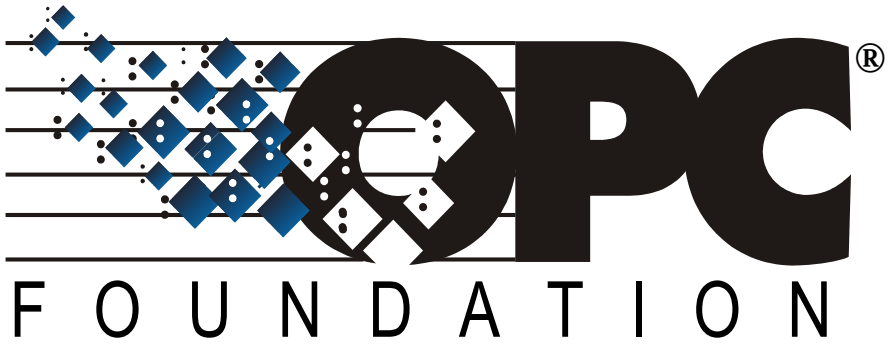

OPC 10000-12

OPC UA Specification

OPC Unified Architecture

Part 12: Discovery and Global Services

Release 1.05.04

2024-10-15

Specification Type	Industry Standard Specification	Comments:	
Document Number	OPC 10000-12		
Title:	OPC Unified Architecture Discovery and Global Services	Date:	2024-10-15
Version:	Release 1.05.04	Software Source:	MS-Word OPC 10000-12 - UA Specification Part 12 - Discovery and Global Services 1.05.04.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.

AGREEMENT OF USE

GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: http://www.opcfoundation.org/errata.

Revision 1.05.04 Highlights

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

Mantis ID	Scope	Summary	Resolution
7928	Feature	No way to get the certificate of a registered application that has a signed certificate from the GDS.	Added explaination on how to handle errors in 7.9.5.
8063	Clarification	Client connection management with GDS.	Clarification of client connection behaviour for use of global services in 6.3.
8685	Errata	UpdateApplication allows to change the ApplicationUri.	No longer allow updates to the ApplicationUri in 6.6.7.
8957	Errata	Explain chicken and egg problem when a Client connects to an unknown GDS.	Add requirements to prevent securit holes in 7.3.
9014	Clarification	Requirements for setting ApplicationType CLIENTANDSERVER.	Added detailed requirements to 7.9.3.
.9048	Clarification	CertificateManagerEndpoint term not defined.	Changed text to “All associations with CertificateManagers are deleted” in 7.10.11.
9059	Clarification	The mDNS section needs more explaination (both Facet/CU and Part 12).	Updated wording in 4.2.3.
9195	Clarification	Define more details about recommended behaviour for Application Setup state and TOFU.	Add rules for SecurityAdmin configuration to G.2.
9247	Clarification	Definition of “normal integrity checks” for the ServerConfiguration UpdateCertificate method.	Added explicit reference to Part 4 in 7.10.5.
9383	Clarification	Missing closing of a secure channel after TrustList update.	Added SecureChannel to the list of actions taken in 7.8.2.3.
9440	Clarification	CredentialSecret format for UserName/Password is not clear.	Explicitly state that the credentialId is a user name in 8.5.6.
9497	Errata	Clients without RCP cannot register.	Removed requirement for RCP support in 6.6.5.
9506	Clarification	Clarifications and fixes for ApplyChanges.	Updated codes for ApplyChanges.
9507	Clarification	No transaction related results for CreateSigningRequest and TrustList.	Updated codes for CreateSigningRequest, UpdateCertificate and TrustList::Open Updated codes for all methods that requires signing/encryption.
9510	Feature	Need to define ServerCapabilities for REGISTRAR (Pull) and DCS waiting for Config (Push).	Added REGISTRAR and DCA to ServerCapabilities in Annex D.
9512	Feature	Make ApplicationConfigurationType defined in Part 21 available in Part 12.	Added **7.10.12**.
9526	Clarification	Clarification for configuration transaction support.	Added 7.10.2 and SupportsTransactions Property in 7.10.4.
9566	Clarification	Invalid links to references.	Updated links to reference documents in 2.
9830	Clarification	0009830: Inconsistency between figures for push order.	Updated Figure 15.
9856	Clarification	Exceeding the MaxTrustListSize Limit when modifying a TrustList	Added new error code to CloseAndUpdate in 7.8.2.3
9854	Clarification	No TrustList validation for RemoveCertificate()	Added new error code to RemoveCertificate in 7.8.2.5.
9853	Clarification	State whether TrustLists can be opened multiple times (concurrently) for reading or not	Deleted the *Bad_InvalidState* code from the *OpenWithMasks Method* in 7.8.2.2.

Mantis ID	Scope	Summary	Resolution
9852	Clarification	Potential deadlock situation when manipulating TrustLists via File-API	Added a requirement to report an error if a transaction on another session exists when CloseAndUpdate is called in 7.8.2.3.

.

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 *AuthorizationServices*.

2 Normative references

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

OPC 10000-1, OPC Unified Architecture - Part 1: Overview and Concepts
http://www.opcfoundation.org/UA/Part1/

OPC 10000-2, OPC Unified Architecture - Part 2: Security Model
http://www.opcfoundation.org/UA/Part2/

OPC 10000-3, OPC Unified Architecture - Part 3: Address Space Model
http://www.opcfoundation.org/UA/Part3/

OPC 10000-4, OPC Unified Architecture - Part 4: Services
http://www.opcfoundation.org/UA/Part4/

OPC 10000-5, OPC Unified Architecture - Part 5: Information Model
http://www.opcfoundation.org/UA/Part5/

OPC 10000-6, OPC Unified Architecture - Part 6: Mappings
http://www.opcfoundation.org/UA/Part6/

OPC 10000-7, OPC Unified Architecture - Part 7: Profiles
http://www.opcfoundation.org/UA/Part7/

OPC 10000-9, OPC Unified Architecture - Part 9: Alarms and Conditions
http://www.opcfoundation.org/UA/Part9/

OPC 10000-14, OPC Unified Architecture - Part 14: PubSub
http://www.opcfoundation.org/UA/Part14/

OPC 10000-17, OPC Unified Architecture – Part 17: Alias Names
http://www.opcfoundation.org/UA/Part17/

OPC 10000-20, OPC Unified Architecture – Part 20: File Transfer
http://www.opcfoundation.org/UA/Part20/

OPC 10000-21, OPC Unified Architecture – Part 21: Device Onboarding
http://www.opcfoundation.org/UA/Part21/

OPC 10000-100, OPC UA Specification: Part 100 - Devices
http://www.opcfoundation.org/UA/Part100/

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

https://www.rfc-editor.org/rfc/rfc3927

DNS-Name, Domain Names – Implementation and Specification

https://www.rfc-editor.org/rfc/rfc1035

DHCP, Dynamic Host Configuration Protocol

https://www.rfc-editor.org/rfc/rfc2131

mDNS, Multicast DNS

https://www.rfc-editor.org/rfc/rfc6762

DNS-SD, DNS Based Service Discovery

https://www.rfc-editor.org/rfc/rfc6763

RFC 5958, Asymmetric Key Packages

https://www.rfc-editor.org/rfc/rfc5958

PKCS #10, Certification Request Syntax Specification

https://www.rfc-editor.org/rfc/rfc2986

PKCS #12, Personal Information Exchange Syntax v1.1

https://www.rfc-editor.org/rfc/rfc7292

RFC 7030, Enrollment over Secure Transport

https://www.rfc-editor.org/rfc/rfc7030

DI, OPC Unified Architecture for Devices (DI)

https://opcfoundation.org/documents/10000-100/

ADI, OPC Unified Architecture for Analyzer Devices (ADI)

https://opcfoundation.org/documents/10020/

PLCopen, OPC Unified Architecture / PLCopen Information Model

https://opcfoundation.org/documents/30000/

FDI, OPC Unified Architecture for FDI

https://opcfoundation.org/documents/30080/

ISA-95, ISA-95 Common Object Model

https://opcfoundation.org/documents/10030/

X.500, ISO/IEC 9594-1:2017 – The Directory Part 2: Overview of concepts

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

IEEE 802.1AR, IEEE Std 802.1AR-2018 – Secure Device Identity

https://standards.ieee.org/standard/802_1AR-2018.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 OPC 10000-1, OPC 10000-2, OPC 10000-3, OPC 10000-4, OPC 10000-6 and OPC 10000-9 apply.

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

3.1.2
CertificateGroup
a context used to manage the *TrustList* and *Certificate(s)* associated with *Applications or Users*.

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

Note 1 to entry: Devices have hardware-based mechanisms, such as a TPM, to protect Private Keys.

3.1.4
ClientUrl
a physical address available on a network that allows *Servers* to initiate a reverse connection.

3.1.5
DirectoryService

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

3.1.6
DiscoveryServer
an *Application* that maintains a list of *OPC UA Applications* that are available on the network and provides mechanisms for other *OPC UA Applications* to obtain this list.

3.1.7
DiscoveryUrl
a URL for a network *Endpoint* that provides the information required to connect to a *Client* or *Server*.

3.1.8
GlobalDiscoveryServer (GDS)

a *Server* that provides numerous services related to discovery and security management.

Note 1 to entry: a GDS may also be a *CertificateManager*.

Note 2 to entry: a GDS may also be a *KeyCredentialService*.

Note 3 to entry: a GDS may also be a *AuthorizationService*.

3.1.9
GlobalService

a *Server* that provides centrally managed capabilities needed for a system.

Note 4 to entry: a *GlobalDiscoveryServer*, a *CertificateManager*, a *KeyCredentialService* and an *AuthorizationService* are all examples of *GlobalServices.*

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
KeyCredential

a unique identifier and a secret used to access an *AuthorizationService* or a *Broker*.

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

3.1.12
KeyCredentialService
a software application that provides *KeyCredentials* needed to access an *AuthorizationService* or a *Broker*.

3.1.13
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.

3.1.14
LocalDiscoveryServer-ME (LDS-ME)
a *LocalDiscoveryServer* that includes the MulticastExtension.

3.1.15
MulticastExtension
an extension to a *LocalDiscoveryServer* that adds support for the mDNS protocol.

3.1.16
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.

3.1.17
Privilege

a named set of rights which cannot be expressed as *Permissions* granted on *Nodes.*

Note 1 to entry: For example, a *Privilege* can be defined when the right to call a *Method* depends on the parameters passed to the *Method*.

Note 5 to entry: a *Privilege* is a document convention that does not appear in the *Server AddressSpace*.

3.1.18
PullManagement

a workflow where a *Client* manages its configuration by using a *GlobalService.*

3.1.19
PushManagement

a workflow where a *GlobalService* manages a *Server's* configuration.

3.1.20
ServerCapabilityIdentifier

a short identifier which uniquely identifies a set of discoverable capabilities supported by an *OPC UA Application*.

Note 1 to entry: the list of the currently defined *CapabilityIdentifiers* 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
HTTP	Hypertext Transfer Protocol
IANA	The Internet Assigned Numbers Authority
JWT	JSON Web Token
LDAP	Lightweight Directory Access Protocol
LDS	Local Discovery Server
LDS-ME	Local Discovery Server with the Multicast Extension
mDNS	Multicast Domain Name System
MQTT	Message Queuing Telemetry Transport
NAT	Network Address Translation
OCSP	Online Certificate Status Protocol
PEM	Privacy Enhanced Mail
PFX	Personal Information Exchange
PKCS	Public Key Cryptography Standards
RSA	Rivest–Shamir–Adleman
SHA1	Secure Hash Algorithm
SSL	Secure Socket Layer
TLS	Transport Layer Security
TPM	Trusted Platform Module
UA	Unified Architecture
UDDI	Universal Description, Discovery and Integration

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 OPC 10000-4. Discoverable applications are generally *Servers*, however, some *Clients* will support reverse connections as described in OPC 10000-6 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 OPC 10000-4. 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 are visible for local discovery (see 4.3.3) and *MulticastSubnet* discovery if the LDS is a LDS-ME (see 4.3.4).

The OPC UA Standard (OPC 10000-4) defines a *RegisterServer2 Service* which provides additional registration information. *All Applications* and *LocalDiscoveryServer* 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. CapabilityIdentifiers* are short, string identifiers of well-known OPC UA features. *Applications* can use these identifiers as a filter during discovery.

The set of known *CapabilityIdentifiers* 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.

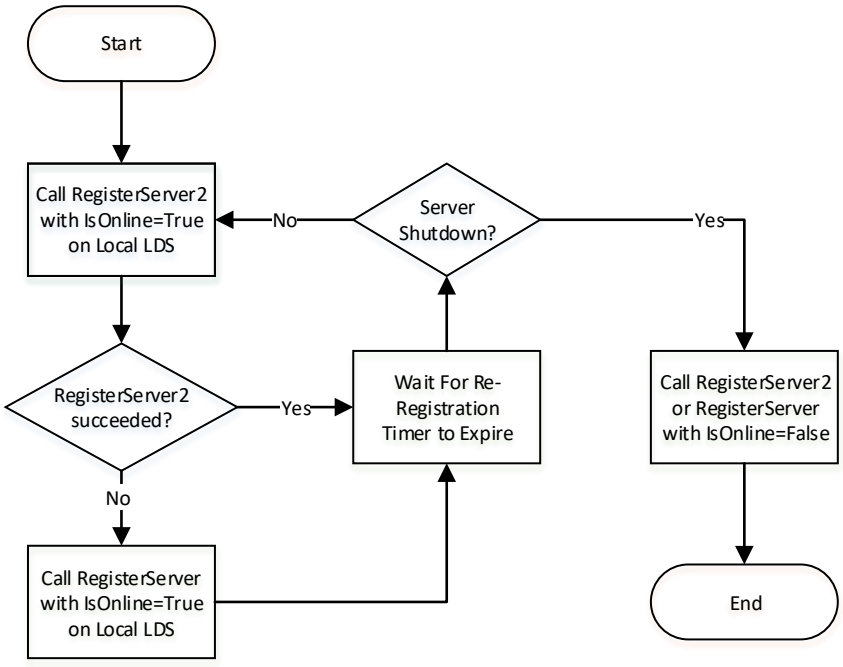

Figure 1 – The Registration Process with an LDS

See OPC 10000-4 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. If implementing an LDS-ME, 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. In addition they shall implement additional OPC UA specific items described in Annex C. 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 OPC 10000-4) 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 4.3.2).

Clients should be aware of rogue *DiscoveryServers* that might direct them to rogue *Servers*. That said, this problem is mitigated when a *Client* connects to a *Server* and verifies that it trusts the *Server*. In addition, the *CreateSession Service* returns parameters that allow *Client* to verify that the previously acquired results from a LDS have not been altered. See OPC 10000-2 and OPC 10000-4 for a detailed discussion of these issues.

A similar potential for a rogue GDS exists if the *Client* has not been configured to trust the GDS *Certificate* or if the *Client* does not use security when connecting to the GDS. Note that a *Client* that uses security but automatically trusts a GDS *Certificate* is not protected from a rogue GDS even though the connection itself is secure. This problem is also mitigated by verifying trust whenever a *Client* connects to a *Server* discovered via the GDS.

4.3.2 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 OPC 10000-4*.*

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

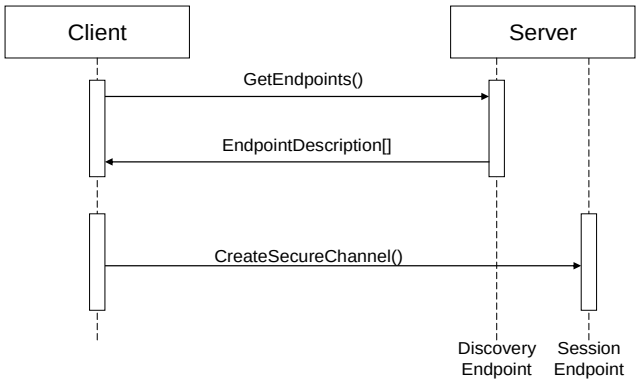

Figure 2 – The Simple Discovery Process

4.3.3 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 OPC 10000-6.

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.

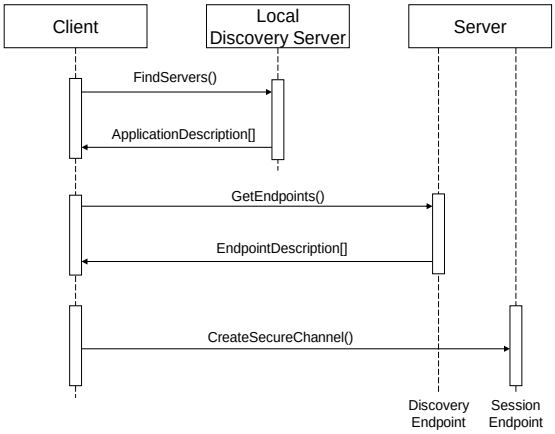

Figure 3 – The Local Discovery Process

4.3.4 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.

