it should limit the available functionality.

Once a Server has been configured it automatically leaves the "provisioning state". This step is necessary to ensure that security is not compromised.

A possible workflow for implementing the "provisioning state" include:

1. A flag in the configuration file that defaults to ON;
2. Always allow Clients to connect securely if the *Trust List* is empty;
3. Connect to the Server and provide administrator credentials where:
    - Toggle a physical switch on the device which enables access for a short period or
    - Provide one-time use password specified via an out-of-band mechanism;

4. Provide a new Certificate (optional) and *Trust List*;
5. Set the configuration flag to OFF;

Subsequent updates to Trust Lists or *Certificates* can be allowed if the Client has a trusted *Certificate* and valid administrator credentials.

In some cases, the *Application* distributor or installer will know the CA used to sign the *Certificate* used by the *CertificateManager* and can add this CA to the Application's *Trust List* during installation. If practical, this approach provides the best protection against accidental configuration by malicious Clients.

If the device is automatically discovered by the *CertificateManager* the *CertificateManager* needs some way to ensure that the device belongs on the network. The manufacturer can provide a unique *ApplicationInstance Certificate* during manufacture and provide the serial numbers to the device installer. The installer would then register the serial number or *Certificate* with the *CertificateManager*. When the *CertificateManager* discovers the device it would check that the *Certificate* is for one of the pre-authorized devices and continue with automatic provisioning of the device.

G.3 Setting Permissions

If a Private Key is stored on a regular file system it shall be protected from unauthorized access. This is best done by setting operating system permissions on the private key file that deny read/write access to anyone who is not using an account authorized to run the *Application*.

In some cases, additional protection can be added by protecting the Private Key with a password. Saving Private Key passwords in files should be avoided. This mode may also work in conjunction with "smart cards" that use hardware to protect the Private Key.

In addition to the Private Key, *Applications* shall be protected from unauthorized updates to their *Trust List*. This can also be done by setting operating system permissions on the directory where the Trust List is stored that deny write access to anyone who is not using an account authorized to administer the *Application*.

Finally, *Applications* may depend on one or more configuration files and/or databases which tell them where there *Trust List* and Private Key can be found. The source of any security related configuration information shall be protected from unauthorized updates. The exact mechanism used to implement these protections depends on the source of the information.

Annex H
(informative)

Comparison with RFC 7030

H.1 Overview

RFC 7030 (Enrolment over Secure Transport or EST) defines a mechanism for the distribution of *Certificates* to devices. This appendix summarizes the capabilities provided by EST and how the same capabilities are provided by the *CertificateManager* defined in Clause 7.

H.2 Obtaining CA Certificates

In EST a web operation returns the CA certificates. In OPC UA the CA *Certificates* are returned when the *CertificateManager* client reads the *Trust List* assigned to the application from the *CertificateManager*. Prior to these operations the *Client* should verify that the server is authorized to provide CAs. Table 75 compares how EST clients verify the EST server with how *CertificateManager* clients verify a *CertificateManager*.

Table 75 – Verifying that a Server is allowed to Provide Certificates

EST	OPC UA
Compare the URL for the EST server with the HTTPS certificate returned in the TLS handshake.	Compare the URL for the *CertificateManager* with the OPC UA *Certificate* returned in *GetEndpoints*.
Preconfigure the client to trust the EST *Server's* HTTPS certificate.	Preconfigure the client by adding the *CertificateManager Certificate* to the client *Trust List*.
Manual approval of the CA *Certificate* after comparing the certificate with out of band information.	Manual approval of the *CertificateManager Certificate* after comparing the *Certificate* with out of band information.
Pre-shared credentials for use with certificate-less TLS.	No equivalent.

H.3 Initial Enrolment

In EST a web operation is used to enrol a client. The EST server authenticates and authorizes the EST client before allowing the operation to proceed. In OPC UA, a *Method* is used to request a *Certificate*. The *CertificateManager* also authenticates and authorizes the client before allowing the operation to proceed. Table 76 compares how EST servers verify the EST client with how a *CertificateManager* verifies a *CertificateManager* client.

Table 76 – Verifying that a Client is allowed to request Certificates

EST	OPC UA
TLS with a client certificate which is previously issued by the EST server.	The *CertificateManager* client has a previously certificate previously issued by the GDS.
TLS with a previously installed certificate which is trusted by the EST server.	The *CertificateManager* client has a certificate which is trusted by the *CertificateManager*.
Shared credentials distributed out of band which are used for certificate-less TLS.	No equivalent.
HTTPS username/password authentication.	The *CertificateManager* client provides appropriate user credentials when it establishes the session.

H.4 Client Certificate Reissuance

In EST a certificate issued by the EST server can be used to as an HTTPS client certificate. This can be used to authorize the re-issue of the certificate. In OPC UA a certificate issued by the GDS can be used to establish a secure channel. This would then allow the GDS client to request that the certificate be re-issued.

In both EST and OPC UA clients can fall back to the authentication mechanisms used for Initial Enrolment if it is not possible to use the current certificate to establish a secure channel with the server.

H.5 Server Key Generation

Both EST and OPC UA allow clients to request new private keys. Both specifications require that encryption be used when returning private key data.

EST allows clients to explicitly request that separate encryption be applied to the private key. The algorithms are specified in the CSR (certificate signing request).

OPC UA allows clients to password protect the key (which uses encryption), however, OPC UA does not allow the client to directly specify the algorithm used. That said, the envelope used to transport private keys can be specified with the *PrivateKeyFormat* parameter and the set of envelope formats supported by the *CertificateManager* is published in the Address Space. It is expected that the envelope format will specify the algorithms used either by explicitly encoding the algorithm within the envelope or as part of the definition of the envelope.

H.6 Certificate Signing Request (CSR) Attributes Request

EST allows the client to request the list of CSR attributes the EST server supports. The attributes can indicate what additional metadata the client can provide or the algorithms that will be used.

In OPC UA the CSR metadata required is fixed by the specification and there is no mechanism to publish extensions. Clients are free to include additional metadata in the CSR, however, the *CertificateManager* may ignore it.

There is no mechanism in OPC UA to publish the algorithms which need to be used for the CSR, however, the *CertificateManager* will reject CSRs that do not meet its requirements.