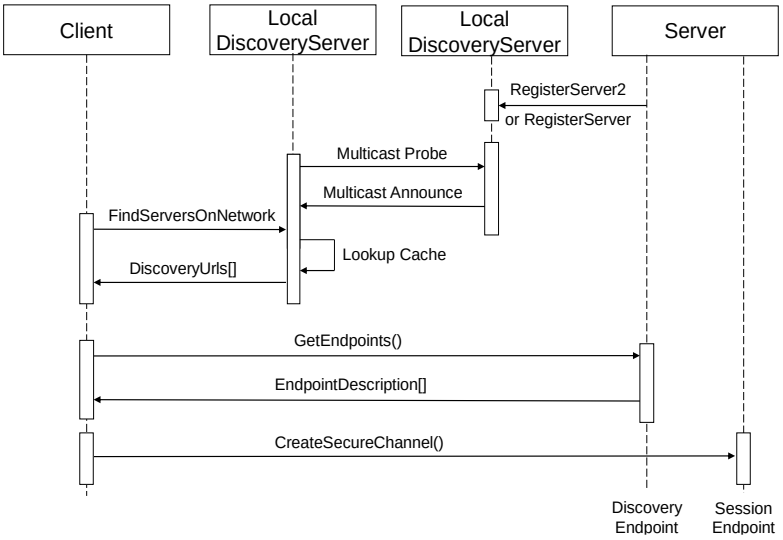

Figure 4 – The MulticastSubnet Discovery Process

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 *CapabilityIdentifiers* 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.

4.3.5 Global Discovery

A GDS is an OPC UA *Server* which allows *Clients* to search for *Servers* within the administrative domain of the GDS. It provides *Methods* that allow applications to search for other applications (see 6). To access the GDS, the *Client* uses the *Call* service to invoke the *QueryApplications Method* (see 6.6.11) to retrieve a list of *Servers* that meet the filter criteria provided. The

QueryApplications 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.

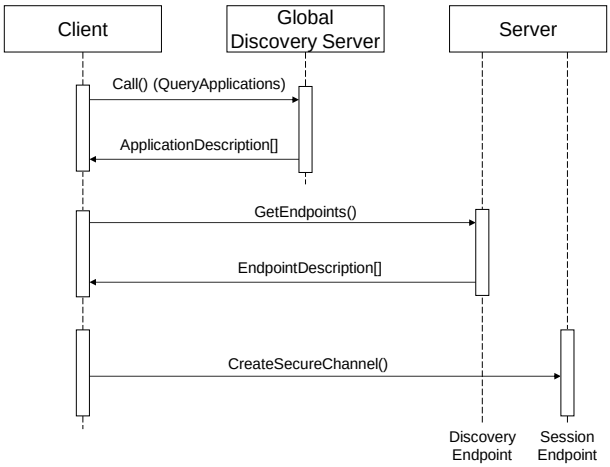

Figure 5 – The Global Discovery Process

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.

4.3.6 Combined Discovery Process for Clients

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

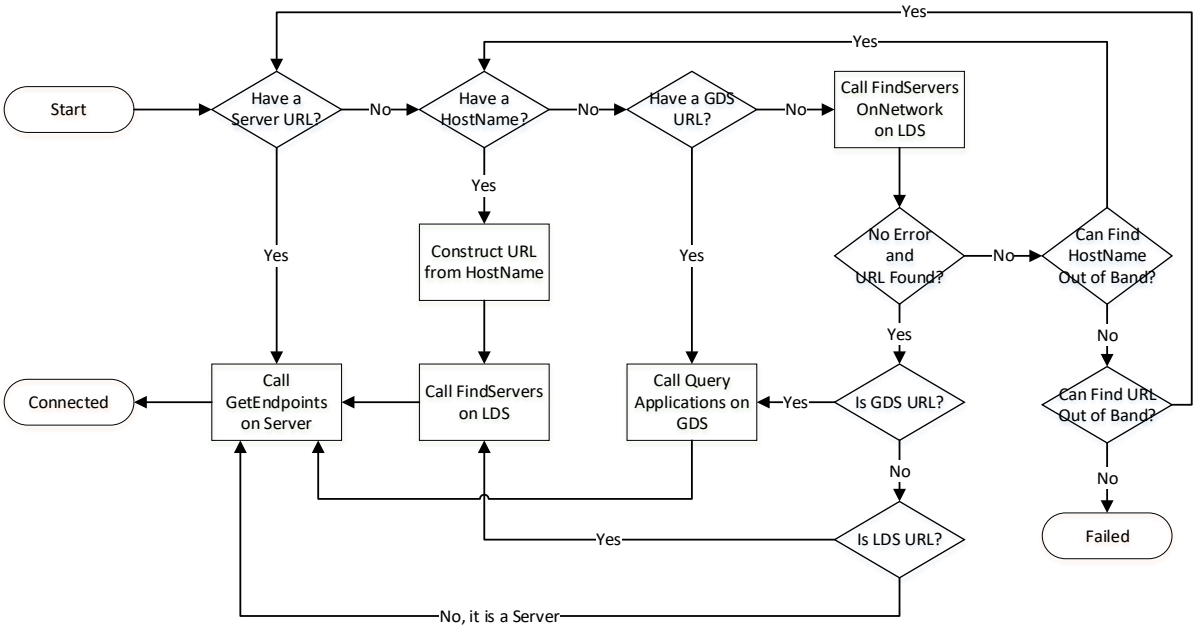

Figure 6 – The Discovery Process for Clients

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.

4.4 The Discovery Process for Reverse Connections

4.4.1 Overview

The discovery process for reverse connect does not serve the same purpose as the discovery process for normal connections because reverse connections require the *Server* to be configured to automatically attempt to connect to the *Client* and the *Client* to be configured so it knows what to do with the *Server* when it receives the connection. The limited mechanisms discussed here may help *SecurityAdmins* with the configuration of *Servers*.

A *SecurityAdmin* tasked with configuring *Servers* needs to determine the *ClientUrls* for *Clients* that support reverse connect.

The following choices are available:

- Out-of-band discovery (i.e. entry into a GUI) of a *ClientUrl* for a *Client*;
- Searching for *Clients* known to a *GlobalDiscoveryServer*.

The mechanisms based on an LDS are not available since *Clients* do not register with the LDS.

4.4.2 Out-of-band Discovery

Every *Client* that supports reverse connect has one or more *ClientUrls* that allow *Servers* to connect. Once the *SecurityAdmin* acquires the *ClientUrl* via an out-of-band mechanism, it can configure the *Server* to use it.

4.4.3 Global Discovery for Reverse Connections

A GDS is a *Server* which allows other *SecurityAdmins* to search for *Clients* that support reverse connnect within the administrative domain of the GDS. The *SecurityAdmin* uses the *Call* service to invoke the *QueryApplications Method* (see 6.6.11) with “RCP” as a *serverCapabilityFilter* to get a list of *Clients* that support reverse connect from the GDS.

The discovery process is illustrated in Figure 5.

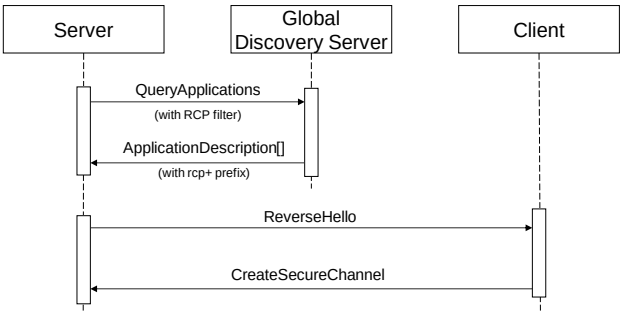

Figure 7 – The Global Discovery Process for Reverse Connections

The *ClientUrls* are returned in the *DiscoveryUrls* parameter of the *ApplicationDescription* record and have the ‘rcp+’ prefix. *DiscoveryUrls* without the prefix are used for forward connections. Once the *SecurityAdmin* has a *ClientUrl* it can configure the *Server* to use it.

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 OPC 10000-4. In addition, the *LocalDiscoveryServer* shall provide at least one *Endpoint* which implements the *RegisterServer* service for these applications.

The *FindServers Service* returns the information for the *LocalDiscoveryServer* and all *Servers* that are known to the LDS.The *GetEndpoints Service* returns the *EndpointDescriptions* for the *LocalDiscoveryServer* that allow *Servers* to call the *RegisterServer* or *RegisterServer2 Services*. The *LocalDiscoveryServer* does not support *Sessions* so information needed for establishing *Sessions*, such as supported *UserTokenPolicies*, is not provided.

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 OPC 10000-4. For backward compatibility a *LocalDiscoveryServer* also supports the *RegisterServer Service* which is defined in OPC 10000-4.

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 OPC 10000-4.

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 capabilities.

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.

5.3 Network Architectures

5.3.1 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.*

5.3.2 Single MulticastSubnet

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

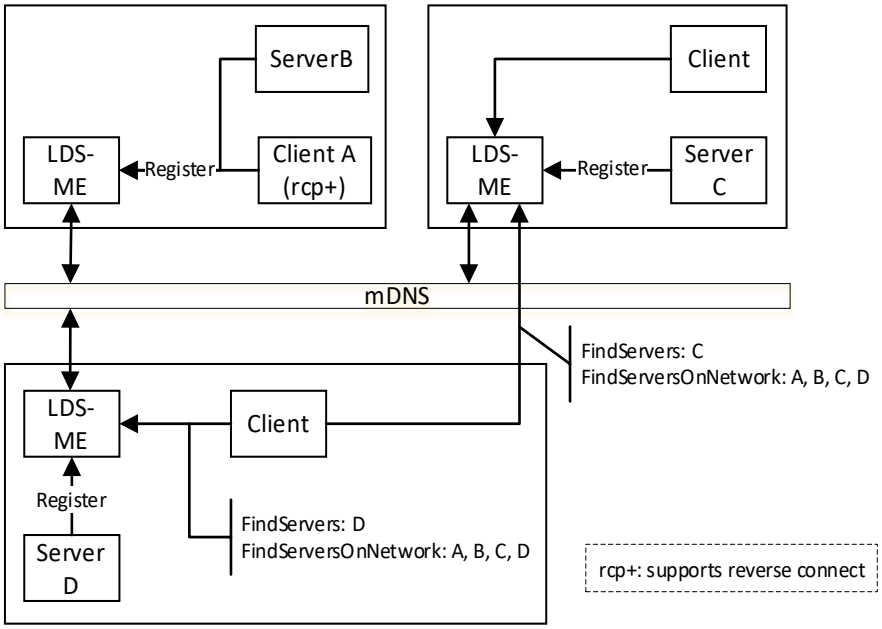

Figure 8 – The Single MulticastSubnet Architecture

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.

5.3.3 Multiple MulticastSubnet

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

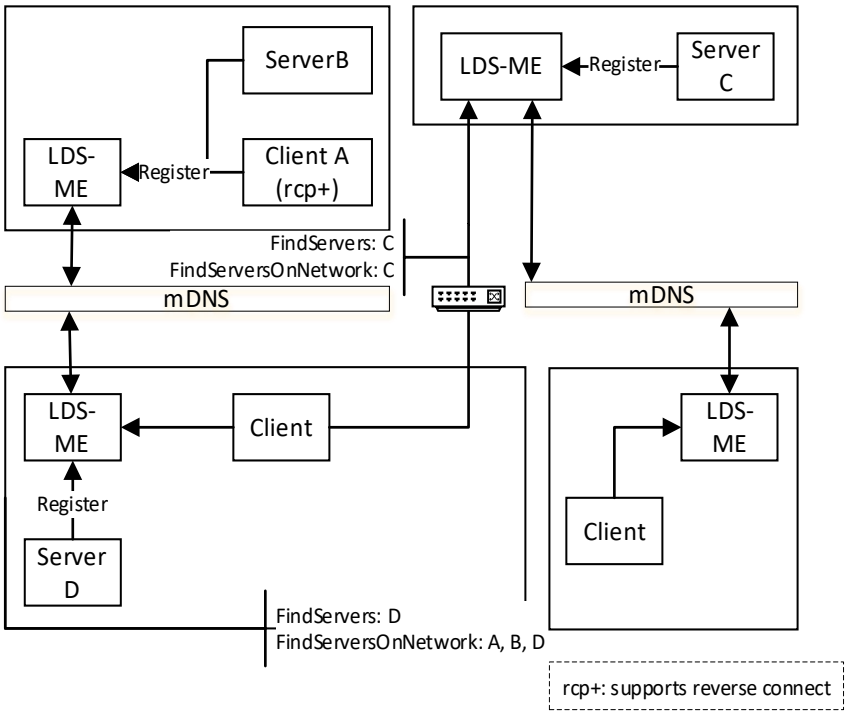

Figure 9 – The Multiple MulticastSubnet Architecture

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.*

5.3.4 No MulticastSubnet

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

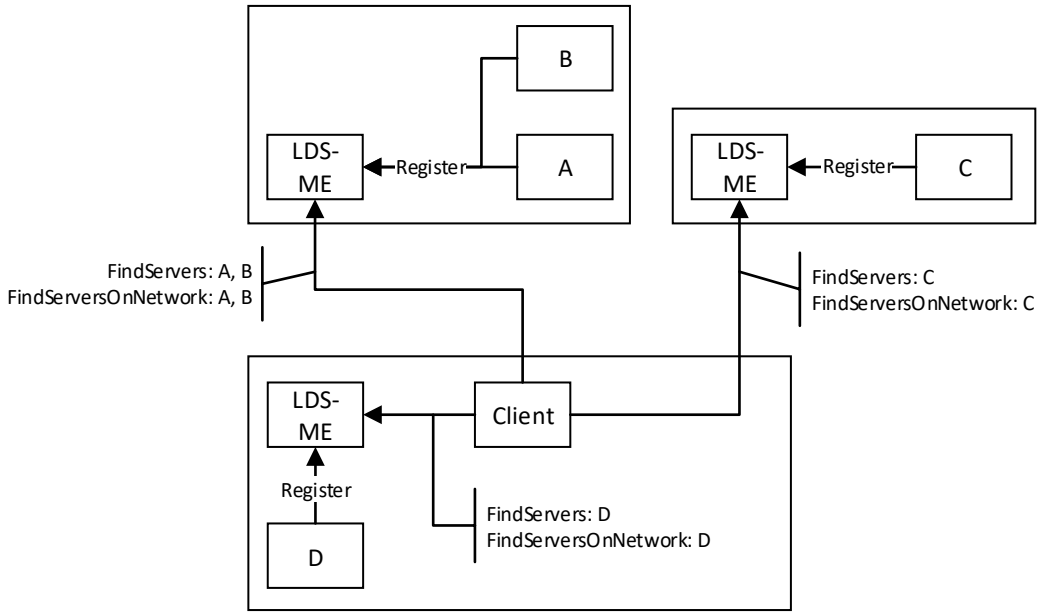

Figure 10 – The No MulticastSubnet Architecture

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 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. *Clients* could also use a GDS if one is available.

5.3.5 Domain Names and MulticastSubnets

The mDNS specification requires that fully qualified domain name be announced 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 considered to be unique within the subnet where the domain name was used. This means *Clients* need to be aware that URLs received from any LDS-ME other than the one on the *Client's* computer could contain 'local' domains which are not reachable or will connect to a different computer 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 computer.

System administrators can eliminate this problem by configuring a normal DNS with the fully qualified domain names for all computers 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 append the 'local' top level domain to any domains declared in their *Certificate*; an unqualified domain name is used if a more appropriate qualifer does not exist. *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 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* within the administrative domain. When compared to the LDS, the GDS provides an authorative source for *Servers* which have been verified by administrators and accessed via a secure communication channel.

The GDS provides *Methods* that allow administrators to register applications and allow applications to search for other applications.

Some GDS implementations may provide a front-end to an existing *DirectoryService* such as LDAP (see Annex E). By standardizing on an OPC UA based interface, *Clients* do not need to have knowledge of different *DirectoryServices.*

6.2 Roles and Privileges

GlobalDiscoveryServers restrict access to many of the features they provide. These restrictions are described either by referring to well-known *Roles* which a *Session* must have access to or by referring to *Privileges* which are assigned to *Sessions* using mechanisms other than the well-known *Roles*. The well-known *Roles* used in for a GDS are listed in Table 1.

Table 1 – Well-known Roles for a GDS

Name	Description
DiscoveryAdmin	This *Role* grants rights to register, update and unregister any *OPC UA Application*.
SecurityAdmin	This *Role* grants the right to change the security configuration of a GDS.

The *Privileges* used in for a GDS are listed in Table 2.

Table 2 – Privileges for a GDS

Name	Description
ApplicationSelfAdmin	This *Privilege* grants an *OPC UA Application* the right to update its own registration. The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application.*
ApplicationAdmin	This *Privilege* grants rights to update one or more registrations. The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application* and what the set of registrations it is authorized to update.

6.3 Client connections to global services

A *GlobalDiscoveryServer* is an OPC UA *Server* implementing different global services for discovery, *Certificate* management, user or PubSub key management, user authorization, software and device management.

The number of OPC UA *Applications* using the different services as OPC UA *Client* may be huge and the OPC UA *Server* is most likely not able to handle connections from all OPC UA *Clients* at the same time. Therefore an OPC UA *Client* connected to a GDS should minimize the time it is connected to the GDS to the currently required actions. The OPC UA *Client* shall

disconnect as soon as it completes the sequence of actions needed to interact with the services. The OPC UA *Clients* shall not keep connections open between the execution of sequences.

A GDS OPC UA *Server* is allowed to close *Sessions* with OPC UA Clients not authenticated as one of the GDS administrative *Roles* if it runs out of connection resources. If the GDS needs to close *Sessions*, it should first close *Sessions* without GDS management *Privileges*. Otherwise it may close the *Session* that was inactive for the longest time not using GDS global services e.g. *Method* calls.

It is also recommended to use a short maximum session timeout on the GDS OPC UA *Server*.

Actions performed cyclically by OPC UA *Applications* during *PullManagement* shall start the second cycle with a random delay that is between one and at least ten percent of the cycle period.

6.4 Local Discovery

If an administrator (e.g. a *Client* with access to the *DirectoryAdmin Role*) registers a *LocalDiscoveryServer* with the GDS, then the GDS periodically adds *Servers* to a list for review by calling *FindServersOnNetwork* or *FindServers* on the LDS. Figure 11 shows the relationship between a GDS and the LDS-ME or LDS.

The GDS shall not make *Servers* discovered in this manner available via *QueryApplications* for *FindApplications* before an administrator has approved the *Server* on a case by case basis or via automated rules. Note that auto-population can result in conflicts where multiple *Servers* have the same *ApplicationUri* due to a configuration error. A GDS should keep track of these conflicts so an administrator can review and resolve them.

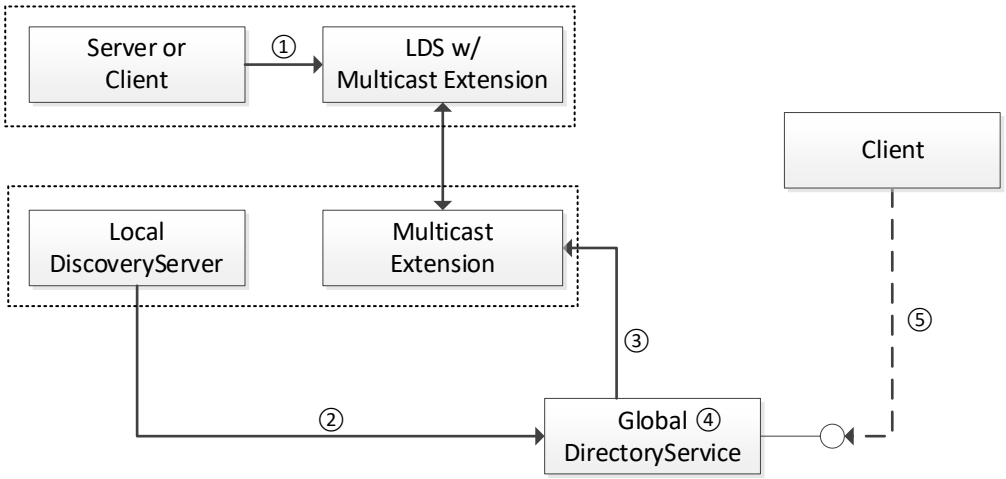

Figure 11 – The Relationship Between GDS and other components

The steps shown in Figure 11 are:

1	The Server calls *RegisterServer2* on the LDS running on the same computer.
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 *DiscoveryAdmin.*

5	The *Client* calls *QueryApplications Method* on the GDS to discover applications.

The *Information Model* used for registration and discovery is shown in 6.6.1**.** Any *Client* shall be able to call the *QueryApplications Method* to find applications known to GDS. The complete definitions for each of the types used are described in 0.

Once a *Server* is registered with the GDS the record does not dissappear if the *Server* goes offline. If a *Server* is permentently taken offline the administrator needs to manually remove the registration.The interactions described above apply to *Servers* automatically discovered via an LDS. *Servers* can also be discovered by another application and registered automatically with the GDS provided the other application has the necessary administrative rights on the GDS.

6.5 Application Registration Workflow

The OPC UA *Application* or the *Application* configuration tool connects to the GDS for initial installation with GDS including *Application* registration. This requires a user that has the *DiscoveryAdmin Role* or *the ApplicationAdmin Privilege*.

The workflow for the *Application* registration is shown in Figure 12.

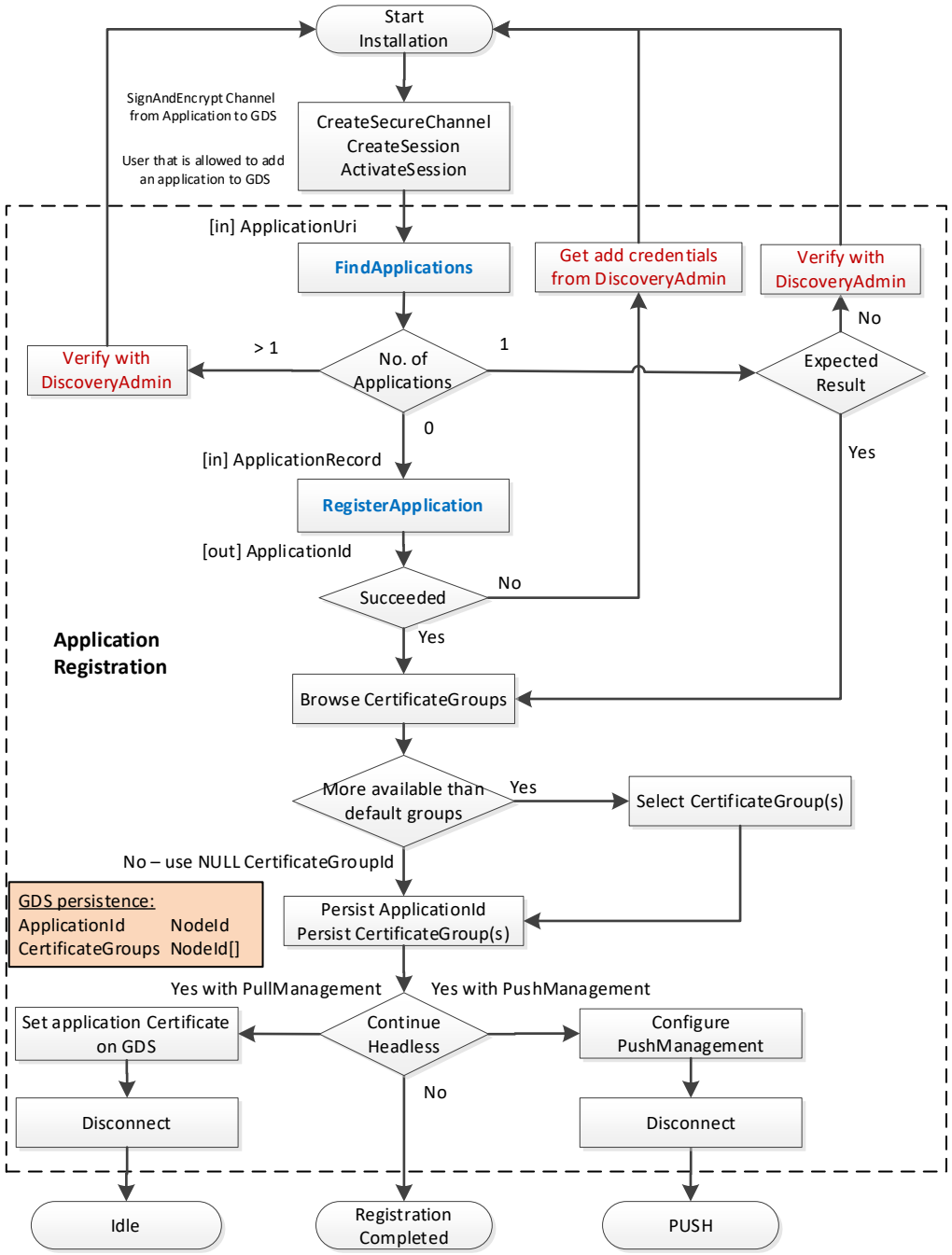

Figure 12 – Application Registration Workflow

The description of the *Application* registration workflow steps is provided in Table 3.

Table 3 – Application Registration Workflow Steps

Step	Description
Application installation	The registration of an application with a GDS is normally executed as part of the initial installation and configuration of the application. It can be executed by a configuration tool that is part of the application or by a generic GDS configuration tool.
Connect	For the connection management with the GDS the services *OpenSecureChannel*, *CreateSession* and *ActivateSession* are used to create a connection with *MessageSecurityMode SignAndEncrypt* and a user that has the permission to register applications with the GDS. If the user does not have sufficient rights, the GDS can provide a mechanism to accept registrations on the GDS side before they are visible to *Clients* through *QueryApplications*.
FindApplications	The first step after connect is to check if there is already a registration available for the *ApplicationUri*. The *DirectoryType Method FindApplications* is used to pass the *ApplicationUri* of the application to the GDS. The Method returns an array of application records where the size of the array defines the next steps. • If the array is empty, the next step is *RegisterApplication*. • If the array size is one, and the record matches the expected application record, the next step is *Browse CertificateGroups*. • If the array size is one and the record does not match the expected application record, the registration must be verified with a *DiscoveryAdmin*. • If the array size is more than one, this indicates a fatal error and the status must be verified with a *DiscoveryAdmin*.
RegisterApplication	The *DirectoryType Method RegisterApplication* is used to pass in an application record with the application information. If the *Method* succeeds an *ApplicationId* is returned. This *ApplicationId* should be persisted for further interaction with the GDS regarding this application. If the *Method* fails, a *DiscoveryAdmin* is needed to identify and correct the issue. Typical errors include insufficient rights or conflicts with other application records.
Browse CertificateGroups	The *Browse Service* is used to get the list of GDS managed *CertificateGroups* by browsing the *CertificateGroups Folder* of the *Directory Object*. If more than one *CertificateGroup* is returned, the user selects the relevant *CertificateGroups* needed for the application. The selected *CertificateGroupIds* should be persisted together with the *ApplicationId*.
Registration end options	The following options are possible to complete the registration with the *CertificateManager*: 1. Continue with *PullManagement* using the existing connection to the GDS. This option is typically used by *Clients* executing the registration in an interactive mode for their own identity. See 7.6 for the *PullManagement* workflow. 2. Continue with *PullManagement* inside a headless application. 3. Continue with *PushManagement*.
Set application *Certificate* on GDS	For option (2) the current application *Certificate* must be configured for the application on the GDS to allow *Application* authentication for the initial *PullManagement* sequence. This configuration in the GDS is currently not in the scope of this specification.
Configure *PushManagement*	For option (3) the application must be configured for *PushManagement* in the *CertificateManager*. The configuration of the *PushManagement* in the *CertificateManager* is currently not in the scope of this specification.
Disconnect	For options (2) and (3) the configuration tool disconnects from the GDS.

6.6 Information Model

6.6.1 Overview

The *GlobalDiscoveryServer Information Model* used for *discovery* is shown in Figure 13. Most of the interactions between the *GlobalDiscoveryServer* and *Application* administrator or the *Client* will be via *Methods* defined on the *Directory* folder.

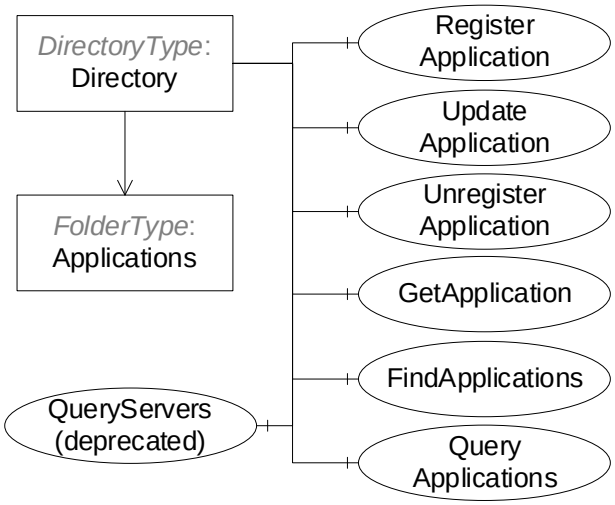

Figure 13 – The Address Space for the GDS

6.6.2 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 OPC 10000-5. 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 can exist for information models, or for optional services or for various locations in an administrative domain. It is defined in Table 4.

Table 4 – Directory Object Definition

Attribute	**Value**				
BrowseName	2:Directory				
TypeDefinition	2:DirectoryType defined in 6.6.3.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Conformance Units					
GDS Application Directory					

6.6.3 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 5.

Table 5 – DirectoryType Definition

Attribute	**Value**				
BrowseName	2:DirectoryType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*FolderType* defined in OPC 10000-5.					
0:HasComponent	Object	2:Applications	-	0:FolderType	Mandatory
0:HasComponent	Method	2:FindApplications	Defined in 6.6.4.		Mandatory
0:HasComponent	Method	2:RegisterApplication	Defined in 6.6.6.		Mandatory
0:HasComponent	Method	2:UpdateApplication	Defined in 6.6.7.		Mandatory
0:HasComponent	Method	2:UnregisterApplication	Defined in 6.6.8.		Mandatory
0:HasComponent	Method	2:GetApplication	Defined in 6.6.9.		Mandatory
0:HasComponent	Method	2:QueryApplications	Defined in 6.6.10.		Mandatory
0:HasComponent	Method	2:QueryServers	Defined in 6.6.11.		Mandatory
Conformance Units					
GDS Application Directory					

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 *QueryApplications 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*

6.6.4 FindApplications

FindApplications is used to find the *ApplicationId* for an approved *OPC UA Application* (see 6.6.6 or 6.4). This list of records returned shall have zero or one element.

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 6.6.5.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The *ApplicationUri* is too long or not a valid URI.

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

Table 6 – FindApplications Method AddressSpace Definition

Attribute	Value				
BrowseName	2:FindApplications				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.5 ApplicationRecordDataType

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

If the *ApplicationType* is *Client* and the *serverCapabilities* includes RCP (reverse connect) then all *DiscoveryUrls* shall begin with the rcp+ prefix which indicates that reverse connections are supported. Otherwise, the *DiscoveryUrls* shall be empty.

If the *ApplicationType* is *ClientAndServer* the *serverCapabilities* may include RCP and all *DiscoveryUrls* that support reverse connect have the rcp+ prefix. If the same URL supports normal connetions and reverse connection then there shall be two elements in the *DiscoveryUrls* array with and without the rcp+ prefix.

Table 7 – ApplicationRecordDataType Structure

Name	Type	Description
ApplicationRecordDataType	Structure	Subtype of the *Structure DataType* defined in OPC 10000-5
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 OPC 10000-4.
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 HTTPS URL specifies the domain used as the Common Name of HTTPS *Certificates*.
ServerCapabilities	String[]	The list of server capability identifiers for the application. The allowed values are defined in Annex D.

Its representation in the *AddressSpace* is defined in Table 8.

Table 8 – ApplicationRecordDataType Definition

Attribute	Value				
BrowseName	2:ApplicationRecordDataType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of the 0:*Structure DataType* defined in OPC 10000-5.					
Conformance Units					
GDS Application Directory					

6.6.6 RegisterApplication

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

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *DiscoveryAdmin Role* or *the ApplicationAdmin Privilege* (see 6.2).

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

Servers that support non-transparent redundancy shall register as different applications. In addition, OPC 10000-4 requires the use of the NTRS *ServerCapability* defined in Annex D.

RegisterApplication shall not create duplicate records. If the *ApplicationUri* already exists the *Method* returns *Bad_EntryExists*.

If *RegisterApplication* succeeds the *OPC UA Application* is approved and is returned by *QueryApplications* and *FindApplications*.

If registration was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChanged AuditEventType* (see 6.6.12).

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_EntryExists	A record with the same *ApplicationUri* already exists.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

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

Table 9 – RegisterApplication Method AddressSpace Definition

Attribute	Value				
BrowseName	2:RegisterApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.7 UpdateApplication

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

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *DiscoveryAdmin Role, the ApplicationSelfAdmin Privilege, or the ApplicationAdmin Privilege* (see 6.2).

When updating an existing *Application* the *ApplicationUri* cannot be changed. If it is changed the *Method* returns *Bad_WriteNotSupported*.

If the update was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChanged AuditEventType* (see 6.6.12).

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_WriteNotSupported	The *applicationUri* was changed and it cannot be updated.
Bad_UserAccessDenied	The current user does not have the rights required.

Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 10 specifies the *AddressSpace* representation for the *UpdateApplication Method*.

Table 10 – UpdateApplication Method AddressSpace Definition

Attribute	Value				
BrowseName	2:UpdateApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.8 UnregisterApplication

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

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *DiscoveryAdmin Role, the ApplicationSelfAdmin Privilege, or the ApplicationAdmin Privilege* (see 6.2).

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 an *Application* has *Certificates* issued by the *CertificateManager*, these *Certificates* shall be revoked when this *Method* is called.

If un-registration was successful and auditing is supported, the GDS shall generate the *ApplicationRegistrationChanged AuditEventType* (see 6.6.12).

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 needed to unregister the application.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

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

Table 11 – UnregisterApplication Method AddressSpace Definition

Attribute	Value				
BrowseName	2:UnregisterApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.9 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 in 6.6.5

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 needed to read the requested record.

Table 12 specifies the *AddressSpace* representation for the *GetApplication Method*.

Table 12 – GetApplication Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetApplication				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.10 QueryApplications

QueryApplications is used to find *Client* or *Server* applications that meet the specified filters. The only *Clients* returned are those that support the reverse connection capability described in OPC 10000-6.

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.

The *lastCounterResetTime* parameter is used to indicate that the counters on records had to be reset for some reason such as a *Server* restart. The *Client* may not use any *nextRecordId* received prior to this time to set the value for the *startingRecordId* in a new call.

The return parameter is a list of *ApplicationDescriptions*. The mapping from a ApplicationRecord to an *ApplicationDescriptions* is shown in Table 13.

Table 13 – ApplicationRecordDataType to ApplicationDescription Mapping

ApplicationRecordDataType	ApplicationDescription	Notes
applicationId	--	Ignored
applicationUri	applicationUri	
applicationType	applicationType	
applicationNames	applicationName	The name that best matches the *preferredLocales* for the current *Session* is returned. If there is no *Session* the first element is returned.
productUri	productUri	
discoveryUrls	discoveryUrls	
--	gatewayServerUri	Set to NULL.
--	discoveryProfileUri	Set to NULL.
serverCapabilities	--	Ignored

Signature

```
QueryApplications(
    [in]  UInt32 startingRecordId
    [in]  UInt32 maxRecordsToReturn
    [in]  String applicationName
    [in]  String applicationUri
    [in]  UInt32 applicationType
    [in]  String productUri
    [in]  String[] capabilities
    [out] UtcTime 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 OPC 10000-4. 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 OPC 10000-4. 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 OPC 10000-4. 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. The allowed values are defined in Annex D.
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 OPC 10000-4.

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

Table 14 – QueryApplications Method AddressSpace Definition

Attribute	Value				
BrowseName	2:QueryApplications				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.11 QueryServers (deprecated)

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.

The *lastCounterResetTime* parameter is used to indicate that the counters on records had to be reset for some reason such as a *Server* restart. The *Client* may not use any *recordId* received prior to this time to set the value for the *startingRecordId* in a new call.

The return parameter is a list of *ServerOnNetwork Structures*. The mapping from a *ApplicationRecordDataType* to an *ServerOnNetwork* is shown in Table 15.

Table 15 – ApplicationRecordDataType to ServerOnNetwork Mapping

ApplicationRecordDataType	ServerOnNetwork	Notes
applicationId	--	Ignored
applicationUri	--	Ignored
applicationType	--	Ignored
applicationNames	serverName	The name that best matches the *preferredLocales* for the current *Session* is returned. If there is no *Session* the first element is returned.
productUri	--	Ignored
discoveryUrls	discoveryUrl	A *ServerOnNetwork* record is returned for each discoveryUrl in the *ApplicationRecord*.
serverCapabilities	serverCapabilities	
--	recordId	This is the recordId assigned by the *QueryServers* call. It may be used as the startedRecordId in a subsequent call to *QueryServers.*

Signature

```
QueryServers(
    [in]  UInt32 startingRecordId
    [in]  UInt32 maxRecordsToReturn
    [in]  String applicationName
    [in]  String applicationUri
    [in]  String productUri
    [in]  String[] serverCapabilities
    [out] UtcTime 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 OPC 10000-4. 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 OPC 10000-4. 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 OPC 10000-4. 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 OPC 10000-4.

Method Result Codes (defined in Call Service)

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

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

Table 16 – QueryServers Method AddressSpace Definition

Attribute	Value				
BrowseName	2:QueryServers				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

6.6.12 ApplicationRegistrationChangedAuditEventType

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

Its representation in the *AddressSpace* is formally defined in Table 17.

Table 17 – ApplicationRegistrationChangedAuditEventType Definition

Attribute	Value				
BrowseName	2:ApplicationRegistrationChangedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
Conformance Units					
GDS Application Directory					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantics are defined in OPC 10000-5.

7 Certificate Management

7.1 Overview

Certificate management functions comprise the management and distribution of certificates and *TrustLists* 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 OPC 10000-2.

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

The *CertificateManager* is intended to work in conjunction with different *Certificate* management services such as Active Directory. The *CertificateManager* 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* should support the following features:

- Onboarding (first time setup for a device/application);
- Renewal (renewing expired or compromised certificates);
- *TrustList* Update (updating the *TrustLists* 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.

OPC 10000-21 defines the complete process to automatically authenticate and onboard new *Devices* that depends on the *Devices* having support built in by the *Manufacturer*. If this support is not present, *Devices* and *OPC UA Applications* have to be manually onboarded using the mechanisms defined in this document.

During manual onboarding, 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 *TrustLists* before connecting to the *CertificateManager* for application setup, *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.

Clients shall only connect to a *CertificateManager* which the *Client* has been configured to trust. This may require an out of band configuration step which is completed prior to starting the manual onboarding process.

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

7.2 Roles and Privileges

CertificateManagers restrict access to many of the features they provide. These restrictions are described either by referring to well-known *Roles* which a *Session* must have access to or by referring to *Privileges* which are assigned to *Sessions* using mechanisms other than the well-known *Roles*. The well-known *Roles* used for *CertificateManagers* are listed in Table 18.

Table 18 – Well-known Roles for a CertificateManager

Name	Description
CertificateAuthorityAdmin	This *Role* grants rights to request or revoke any *Certificate*, update any *TrustList* or assign *CertificateGroups* to *OPC UA Applications*.
RegistrationAuthorityAdmin	This *Role* grants rights to approve *Certificate* Signing requests or NewKeyPair requests.

SecurityAdmin	This *Role* grants the right to change the security configuration of a *CertificateManager*.

The well-known *Roles* for *Server* managed by a *CertificateManager* are listed in Table 19.

Table 19 – Well-known Roles for Server managed by a CertificateManager

Name	Description
SecurityAdmin	For *PushManagement*, this *Role* grants the right to change the security configuration of a *Server* managed by a *CertificateManager*.

The *Privileges* used in for *CertificateManagers* are listed in Table 20.

Table 20 – Privileges for a CertificateManager

Name	Description
ApplicationSelfAdmin	This *Privilege* grants an *OPC UA Application* the right to renew its own *Certificate* or read its own *CertificateGroups* and *TrustLists*. The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application*.
ApplicationAdmin	This *Privilege* grants rights to request or renew *Certificates,* read *TrustLists* or *CertificateGroups* for one or more *OPC UA Applications.* The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application* and the set of *OPC UA Applications* that it is authorized to manage.

7.3 Pull Management

PullManagement is performed by using the *CertificateManager* information model, in particular the Methods defined in 7.9. The interactions between *Application* and *CertificateManager* during *PullManagement* are illustrated in Figure 14.

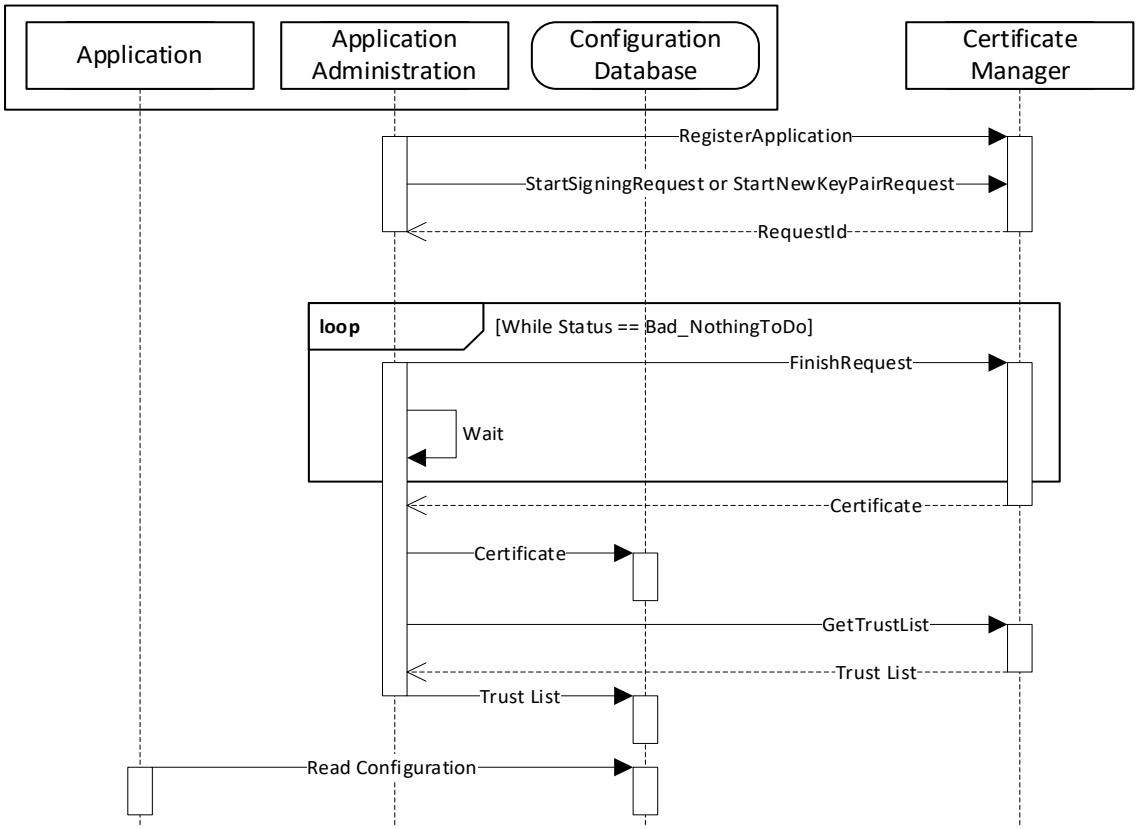

Figure 14 – The Pull Management Model for Certificates

The Application Administration component may be part of the *Client* or *Server* 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 *TrustList*.

Security in *PullManagement* requires an encrypted channel and authorized credentials. These credentials may be user credentials for a *CertificateAuthorityAdmin* or application credentials determined by the *Certificate* used to create the *SecureChannel.* Examples of the application credentials include *Certificates* previously issued to the application being accessed, *Device Certificates* issued by the *Registrar* defined in OPC 10000-21 or *Certificates* isssued to an application with accesss to the *ApplicationAdmin Privilege* (see 6.2).

Before a *Client* provides any secrets associated with credentials to a *CertificateManager* it needs to know that it can trust the *CertificateManager*. This can be done by an administrator that adds the *CertificateManager* to the *Client TrustList* before the *Client* attempts to connect to the *CertificateManager*. If the *Client* finds a *CertificateManager* on a network via mDNS or other insecure mechanism it could trust the *CertificateManager* if it has some independently acquired information that allows it to trust the *CertificateManager*. For example, the DNS address of the *CertificateManager* could be provided by a trusted authority and this address appears in the *Certificate* of the *CertificateManager* being used and the address was used to connect.

Once the *Client* finds a *CertificateManager* that it can trust, it needs to provide credentials that allows the *CertificateManager* to know that it can issue *Certificates*. The *Certificate* used by the *Client* can be the credential if there is an out of band process to provide the *Certificate* to the *CertificateManager*. The *CertificateManager* could provide a one-time password to the *Client* via email or other mechanisms.

The *CertificateManager* can only issue *Certificates* to authenticated *Clients*. There are a number of ways to authenticate *Clients*:

1) The *CertificateManager* is pre-configured with information about the *Client Certificate* that allows the *CertificateManager* to know that the *Client* can request *Certificates* even if anonymous user credentials are used. The *Client* may be a DCA authenticated by a *Registrar* (see OPC 10000-21), a *Client* with a previously issued *Certificate*, or a *Client* authorized to create *Certificates* on behalf of other applications.

2) The *CertificateManager* may have a manual process where an administrator reviews each request before issuing a *Certificate*.

3) The *Client* provides user credentials. A *Client* shall not provide a secret (e.g. a password) to an untrusted *CertificateManager*.

7.4 Push Management

PushManagement is targeted at *Server* applications and relies on *Methods* defined in 7.10 to get a *CertificateRequest* 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 *PushManagement* are illustrated in Figure 15.

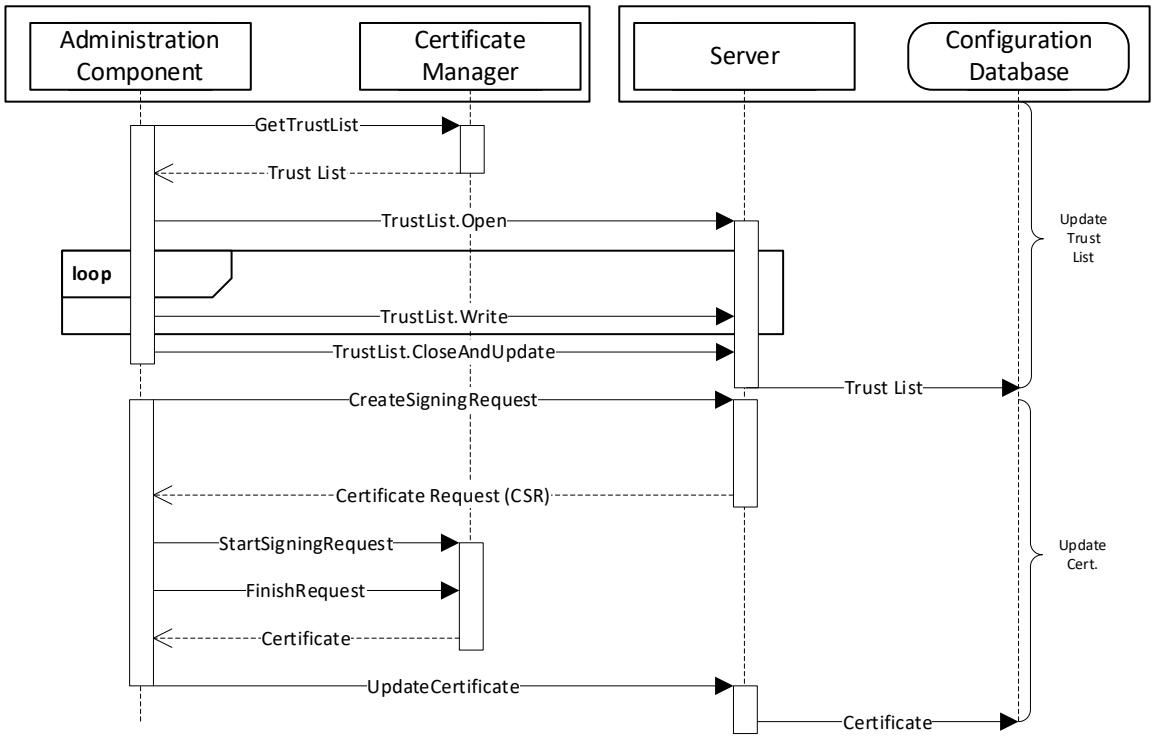

Figure 15 – The Push Certificate Management Model

The Administration Component may be part of the *CertificateManager* or a standalone utility that uses OPC UA to communicate with the *CertificateManager* (see 7.3 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 *TrustList*. In Figure 15 the *TrustList* update is shown to happen first. This is necessary to ensure any CRLs are provided to the *Server* before the new *Certificate* is updated. The *TrustList* update may be skipped If the current *TrustList* allows the *Server* to validate the new *Certificate*.

Security when using the *PushManagement* model requires an encrypted channel and a *Client* with access to the *SecurityAdmin Role*. For example, *SecurityAdmin Role* could be mapped to user credentials for an administrator or to a *ApplicationInstance Certificate* issued to a configuration tool. OPC 10000-21 defines a mechanism to install administrative *Client Certificates* into the *Server TrustList*.

7.5 Application Setup

Application Setup is the initial installation of an OPC UA *Server* or *Client* into a system in which a GDS is available and managing *Certificates*. Applications using a *Client* interface can be setup using the *PullManagement*. Applications using a *Server* interface can be setup using the *PushManagement*.

The push and *PullManagement* are also integrated into OPC 10000-21 which specifies how new *Devices* can be authenticated when they are added to the network. Once a *Device* is authenticated the *Device* is trusted and can use the push or *PullManagement* without additional administrator credentials.

OPC UA *Servers* that do not support OPC 10000-21 typically auto-generate a self-signed *Certificate* when they first start. They may also have a pre-configured *TrustList* with *Applications* that are allowed to setup the *Server*. For example, a machine 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 *SecurityAdmin* credentials to create the secure connection needed for setup using *PushManagement*. Once the *Server* has been given its initial *TrustList* the *Server* should then restrict access to those *Clients* with *Certificates* in the *TrustList*. A vendor specific process for setup is required if a device restricts the *Clients* allowed to connect securely.

See Annex G for more specific examples of how to provision an application when OPC 10000-21 is not used.

7.6 Pull Management Workflow

In this workflow the OPC UA *Application* that gets *Certificates* from the *CertificateManager* is the *Client* that executes the workflow and the *CertificateManager* is the *Server* processing the request in the workflow.

The *Application* is authenticated with the *Certificate* signed by the *CertificateManager* (or the *Certificate* assigned during registration). The *UserTokenType* is always *Anonymous* using the *ApplicationSelfAdmin Privilege*.

The workflow for *PullManagement* is shown in Figure 16 and the steps are described in Table 21. The two options for the key pair creation are described in Figure 17. The boxes with **blue text** indicate *Method* calls.

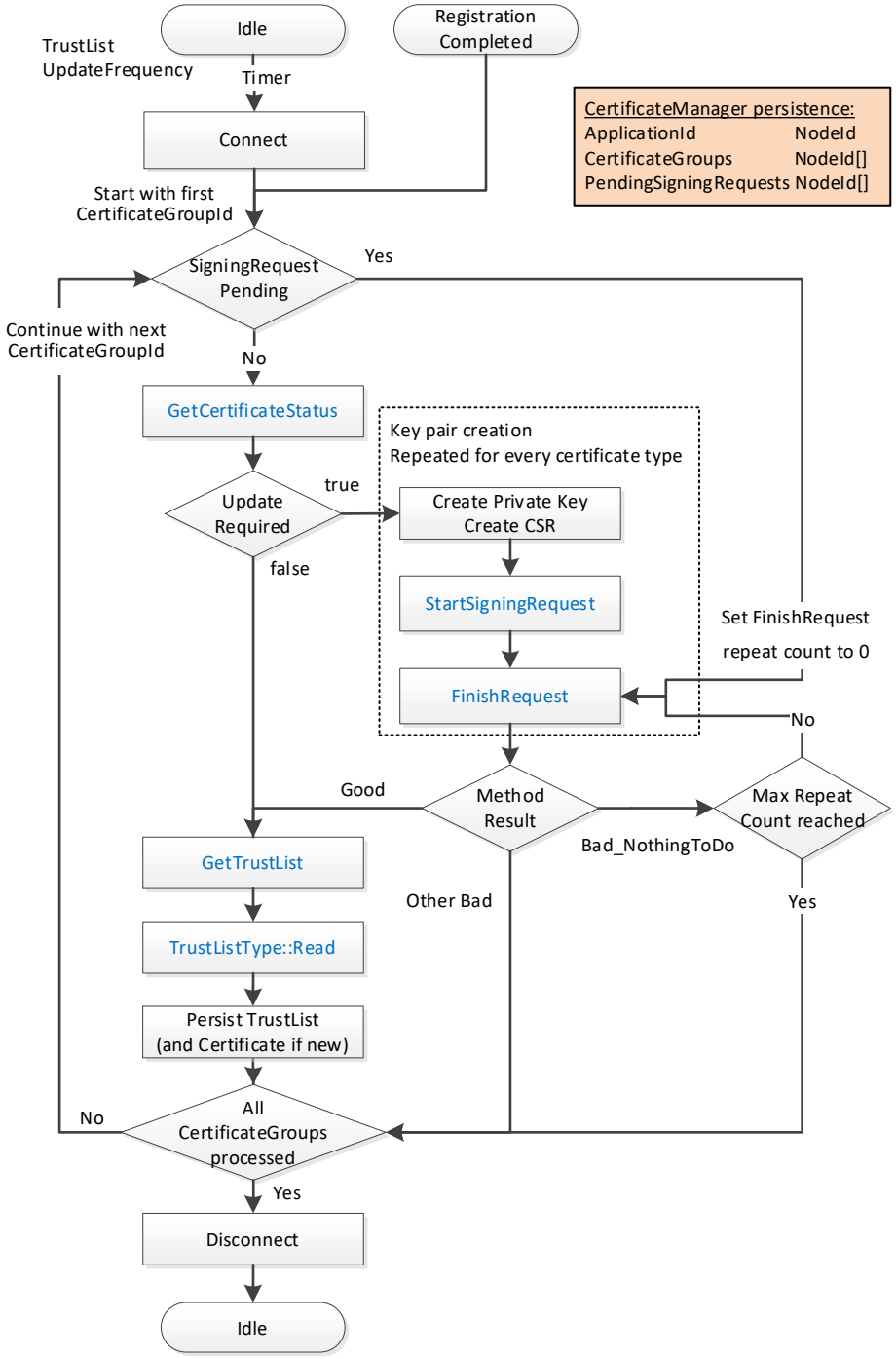

Figure 16 – Certificate Pull Management Workflow

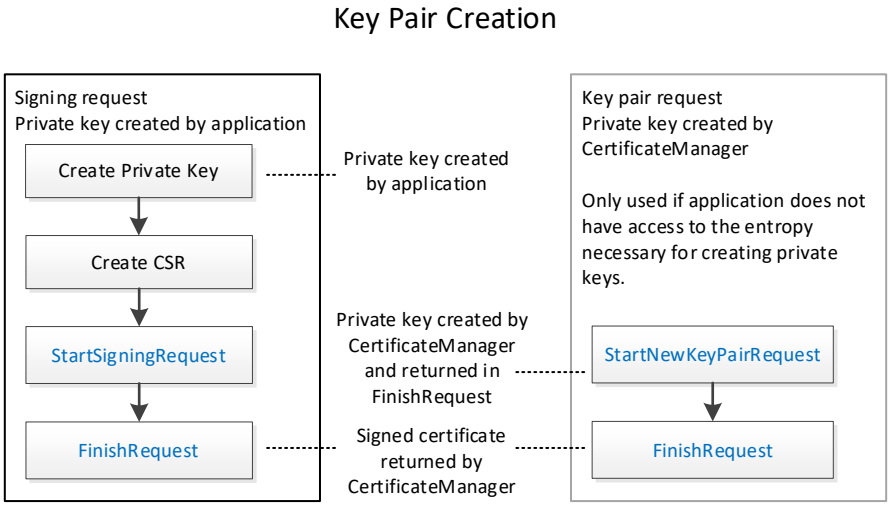

Figure 17 – The Pull Management Options for Key Pair Creation

The steps of the *PullManagement* workflow are described in detail in Table 21.

Table 21 – Certificate Pull Management Workflow Steps

Step	Description
Certificate management begin options	The following options are possible to start the *PullManagement*. 1. Continue application setup using the *Session* available from the application registration workflow described in 6.5. 2. Cyclic check of the application status using a new connection to the *CertificateManager*. The cycle time is defined by the *UpdateFrequency* on the related *TrustList Object* in the *CertificateManager*.
Connect	Create a connection for option (2). For the connection management with the *CertificateManager* the *Services OpenSecureChannel*, *CreateSession* and *ActivateSession* are used to create a connection with *MessageSecurityMode SignAndEncrypt* and an *Anonymous* user. *Application* authentication is used by the *CertificateManager* to allow OPC UA *Applications* to access the necessary resources to update themselves using the *ApplicationSelfAdmin Privilege*.
Required information	The OPC UA *Application* needs to know the following information to execute the *PullManagement* workflow • ApplicationId *NodeId* of the OPC UA *Application* in the *CertificateManager*. • CertificateGroupIds *NodeIds* for each *CertificateGroup* in the *CertificateManager* that are relevant to the OPC UA *Application*. This includes a mapping to the related internal *CertificateGroup* and the *CertificateTypes* needed. • Pending signing requests *RequestIds* for pending signing requests that need to be completed and their relationship with a *CertificateGroup* and *CertificateType*.
SigningRequestPending	If one or more signing requests are pending for a *CertificateGroup*, the *FinishRequest Method* is called directly with the *ApplicationId* and the *RequestId* for the pending signing request. The repeat count is set to 0 in this case.
GetCertificateStatus	The *Method GetCertificateStatus* is called with the *ApplicationId* and the *CertificateGroupId* to check if a certificate update is needed. This is repeated for each *CertificateType* needed for the *CertificateGroup*.
Update Required	If *GetCertificateStatus* returns updateRequired set to True for one or more combinations of *CertificateGroup* and *CertificateType*, the process for key pair creation is started for the affected combinations.
Create CSR	The application creates a certificate signing request (CSR). It is strongly recommended, that the OPC UA *Application* creates a new private key for each signing request.
StartSigningRequest	The *Method StartSigningRequest* is called for each *CertificateGroup* and *CertificateType* together with the CSR to request a signed *Certificate* from the *CertificateManager*. Each *Method* call needs it's own CSR. As alternative for OPC UA Applications who do not have access to a cryptograhically sufficient entropy source, the *Method StartNewKeyPairRequest* can be used. In this case the private key is created by the *CertificateManager*.

	Both Methods return a *RequestId* that can be passed to the *FinishRequest Method*. The repeat count for *FinishRequest* is set to a small number like 2.
FinishRequest	The *Method FinishRequest* is called to check the results of a previous *StartSigningRequest* or *StartNewKeyPairRequest*. The following results are possible: • If *FinishRequest* returns a *Good* result, the *Method* returns the signed *Certificate* and optionally the private key for the *StartNewKeyPairRequest* case. • If *FinishRequest* returns *Bad_NothingToDo* it indicates that the request is not completed yet. If the repeat count is not 0, the repeat count is decremented and *FinishRequest* is repeated after a short delay. If the repeat count is 0, the *RequestId* is persisted and the next *CertificateGroup* or *CertificateType* is processed • If *FinishRequest* returns any other *Bad* result, a new request must be sent in the next cycle
GetTrustList	If all *Certificates* for a *CertificateGroup* are up-to-date, the *TrustList* is checked for updates by calling the *Method GetTrustList*. The Method returns the *NodeId* of the *TrustList Object* for the *CertificateGroup*. The *LastUpdateTime* of *TrustList Object* indicates when the contents of the *TrustList* changed. When using *PullManagement*, the *Client* should check this *Property* before downloading the *TrustList*.
TrustListType::Read	The *NodeId* of the *TrustList Object* returned by *GetTrustList* is used to open the *TrustList* for reading and to read the current content of the *TrustList*.
Persist TrustList	If a *TrustList* update or *Certificate* updates are available, they are persisted for further use by the OPC UA *Application*. They must be persisted at the same time to ensure a consistent setup.
Repeat for all CertificateGroups	Repeat the process for all *CertificateGroups*.
Disconnect	Disconnect from *CertificateManager*.

7.7 Push Management Workflow

In this workflow the *CertificateManager* is the *Client* that executes the steps and the OPC UA *Application* is the *Server* that is processing the request in the sequence. The workflow is started if the *CertificateManager* determines that an update is required.

The workflow for *PushManagement* is shown in Figure 18. The two options for the key pair creation are described in Figure 19. The boxes with **blue text** indicate *Method* calls.

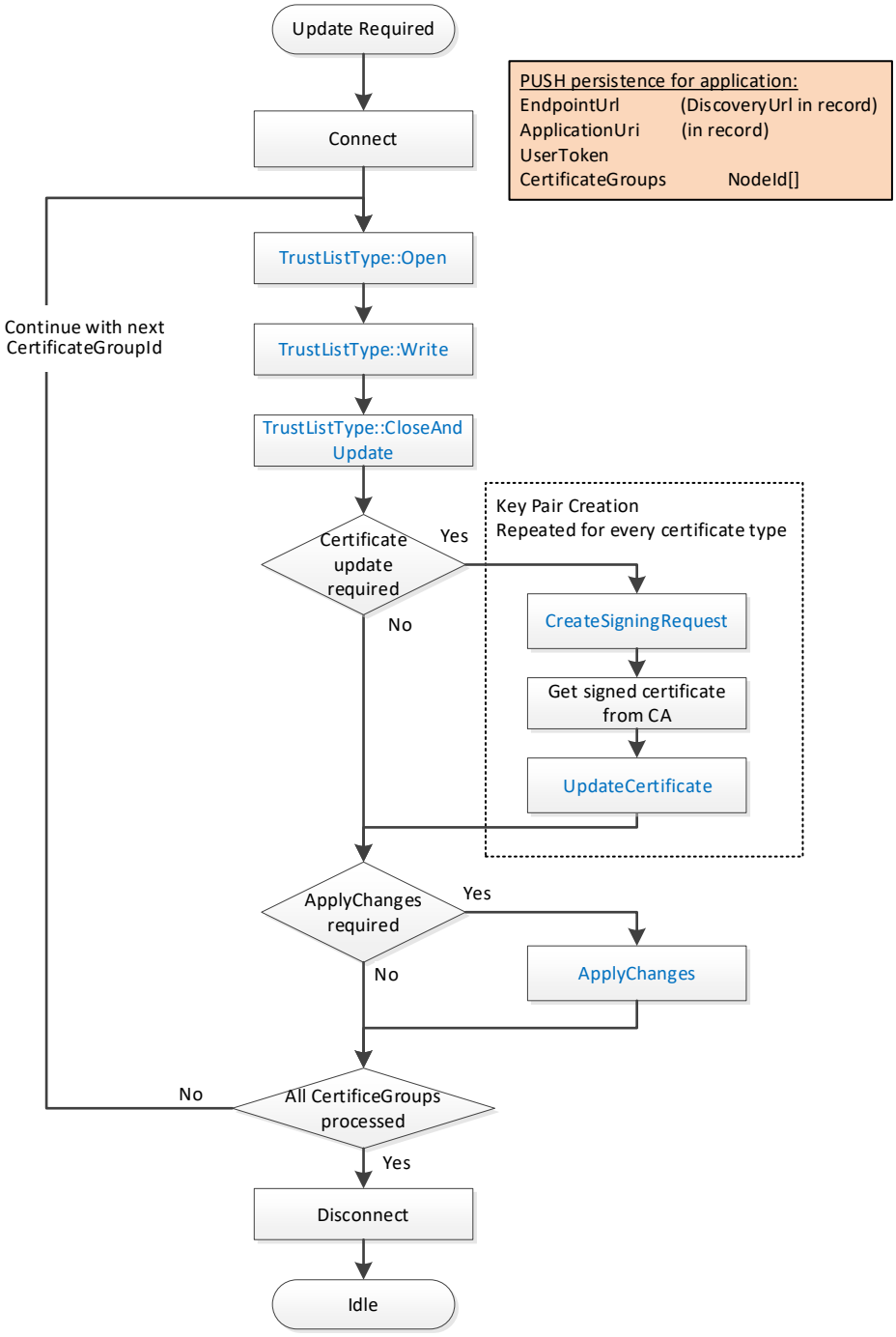

Figure 18 – The Certificate Push Management Workflow

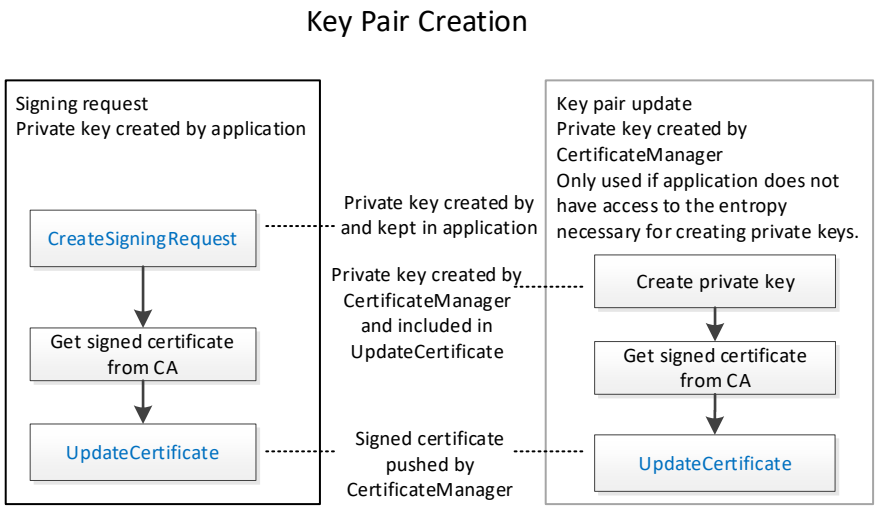

Figure 19 – The Push Management Options for Key Pair Creation

7.8 Common Information Model

7.8.1 Overview

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

7.8.2 TrustLists

7.8.2.1 TrustListType

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

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 *TrustList*, it may be an Administrative user that is reading a *TrustList* to deploy to an Application host or it may be an Application that can only access the *TrustList* assigned to it.

The *TrustList* file is a UA Binary encoded stream containing an instance of *TrustListDataType* (see 7.8.2.6).

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

If a transaction is in progress (see 7.10.7) on another *Session* then the *Server* shall return *Bad_TransactionPending* if *Open* is called with the *Write Mode* bit set. If the *Server* supports transactions then the *Server* creates a new transaction or continues an existing transaction if *Open* is called with the *Write Mode* bit set.

If the *SecureChannel* is not authenticated the *Server* shall return *Bad_SecurityModeInsufficient*.

Servers shall automatically *Close TrustLists* if there are no calls to *Methods* on the *TrustList Object* within the time specified by the *ActivityTimeout Property*.

The *Size Property* inherited from *FileType* has no meaning for *TrustList* and returns the error code defined in OPC 10000-20.

When a *Client* opens the file for writing the *Server* will not actually update the *TrustList* 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 *TrustList*.

When the *CloseAndUpdate Method* is called the *Server* will validate all new *Certificates* and *CRLs.* If this validation fails the *TrustList* is not updated and the *Server* returns the appropriate *Certificate* error code (see OPC 10000-4).

Its representation in the *AddressSpace* is formally defined in Table 22.

Table 22 – TrustListType Definition

Attribute	**Value**				
BrowseName	0:TrustListType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*FileType* defined in OPC 10000-20.					
0:HasProperty	Variable	0:LastUpdateTime	0:UtcTime	0:PropertyType	Mandatory
0:HasProperty	Variable	0:UpdateFrequency	0:Duration	0:PropertyType	Optional
0:HasProperty	Variable	0:ActivityTimeout	0:Duration	0:PropertyType	Optional
0:HasProperty	Variable	0:DefaultValidationOptions	TrustListValidationOptions	0:PropertyType	Optional
0:HasComponent	Method	0:OpenWithMasks	Defined in 7.8.2.2.		Mandatory
0:HasComponent	Method	0:CloseAndUpdate	Defined in 7.8.2.3.		Mandatory
0:HasComponent	Method	0:AddCertificate	Defined in 7.8.2.4.		Mandatory
0:HasComponent	Method	0:RemoveCertificate	Defined in 7.8.2.5.		Mandatory
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

The *LastUpdateTime* indicates when the *TrustList* was last updated. The *LastUpdateTime* shall reflect changes made using the *TrustList Object Methods*. A *TrustList Object* in a *CertificateManager* shall also reflect changes made in other ways.

The *LastUpdateTime* of a *TrustList Object* in a *CertificateManager* allows *Clients* using the *PullManagement* to know whether the *TrustList* has changed since the last time they accessed it. The *LastUpdateTime* of a *TrustList Object* in the *ServerConfiguration* allows administration *Clients* to check for out of date *TrustLists*.

The *UpdateFrequency Property* specifies how often the *TrustList* needs to be checked for changes. When the *CertificateManager* specifies this value, all *Clients* that read a copy of the *TrustList* should connect to the *CertificateManager* and check for updates to the *TrustList* within 2 times the *UpdateFrequency*. The choice of *UpdateFrequency* depends on how quickly system changes need to be detected and the performance constraints of the system. *UpdateFrequencies* that are too long create security risks because of out of date CRLs. *UpdateFrequencies* that are too short negatively impact system performance. If the *TrustList Object* is contained within a *ServerConfiguration Object* then this Property is not present.

The *ActivityTimeout Property* specifies the maximum elapsed time between the calls to *Methods* on the *TrustList Object* after *Open* or *OpenWithMasks* is called. If this time elapses the *TrustList* is automatically closed by the *Server* and any changes are discarded. The default value is 60 000 milliseconds (1 minute).

The *DefaultValidationOptions Property* specifies the default options to use when validating *Certificates* with the *TrustList*. The *TrustListValidationOptions DataType* is defined in 7.8.2.8. This *Property* may be updated by *Clients* with access to the *SecurityAdmin Role*.

If auditing is supported, the *CertificateManager* shall generate the *TrustListUpdated*AuditEventType (see 7.8.2.11) when the *TrustList* is updated via the *CloseAndUpdate*, *AddCertificate*, *RemoveCertificate* or *ApplyChanges* (see 7.10.7) *Methods*. The *Event* is only raised once after the asynchronous update process completes.

7.8.2.2 OpenWithMasks

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

This *Method* can only be used to read the *TrustList*.

After calling this *Method*, the *Client* calls *Read* one or more times to get the *TrustList*. If the *Server* is able to detect out of band changes to *theTrustList* before the *Client* calls the *Close Method*, then the next *Read* returns *Bad_InvalidState*. If the *Server* cannot detect out of band changes it shall ensure the *Client* receives a consistent snapshot.

For *PullManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationSelfAdmin Privilege,* or the *ApplicationAdmin Privilege* (see 7.2).

For *PushManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

Signature

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

Argument	Description
masks	The parts of the *TrustList* that are include in the file to read. The masks are defined in 7.8.2.7.
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.
Bad_TransactionPending	The *TrustList* cannot be opened because it is part of a transaction is in progress.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 23 specifies the *AddressSpace* representation for the *OpenWithMasks Method*.

Table 23 – OpenWithMasks Method AddressSpace Definition

Attribute	Value				
BrowseName	0:OpenWithMasks				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.8.2.3 CloseAndUpdate

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

If only part of the *TrustList* is being updated the *Server* creates a new *TrustList* that includes the existing *TrustList* plus any updates and validates the new *TrustList*.

The *Server* shall verify that every *Certificate* in the new *TrustList* is valid using the validation process defined in OPC 10000-4. If an invalid *Certificate* is found the *Server* shall return an error and shall not replace the existing *TrustList*.

If the *Server* does not support transactions it applies the changes immediately and sets *applyChangesRequired* to FALSE. If the *Server* supports transactions then the *Server* creates a new transaction or continues an existing transaction and sets *applyChangesRequired* to TRUE.

If a transaction exists on the current *Session*, the *Server* does not update the *TrustList* until *ApplyChanges* (see 7.10.7) is called. Any *Clients* that read the *TrustList* before *ApplyChanges* is called will receive the existing *TrustList* before the transaction started.

If errors occur, the new *TrustList* is discarded.

When the *TrustList* changes the *Server* shall re-evaluate the *Certificate* associated with any open *Sessions* and *SecureChannels. Sessions* or *SecureChannels* with an untrusted or revoked *Certificate* shall be closed. This process may not complete before the *Method* returns and could take a significant amount of time on systems with limited resources.

The structure uploaded includes a mask (see 7.8.2.7) 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	If TRUE the *ApplyChanges* Method (see 7.10.7) shall be called before the new *TrustList* will be used by the *Server*. If FALSE the *TrustList* is now in use.

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 TrustList. The DiagnosticInfo shall specify which Certificate(s) are invalid and the specific error.
Bad_RequestTooLarge	The changes would result in a *TrustList* that exceeds the *MaxTrustListSize* for the *Server*.
Bad_TransactionPending	Changes are queued on another *Session* (see 7.10.7)

Table 24 specifies the *AddressSpace* representation for the *CloseAndUpdate Method*.

Table 24 – CloseAndUpdate Method AddressSpace Definition

Attribute	Value				
BrowseName	0:CloseAndUpdate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.8.2.4 AddCertificate

The *AddCertificate Method* allows a *Client* to add a single *Certificate* to the *TrustList*. The *Server* shall verify that the *Certificate* using the validation process defined in OPC 10000-4. If an invalid *Certificate* is found the *Server* shall return an error and shall not update the *TrustList*.

This *Method* will return a validation error if the *Certificate* is issued by a CA and the *Certificate* for the issuer is not in the *TrustList*.

This *Method* cannot provide CRLs so issuer *Certificates* cannot be added with this *Method*. Instead, CA *Certificates* and their CRLs shall be managed with the *Write Method* on the containing *TrustList Object.*

This *Method* cannot be called if the containing *TrustList Object* is open.

This *Method* returns *Bad_TransactionPending* if a transaction is in progress (see 7.10.7).

This *Method* returns *Bad_NotWritable* if the *TrustList Object* is read only.

For *PullManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role* (see 7.2).

For *PushManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

Signature

```
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 *trustedCertificates* list. If FALSE *Bad_CertificateInvalid* is returned.

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 *Open Method* was called with write access and the *CloseAndUpdate* Method has not been called.
Bad_RequestTooLarge	The changes would result in a *TrustList* that exceeds the *MaxTrustListSize* for the *Server*.
Bad_TransactionPending	Transaction has started and *ApplyChanges* or *CancelChanges* has not been called.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 25 specifies the *AddressSpace* representation for the *AddCertificate Method*.

Table 25 – AddCertificate Method AddressSpace Definition

Attribute	Value				
BrowseName	0:AddCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

7.8.2.5 RemoveCertificate

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

If the *Certificate* is a CA *Certificate* that has CRLs then all CRLs for that CA are removed as well.

This *Method* returns *Bad_CertificateChainIncomplete* if the *Certificate* is a CA *Certificate* needed to validate another *Certificate* in the *TrustList*.

This *Method* returns *Bad_TransactionPending* if a transaction is in progress (see 7.10.7).

This *Method* returns *Bad_NotWritable* if the *TrustList Object* is read only. For *PullManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Session* that has access to the *CertificateAuthorityAdmin Role* (see 7.2).

For *PushManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Session* that has access to the *SecurityAdmin Role* (see 7.2).

Signature

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

Argument	Description
Thumbprint	The *CertificateDigest* 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 *Open Method* was called with write access and the *CloseAndUpdate* Method has not been called.
Bad_CertificateChainIncomplete	The *Certificate* is needed to validate another *Certificate* in the *TrustList.*
Bad_TransactionPending	Transaction has started and *ApplyChanges* or *CancelChanges* has not been called.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 26 specifies the *AddressSpace* representation for the *RemoveCertificate Method*.

Table 26 – RemoveCertificate Method AddressSpace Definition

Attribute	Value				
BrowseName	0:RemoveCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

7.8.2.6 TrustListDataType

This type defines a DataType which stores the *TrustList* of a *Server.* Its values are defined in Table 27.

Table 27 – TrustListDataType Structure

Name	Type	Description
TrustListDataType	Structure	Subtype of the *Structure DataType* defined in OPC 10000-5
specifiedLists	UInt32	A bit mask which indicates which lists contain information. The *TrustListMasks* enumeration in 7.8.2.7 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.

Its representation in the *AddressSpace* is defined in Table 28.

Table 28 – TrustListDataType Definition

Attribute	Value				
BrowseName	0:TrustListDataType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of the 0:*Structure DataType* defined in OPC 10000-5.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.2.7 TrustListMasks

This is a DataType that defines the values used for the SpecifiedLists field in the *TrustListDataType*. Its values are defined in Table 29.

Table 29 – TrustListMasks Enumeration

Name	Value	Description
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.

Its representation in the *AddressSpace* is defined in Table 30.

Table 30 – TrustListMasks Definition

Attribute	Value				
BrowseName	0:TrustListMasks				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of the *Enumeration DataType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:EnumValues	0:EnumValueType []	0:PropertyType	
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.2.8 TrustListValidationOptions

This *DataType* defines flags for *TrustListValidationOptions* is formally defined in Table 31.

Table 31 – TrustListValidationOptions Values

Value	Bit No.	Description
SuppressCertificateExpired	0	Ignore errors related to the validity time of the *Certificate*.
SuppressHostNameInvalid	1	Ignore mismatches between the host name or *ApplicationUri*.
SuppressRevocationStatusUnknown	2	Ignore errors if the revocation list cannot be found for the issuer of the *Certificate*.
SuppressIssuerCertificateExpired	3	Ignore errors if an issuer has an expired *Certificate*.
SuppressIssuerRevocationStatusUnknown	4	Ignore errors if the revocation list cannot be found for any issuer of issuer *Certificates*.
CheckRevocationStatusOnline	5	Check the revocation status online.
CheckRevocationStatusOffline	6	Check the revocation status offline.

If *CheckRevocationStatusOnline* is set, the *Certificate* validation process defined in OPC 10000-4 will look for the *authorityInformationAccess* extension to find an OCSP (RFC 6960) endpoint which can be used to determine if the *Certificate* has been revoked.

If the OCSP endpoint is not reachable then the *Certificate* validation process looks for offline CRLs if the *CheckRevocationStatusOffline* bit is set. Otherwise, validation fails.

The revocation status flags only have meaning for issuer *Certificates* and are used when validating *Certificates* issued by that issuer.

The default value for this *DataType* only has the *CheckRevocationStatusOffline* bit set.

The *TrustListValidationOptions* representation in the *AddressSpace* is defined in Table 32.

Table 32 – TrustListValidationOptions Definition

Attribute		Value			
BrowseName		0:TrustListValidationOptions			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of the 0:*UInt32 DataType* defined in OPC 10000-5					
0:HasProperty	Variable	0:OptionSetValues	0:LocalizedText []	0:PropertyType	
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.2.9 TrustListOutOfDateAlarmType

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

Table 33 – TrustListOutOfDateAlarmType definition

Attribute	Value				
BrowseName	0:TrustListOutOfDateAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the SystemOffNormalAlarmType defined in OPC 10000-9.					
0:HasProperty	Variable	0:TrustListId	0:NodeId	0:PropertyType	Mandatory
0:HasProperty	Variable	0:LastUpdateTime	0:UtcTime	0:PropertyType	Mandatory
0:HasProperty	Variable	0:UpdateFrequency	0:Duration	0:PropertyType	Mandatory
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

TrustListId Property specifies the *NodeId* of the out-of-date *TrustList Object*.

LastUpdateTime Property specifies when the *TrustList* was last updated.

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

7.8.2.10 TrustListUpdateRequestedAuditEventType

This event is raised when a *Method* that changes the *TrustList* is called

It is raised when *CloseAndUpdate, AddCertificate or RemoveCertificate Method* on a *TrustListType Object* is called.

Its representation in the *AddressSpace* is formally defined in Table 34.

Table 34 – TrustListUpdateRequestedAuditEventType Definition

Attribute	Value				
BrowseName	0:TrustListUpdateRequestedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

7.8.2.11 TrustListUpdatedAuditEventType

This event is raised when a *TrustList* is successfully changed.

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

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

Its representation in the *AddressSpace* is formally defined inTable 35.

Table 35 – TrustListUpdatedAuditEventType Definition

Attribute	**Value**				
BrowseName	0:TrustListUpdatedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:TrustListId	0:NodeId	0:PropertyType	Mandatory
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

The *TrustListId Property* is the *NodeId* of the *TrustList Object* that was changed.

7.8.3 CertificateGroups

7.8.3.1 CertificateGroupType

This *ObjectType* is used for *Objects* which represent *CertificateGroups* in the *AddressSpace*. A *CertificateGroup* is a context that contains a *TrustList* and one or more *CertificateTypes* that can be assigned to an *Application*. This *ObjectType* allows an *Application* which has multiple *TrustLists* and/or *ApplicationInstance Certificates* to express them in its *AddressSpace*.

A *CertificateManager* can have many *CertificateGroups* which manage *CertificateTypes* and *TrustLists* for the applications in the system.

A *Server* has one or more C*ertificateGroups* which specify the *CertificateTypes* and *TrustLists* managed by the *Server*. Typically, there is a mapping between a *CertificateGroup* in a *Server* and a *CertificateGroup* in the *CertificateManager*. The mechanisms for creating that mapping are outside the scope of this specification.

This type is defined in Table 36.

Table 36 – CertificateGroupType Definition

Attribute	**Value**				
BrowseName	0:CertificateGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasComponent	Object	0:TrustList		0:TrustListType	Mandatory
0:HasProperty	Variable	0:CertificateTypes	0:NodeId[]	0:PropertyType	Mandatory
0:HasComponent	Object	0:CertificateExpired		0:CertificateExpirationAlarmType	Optional
0:HasCondition	ObjectType	0:CertificateExpirationAlarmType			
0:HasComponent	Object	0:TrustListOutOfDate		0:TrustListOutOfDateAlarmType	Optional
0:HasComponent	Method	0:GetRejectedList	Defined in 7.8.3.2.		Optional
Conformance Units					

GDS Certificate Manager Pull Model
Push Model for Global Certificate and TrustList Management

The *TrustList Object* is the *TrustList* associated with the *CertificateGroup*.

The *CertificateTypes Property* specifies the *NodeIds* of the *CertificateTypes* which may be assigned to Applications which belong to the *CertificateGroup*. For example, a *CertificateGroup* with the NodeId of *RsaMinApplicationCertificateType* (see 7.8.4.4) and the NodeId *RsaSha256ApplicationCertificate* (see 7.8.4.5) 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 *CertificateGroup* does not allow *Certificates* to be assigned to *Applications* (i.e. a *UserToken CertificateGroup* only exists to allow the associated *TrustList* to be read or updated). All *CertificateTypes* for a given *CertificateGroup* 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 a *Certificate* associated with the *CertificateGroup* is about to expire. If multiple *Certificates* are about to expiry an Alarm for each *Certificate* is raised. The *CertificateExpirationAlarmType* is defined in OPC 10000-9.

The *TrustListOutOfDate Object* is an Alarm which is raised when the *TrustList* has not been updated within the period specified by the *UpdateFrequency* (see 7.8.2.1). The *TrustListOutOfDateAlarmType* is defined in 7.8.2.9.

The *GetRejectedList Method* returns the list of *Certificates* that have been rejected by the *Server* when using the *TrustList* associated with the CertificateGroup. It can be used to track activity or allow administrators to move a rejected *Certificate* into the *TrustList*. This *Method* shall only be present on *CertificateGroups* which are part of the *ServerConfiguration Object* defined in 7.10.3.

7.8.3.2 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* can delete entries from the list returned if the maximum message size is not large enough to allow the entire list to be returned.

Servers only add *Certificates* to this list that have no unsuppressed validation errors but are not trusted.

For *PullManagement*, this *Method* is not present on the *CertificateGroup*.

For *PushManagement*, this *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 37 specifies the *AddressSpace* representation for the *GetRejectedList Method*.

Table 37 – GetRejectedList Method AddressSpace Definition

Attribute	Value				
BrowseName	0:GetRejectedList				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.8.3.3 CertificateGroupFolderType

This type is used for *Folders* which organize *Certificate Groups* in the *AddressSpace*. This type is defined in Table 38.

Table 38 – CertificateGroupFolderType Definition

Attribute	Value				
BrowseName	0:CertificateGroupFolderType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FolderType* defined in OPC 10000-5.					
0:HasComponent	Object	0:DefaultApplicationGroup		0:CertificateGroupType	Mandatory
0:HasComponent	Object	0:DefaultHttpsGroup		0:CertificateGroupType	Optional
0:HasComponent	Object	0:DefaultUserTokenGroup		0:CertificateGroupType	Optional
0:Organizes	Object	0:<AdditionalGroup>		0:CertificateGroupType	Optional Placeholder
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

The *DefaultApplicationGroup Object* represents the default *CertificateGroup* for *Applications*. It is used to access the default *Application TrustList* and to define the *CertificateTypes* allowed for the *ApplicationInstanceCertificate*. This *Object* shall specify the *ApplicationCertificateType NodeId* (see 7.8.4.2) as a single entry in the *CertificateTypes* list or it shall specify one or more subtypes of *ApplicationCertificateType*.

The *DefaultHttpsGroup Object* represents the default *CertificateGroup* for HTTPS communication. It is used to access the default HTTPS *TrustList* and to define the *CertificateTypes* allowed for the *HTTPS Certificate*. This *Object* shall specify the *HttpsCertificateType NodeId* (see 7.8.4.3) as a single entry in the *CertificateTypes* list or it shall specify one or more subtypes of *HttpsCertificateType*.

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

Any additional *CertificateGroups* shall have a *BrowseName* where the *Name* is unique within the *CertificateGroupFolder*.

7.8.4 CertificateTypes

7.8.4.1 CertificateType

This type is an abstract base type for types that describe the purpose of a *Certificate*. This type is defined in Table 39.

Table 39 – CertificateType Definition

Attribute	Value				
BrowseName	0:CertificateType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*BaseObjectType* defined in OPC 10000-5.					
0:HasSubtype	ObjectType	0:ApplicationCertificateType	Defined in 7.8.4.2.		

0:HasSubtype	ObjectType	0:HttpsCertificateType	Defined in 7.8.4.3.
Conformance Units			
GDS Certificate Manager Pull Model			
Push Model for Global Certificate and TrustList Management			

7.8.4.2 ApplicationCertificateType

This type is an abstract base type for types that describe the purpose of an *ApplicationInstanceCertificate*. This type is defined in Table 40.

Table 40 – ApplicationCertificateType Definition

Attribute	**Value**				
BrowseName	0:ApplicationCertificateType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *CertificateType* defined in 7.8.4.					
0:HasSubtype	ObjectType	0:RsaMinApplicationCertificateType		Defined in 7.8.4.4.	
0:HasSubtype	ObjectType	0:RsaSha256ApplicationCertificateType		Defined in 7.8.4.5.	
0:HasSubtype	ObjectType	0:EccApplicationCertificateType		Defined in 7.8.4.6.	
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.3 HttpsCertificateType

This type is used to describe Certificates that are intended for use as HTTPS *Certificates*. This type is defined in Table 41.

Table 41 – HttpsCertificateType Definition

Attribute	**Value**				
BrowseName	0:HttpsCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*CertificateType* defined in 7.8.4.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.4 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 OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 42.

Table 42 – RsaMinApplicationCertificateType Definition

Attribute	**Value**				
BrowseName	0:RsaMinApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*ApplicationCertificateType* defined in 7.8.4.2					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.5 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 OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 43.

Table 43 – RsaSha256ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:RsaSha256ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*ApplicationCertificateType* defined in 7.8.4.2					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.6 EccApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC *Public Key*. *Applications* which support the ECC profiles (see OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 44.

Table 44 – EccApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*ApplicationCertificateType* defined in 7.8.4.2.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.7 EccNistP256ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC nistP256 *Public Key*. *Applications* which support the ECC NIST P256 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type or a *Certificate* of the EccNistP384ApplicationCertificateType defined in 7.8.4.8. This type is defined in Table 45.

Table 45 – EccNistP256ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccNistP256ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.8 EccNistP384ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC nistP384 *Public Key*. *Applications* which support the ECC NIST P384 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 46.

Table 46 – EccNistP384ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccNistP384ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.9 EccBrainpoolP256r1ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC brainpoolP256r1 *Public Key*. *Applications* which support the ECC brainpoolP256r1 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type or a *Certificate* of the *EccBrainpoolP384r1ApplicationCertificateType* defined in 7.8.4.10. This type is defined in Table 47.

Table 47 – EccBrainpoolP256r1ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccBrainpoolP256r1ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.10 EccBrainpoolP384r1ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC brainpoolP384r1 *Public Key*. *Applications* which support the ECC brainpoolP384r1 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 48.

Table 48 – EccBrainpoolP384r1ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccBrainpoolP384r1ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.11 EccCurve25519ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC curve25519 *Public Key*. *Applications* which support the ECC curve25519 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 49.

Table 49 – EccCurve25519ApplicationCertificateType Definition

Attribute	Value				
BrowseName	0:EccCurve25519ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.8.4.12 EccCurve448ApplicationCertificateType

This type is used to describe *Certificates* intended for use as an *ApplicationInstanceCertificate*. They shall have an ECC curve448 *Public Key*. *Applications* which support the ECC curve448 curve profiles (see OPC 10000-7) shall have a *Certificate* of this type. This type is defined in Table 50.

Table 50 – EccCurve448ApplicationCertificateType Definition

Attribute	Value

BrowseName	0:EccCurve448ApplicationCertificateType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*EccApplicationCertificateType* defined in 7.8.4.6.					
Conformance Units					
GDS Certificate Manager Pull Model					
Push Model for Global Certificate and TrustList Management					

7.9 Information Model for Pull Certificate Management

7.9.1 Overview

The *GlobalDiscoveryServer AddressSpace* used for *Certificate* management is shown in Figure 20. Most of the interactions between the *GlobalDiscoveryServer* and *Application* administrator or the *Client* will be via *Methods* defined on the *Directory* folder.

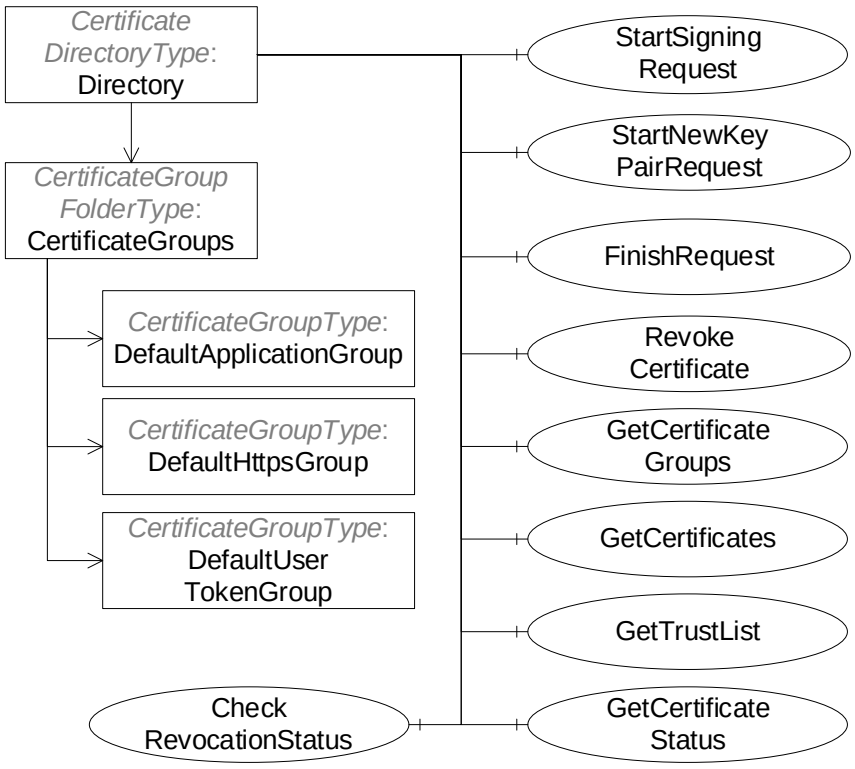

Figure 20 – The Certificate Management AddressSpace for the GlobalDiscoveryServer

7.9.2 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 51.

Table 51 – CertificateDirectoryType ObjectType Definition

Attribute	Value				
BrowseName	2:CertificateDirectoryType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 2:*DirectoryType* defined in 6.6.3.					
0:Organizes	Object	2:CertificateGroups		0:CertificateGroup FolderType	Mandatory
0:HasComponent	Method	2:StartSigningRequest	Defined in 7.9.3.		Mandatory
0:HasComponent	Method	2:StartNewKeyPairRequest	Defined in 7.9.4.		Mandatory
0:HasComponent	Method	2:FinishRequest	Defined in 7.9.5.		Mandatory
0:HasComponent	Method	2:RevokeCertificate	Defined in 7.9.6.		Optional
0:HasComponent	Method	2:GetCertificateGroups	Defined in 7.9.7.		Mandatory
0:HasComponent	Method	2:GetCertificates	Defined in 7.9.8.		Optional
0:HasComponent	Method	2:GetTrustList	Defined in 7.9.9.		Mandatory
0:HasComponent	Method	2:GetCertificateStatus	Defined in 7.9.10.		Mandatory
0:HasComponent	Method	2:CheckRevocationStatus	Defined in 7.9.11.		Optional
Conformance Units					
GDS Certificate Manager Pull Model					

The *CertificateGroups Object* organizes the *CertificateGroups* supported by the *CertificateManager*. It is described in 7.8.4.6. *CertificateManagers* shall support the *DefaultApplicationGroup* and may support the *DefaultHttpsGroup* or the *DefaultUserTokenGroup*. *CertificateManagers* may support additional *CertificateGroups* 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*.

CertificateGroups assigned by the *CertificateManager* to specific applications are persisted by *PullManagement Clients*. These *Clients* use the *NodeIds* to relate their local configuration to the *CertificateGroup* in the *CertificateManager*.

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 an entity with access to the *RegistrationAuthorityAdmin Role.* 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 *TrustLists* for an *Application*.

The *GetCertificates Method* returns a list of *Certificates* assigned to the *Application* for a *CertificateGroup.*

The *GetTrustList Method* returns a *NodeId* of a *TrustListType Object* that belongs to a *CertificateGroup* assigned to an *Application*.

The *GetCertificateStatus Method* checks whether the *Application* needs to update the *Certificate* identified in the call.

The *CheckRevocationStatus Method* checks the revocation status of a *Certificate.*

7.9.3 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 CertificateGroup 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. If the *CertificateRequest* is for an *ApplicationInstance Certificate* then it shall include all fields required by OPC 10000-6 such as the *subjectAltName*.
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 *CertificateGroup* that provides context for the request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*. If the *Application* does not currently belong to the requested *CertificateGroup* the *CertificateManager* shall verify that the *Application* is allowed to join the *CertificateGroup and* then, if permitted, add the *Application* to the *CertificateGroup.*The *CertificateGroup* verification and assignment may occur anytime before *FinishRequest* returns success*.*

The set of available *CertificateGroups* are found in the *CertificateGroups* folder described in 7.9.2. The *CertificateGroups* allowed for an *Application* are returned by the *GetCertificateGroups Method* (see 7.9.7).

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 *CertificateRequest*. The *subject*, *subjectAltName* and *Public Key* are copied into the new *Certificate*.

If the *certificateTypeId* is a subtype of *ApplicationCertificateType* the *subject* conforms to the requirements defined in OPC 10000-6. 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* shall have an organization (O=) field.

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.

The subject in the CSR may be ignored by the *CertificateManager.* The *CertificateManager* may update the subject to comply with policy requirements and to ensure global uniqueness.

Any bits set in *basicConstraints* or *extendedKeyUsage* fields in the CSR are ignored by the *CertificateManager*. The *CertificateManager* uses values that are appropriate and compliant with requirements defined for *Application Instance Certificates* in OPC 10000-6.

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* shall be called from an encrypted *SecureChannel* and from a *Session* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

If auditing is supported, the *CertificateManager* shall generate the *CertificateRequested AuditEventType* (see 7.9.12) 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	One or more of the certificateGroupId, *certificateTypeId* or *certificateRequest* arguments 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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 52 specifies the *AddressSpace* representation for the *StartSigningRequest Method*.

Table 52 – StartSigningRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	2:StartSigningRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.4 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 CertificateGroup 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* 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* 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 - PKCS #8 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 *CertificateGroup* that provides context for the request. If null the *CertificateManager* shall choose the *DefaultApplicationGroup*. If the *Application* does not currently belong to the requested *CertificateGroup* the *CertificateManager* shall verify that the *Application* is allowed to join the *CertificateGroup and* then, if permitted, add the *Application* to the *CertificateGroup.*

The set of available *CertificateGroups* are found in the *CertificateGroups* folder described in 7.9.2. The *CertificateGroups* allowed for an *Application* are returned by the *GetCertificateGroups Method* (see 7.9.7).

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* 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* is a logical domain used to qualify the *subject* 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* shall have an organization (O=) field.

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

The requested subject may be ignored by the *CertificateManager*. The *CertificateManager* may update the subject to comply with policy requirements and to ensure global uniqueness.

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* shall be called from an encrypted *SecureChannel* and from a *Session* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

If auditing is supported, the *CertificateManager* shall generate the *CertificateRequested AuditEventType* (see 7.9.12) 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* (deprecated).

Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	One or more of the certificateGroupId, *certificateTypeId*, *subjectName*, *domainNames or privateKeyFormat* parameters 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_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 53 specifies the *AddressSpace* representation for the *StartNewKeyPairRequest Method*.

Table 53 – StartNewKeyPairRequest Method AddressSpace Definition

Attribute	**Value**				
BrowseName	2:StartNewKeyPairRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.5 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 an entity with access to the *RegistrationAuthorityAdmin Role* has approved the request.

If the *Client* experiences a network failure while waiting for a completed request it may receive a *Bad_InvalidArgument* error when it calls the *Method* again. Recovering from this error is done by:

- If the *Client* originally called *StartSigningRequest* it can retrieve the *Certificate* by calling *GetCertificates* (see 7.9.8).
- If the *Client* originally called *StartNewKeyPairRequest* it shall restart the process by calling *StartNewKeyPairRequest again.*

This *Method* shall be called from an encrypted *SecureChannel* and from a *Session* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2). In addition, the *Client Certificate* shall be the same as the one used to call *StartSigningRequest* or *StartNewKeyPairRequest*.

If auditing is supported, the *CertificateManager* shall generate the *CertificateDeliveredAuditEventType* (see 7.9.13) 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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 54 specifies the *AddressSpace* representation for the *FinishRequest Method*.

Table 54 – FinishRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	2:FinishRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.6 RevokeCertificate

RevokeCertificate is used to revoke a *Certificate* issued by the *CertificateManager*.

When a *Certificate* is revoked it shall be removed from any *TrustLists* that it is in and *TrustLists* with the issuer *Certificate* shall be updated with the new CRL.

Certificates assigned to an *Application* are automatically revoked when the *UnregisterApplication Method* is called (see 6.6.8).

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role* (see 7.2).

Signature

```
RevokeCertificate (
    [in]  NodeId applicationId
    [in]  ByteString certificate
  );
```

Argument	Description
applicationId	The identifier assigned to the *Application* by the *CertificateManager*.
certificate	The DER encoded *Certificate* to revoke.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	The *applicationId* does not refer to a registered *Application*.
Bad_InvalidArgument	The *certificate* is not a *Certificate* for the specified *Application* that was issued by the *CertificateManager*.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 55 specifies the *AddressSpace* representation for the *RevokeCertificate Method*.

Table 55 – RevokeCertificate Method AddressSpace Definition

Attribute	Value				
BrowseName	2:RevokeCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.7 GetCertificateGroups

GetCertificateGroups returns the CertificateGroups 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 *CertificateGroups* assigned to the *Application*.

A *CertificateGroup* provides a *TrustList* and one or more *CertificateTypes* which may be assigned to an *Application*.

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 58 specifies the *AddressSpace* representation for the *GetCertificateGroups Method*.

Table 56 – GetCertificateGroups Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetCertificateGroups				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.8 GetCertificates

GetCertificates returns the *Certificates* assigned to *Application* and associated with the *CertificateGroup*.

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

Signature

```
GetCertificates(
    [in]  NodeId applicationId
    [in]  NodeId certificateGroupId
    [out] NodeId[] certificateTypeIds
    [out] ByteString[] certificates
);
```

Argument	Description
applicationId	The identifier assigned to the *Application* by the GDS.
certificateGroupId	An identifier for the *CertificateGroup* that the *Certificates* belong to. If null, the *CertificateManager* shall return the *Certificates* for all *CertificateGroups* assigned to the *Application*.
certificateTypeIds	The *CertificateTypes* that currently have a *Certificate* assigned. The length of this list is the same as the length as *certificates* list.
certificates	A list of DER encoded *Certificates* assigned to *Application*. This list only includes *Certificates* that are currently valid.

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* is not recognized or not valid for the *Application*.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

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

Table 57 – GetCertificates Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetCertificates				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.9 GetTrustList

GetTrustList is used to retrieve the *NodeId* of a *TrustList* 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 *CertificateGroup* that the *Application* belongs to. If null, the *CertificateManager* shall return the *trustListId* for a suitable default group for the *Application*.
trustListId	The *NodeId* for a *TrustList Object* that can be used to download the *TrustList* assigned to the *Application*.

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

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 58 specifies the *AddressSpace* representation for the *GetTrustList Method*.

Table 58 – GetTrustList Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetTrustList				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.10 GetCertificateStatus

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

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *CertificateAuthorityAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 7.2).

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 *CertificateGroup* 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*.

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.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 59 specifies the *AddressSpace* representation for the *GetCertificateStatus Method*.

Table 59 – GetCertificateStatus Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetCertificateStatus				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.11 CheckRevocationStatus

CheckRevocationStatus Method is used to check the revocation status of an *Certificate.*

Clients or *Servers* may use this *Method* if the issuer *Certificate* has a crlDistributionPoint extension, an authorityInformationAccess extension (see RFC 6960) or the *TrustList* is configured to require online *Certificate* revocation checks (see 7.8.2.1).

The *CertificateManager* will typically use a protocol such as OCSP (see RFC 6960) to verify the *Certificate* status using the endpoint in the CDP extension, however, it may also optimize performance by maintaining a cache of recently verified *Certificate* and/or maintaining it's own offline CRLs. The *validityTime* parameter provides guidance on how long a result can be kept in a local cache.

The caller shall perform all validation checks other than the revocation status check (see OPC 10000-4) on the *Certificate* before calling this *Method*. The *CertificateManager* shall check the *Signature* on the *Certificate* and may do additional validation.

This *Method* shall be called from an authenticated *SecureChannel*.

Signature

```
CheckRevocationStatus (
    [in]  ByteString certificate
```

```
    [out] StatusCode certificateStatus
    [out] UtcTime validityTime
  );
```

Argument	Description
INPUTS	
certificate	The DER encoded form of the Certificate to check.
OUTPUTS	
certificateStatus	The first error encountered when validating the *Certificate*.
validityTime	When the result expires and should be rechecked. DateTime.MinValue is this is unknown.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 60 specifies the *AddressSpace* representation for the *CheckRevocationStatus Method*.

Table 60 – CheckRevocationStatus Method AddressSpace Definition

Attribute	Value				
BrowseName	2:CheckRevocationStatus				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.9.12 CertificateRequestedAuditEventType

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

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

Its representation in the *AddressSpace* is formally defined in Table 61.

Table 61 – CertificateRequestedAuditEventType Definition

Attribute	Value				
BrowseName	2:CertificateRequestedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
0:HasProperty	Variable	2:CertificateGroup	0:NodeId	0:PropertyType	Mandatory
0:HasProperty	Variable	2:CertificateType	0:NodeId	0:PropertyType	Mandatory
Conformance Units					
GDS Certificate Manager Pull Model					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

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

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

7.9.13 CertificateDeliveredAuditEventType

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

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

Its representation in the *AddressSpace* is formally defined in Table 62.

Table 62 – CertificateDeliveredAuditEventType Definition

Attribute	Value				
BrowseName	2:CertificateDeliveredAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
0:HasProperty	Variable	2:CertificateGroup	0:NodeId	0:PropertyType	Mandatory
0:HasProperty	Variable	2:CertificateType	0:NodeId	0:PropertyType	Mandatory
Conformance Units					
GDS Certificate Manager Pull Model					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

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

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

7.10 Information Model for Push Certificate Management

7.10.1 Overview

If a *Server* supports *PushManagement* it is required to support an information model as part of its *AddressSpace*. It shall support the *ServerConfiguration Object* shown in Figure 21.

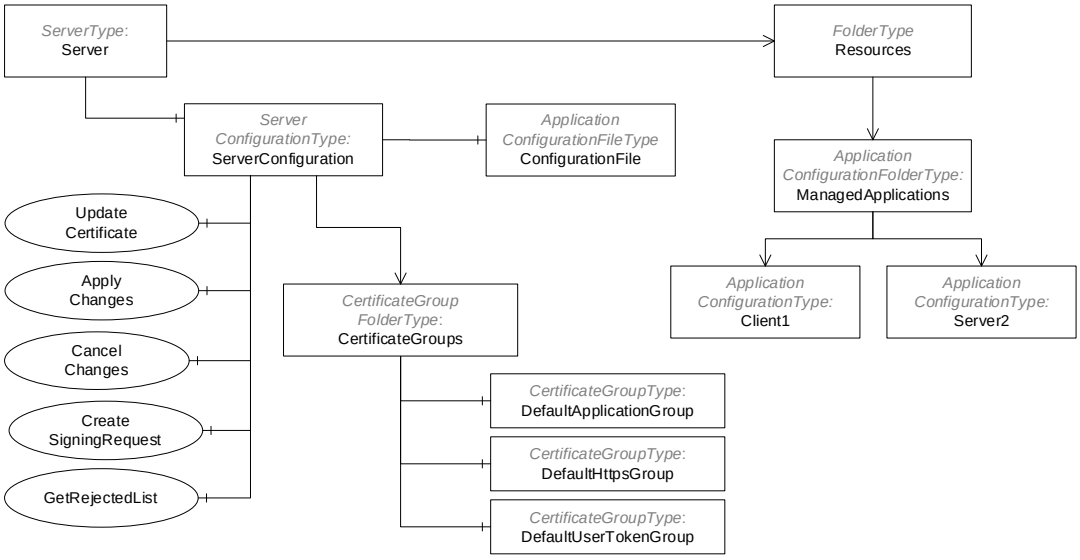

Figure 21 – The AddressSpace for the Server that supports Push Management

The *ServerConfiguration Object* is used to manage the Server. The *ManagedApplications Folder* collects *ApplicationConfiguration Objects* for other applications which the *Server* is able to manage. For example, a *Server* may have associated *Client* applications that do not support *PushManagement* so the *Server* can become a proxy for these *Clients*.

7.10.2 Transaction Lifecycle

The *CertificateGroups* and *TrustLists* used by a *Server* may be updated as part of a transaction where multiple *Methods* are invoked, however, no changes will have any effect until *ApplyChanges* is called (see 7.10.7). These transactions are created automatically and the *Server* returns *applyChangesRequired* =TRUE in a *Method* response to tell the *Client* that a transaction is active. *Servers* that do not support transactions return *applyChangesRequired* =FALSE and apply any changes before returning a *Method* response.

If a *Method* called within a transaction fails (e.g. a parameter was invalid) the transaction state shall not change and all previous changes are applied when *ApplyChanges* is called.

Once a transaction is created, a *Server* shall queue the changes in the order that they were requested within the current *Session*. When *ApplyChanges* is called the *Server* verifies that all the changes are consistent and can be applied without errors. If any errors are found then all changes are discarded. If no errors are found, the *Server* applies all changes.

If errors occur, they are reported in the *TransactionDiagnostics Object* (see 7.10.12).

The life cycle of a transaction is shown in Figure 22.

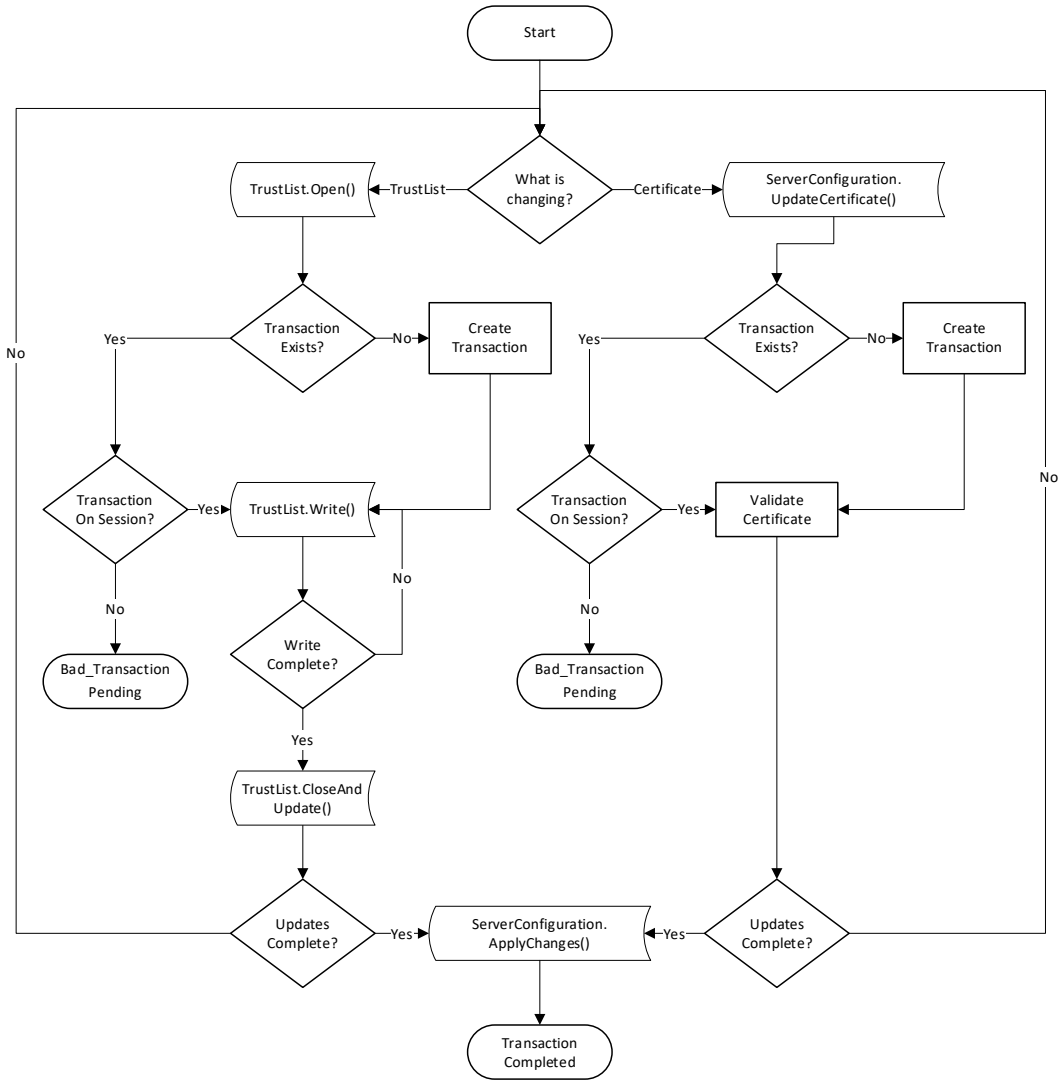

Figure 22 – The Transaction Lifecycle when using PushManagement

Servers that implement the transaction model shall support the *CancelChanges Method* and always set *applyChangesRequired* to TRUE.

Servers that support the transaction model are expected to support exactly one active transaction. Once a transaction has started in *Session* all other *Sessions* will not be able to modify *TrustLists* or *Certificates*. Transactions are automatically cancelled when the *Session* that created it is closed or when the *CancelChanges Method* is called.

If the transaction model is not supported and *applyChangesRequired* is TRUE then the behaviour of the *Server* for multiple changes is undefined.

If *applyChangesRequired* is FALSE then any changes are applied before the *Method* response is sent.

7.10.3 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 OPC 10000-5.

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 *Clients* with access to the *SecurityAdmin Role*.

Its representation in the *AddressSpace* is formally defined in Table 63.

Table 63 – ServerConfiguration Object Definition

Attribute	**Value**				
BrowseName	0:ServerConfiguration				
TypeDefinition	0:ServerConfigurationType defined in 7.10.4.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Conformance Units					
Push Model for Global Certificate and TrustList Management					

7.10.4 ServerConfigurationType

This type defines an *ObjectType* which represents the configuration of a *Server* which supports *PushManagement*. Its values are defined in Table 64. There is always a well-known instance in the *Server AddressSpace* (see 7.10.3) that can be use to configure a *Server*.

Table 64 – ServerConfigurationType Definition

Attribute	**Value**				
BrowseName	0:ServerConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**Type Definition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:ApplicationUri	0:UriString	0:PropertyType	Optional
0:HasProperty	Variable	0:ProductUri	0:UriString	0:PropertyType	Optional
0:HasProperty	Variable	0:ApplicationType	0:ApplicationType	0:PropertyType	Optional
0:HasProperty	Variable	0:ApplicationNames	0:LocalizedText[]	0:PropertyType	Optional
0:HasProperty	Variable	0:ServerCapabilities	0:String[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:SupportedPrivateKeyFormats	0:String[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:MaxTrustListSize	0:UInt32	0:PropertyType	Mandatory
0:HasProperty	Variable	0:MulticastDnsEnabled	0:Boolean	0:PropertyType	Mandatory
0:HasProperty	Variable	0:HasSecureElement	0:Boolean	0:PropertyType	Optional
0:HasProperty	Variable	0:SupportsTransactions	0:Boolean	0:PropertyType	Optional
0:HasProperty	Variable	0:InApplicationSetup	0:Boolean	0:PropertyType	Optional
0:HasComponent	Method	0:UpdateCertificate	See 7.10.5.		Mandatory
0:HasComponent	Method	0:GetCertificates	See 7.10.6.		Optional
0:HasComponent	Method	0:ApplyChanges	See 7.10.7.		Mandatory
0:HasComponent	Method	0:CancelChanges	See 7.10.9.		Optional
0:HasComponent	Method	0:CreateSigningRequest	See 7.10.8.		Mandatory
0:HasComponent	Method	0:GetRejectedList	See 7.10.10.		Mandatory
0:HasComponent	Method	0:ResetToServerDefaults	See 7.10.11.		Optional
0:HasComponent	Object	0:CertificateGroups		0:CertificateGroupFolderType	Mandatory
0:HasComponent	Object	0:TransactionDiagnostics		0:TransactionDiagnosticsType	Optional
Conformance Units					
Push Model for Global Certificate and TrustList Management					

The *CertificateGroups Object* organizes the *Certificate Groups* supported by the *Server*. It is described in 7.8.4.6. *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 *TrustList* for each interface. The second *TrustList* would be represented as a new *CertificateGroupType* Object organized by *CertificateGroups Folder.*

The *ApplicationUri Property* specifies the *ApplicationUri* assigned to the *Server*. It can be updated by a *Client* with access to the *SecurityAdmin Role*.

The *ApplicationNames Property* is a list of localized names for the application that may be used to when registering with a GDS.

The *ProductUri Property* specifies the *ProductUri* for the *Server* that appears in the *ApplicationDescription*. It is read-only.

The *ApplicationType Property* specifies the *ApplicationType* for the *Server* that appears in the *ApplicationDescription*. It is read-only.

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 *TrustList* 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*.

If *HasSecureElement* is TRUE then the *Server* has access to hardware based secure storage for the *PrivateKeys* associated with its *Certificates*.

If the *SupportsTransactions Property* is TRUE, the *Server* supports the transaction lifecyle defined in 7.10.2. If it is FALSE or not present, the *Server* only supports delaying application of changes until *ApplyChanges* is called.

If the *InApplicationSetup Property* is TRUE then the *Server* is in the application setup state described in G.2.The *UpdateCertificate Method* is used to update a *Certificate*.

The *GetCertificates Method* returns the *Certificates* assigned to each of the *CertificateTypes in a CertificateGroup.*

The *ApplyChanges Method* is used complete changes made to *CertificateGroups* and/or *TrustLists* within the context of a transaction.

The *CancelChanges Method* is used to cancel an existing transaction.

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

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 *TrustList*. This *Method* is the a shortcut for the *GetRejectedList Method* (see 7.8.3.2) on the *DefaultApplicationGroup CertificateGroup* (see 7.8.3.3).

The *ResetToServerDefaults Method* is used reset the *Server* security configuration to a default state.

The *TransactionDiagnostics Object* reports detailed error information for the current or most recently completed transaction. The *TransactionDiagnostics Object* is only visible to *Clients* with access to the *SecurityAdmin Role*.

7.10.5 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 7.10.8. 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 follow the validation process defined in OPC 10000-4 on the *Certificate* and all of the issuer *Certificates*. If errors occur the *Bad_SecurityChecksFailed* error is returned. Note that the validation process requires that the *TrustList* associated with the *CertificateGroup* already contain the *Issuer Certificates* and their CRLs or that the issuers support online CRL checks.

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* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

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* (PKCS #12 encoded and PKCS #8 Base64 encoded DER (see RFC 5958)). 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*.
Bad_TransactionPending	There is already a transaction active for another session.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 65 specifies the *AddressSpace* representation for the *UpdateCertificate Method*.

Table 65 – UpdateCertificate Method AddressSpace Definition

Attribute	Value				
BrowseName	0:UpdateCertificate				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

7.10.6 GetCertificates

GetCertificates returns the *Certificates* assigned to *CertificateTypes* associated with a *CertificateGroup*.

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

Signature

```
GetCertificates(
    [in] NodeId certificateGroupId
    [out] NodeId[] certificateTypeIds
    [out] ByteString[] certificates
);
```

Argument	Description
certificateGroupId	The identifier for the *CertificateGroup*.
certificateTypeIds	The *CertificateTypes* that currently have a *Certificate* assigned. The length of this list is the same as the length as *certificates* list. An empty list if the *CertificateGroup* does not have any *CertificateTypes*.
certificates	A list of DER encoded *Certificates* assigned to *CertificateGroup*. The *certificateType* for the *Certificate* is specified by the corresponding element in the *certificateTypes* parameter.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_InvalidArgument	The certificateGroupId is not valid.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 66 specifies the *AddressSpace* representation for the *GetCertificates Method*.

Table 66 – GetCertificates Method AddressSpace Definition

Attribute	Value				
BrowseName	0:GetCertificates				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.10.7 ApplyChanges

ApplyChanges is used to apply pending *Certificate* and *TrustList* updates and to complete a transaction as described in 7.10.2.

ApplyChanges returns *Bad_InvalidState* if any *TrustList* is still open for writing. No changes are applied and *ApplyChanges* can be called again after the *TrustList* is closed.

If a *Session* is closed or abandoned then the transaction is closed and all pending changes are discarded.

If *ApplyChanges* is called and there is no active transaction then the *Server* returns *Bad_NothingToDo.* If there is an active transaction, however, no changes are pending the result is *Good* and the transaction is closed.

When a *Server Certificate* or *TrustList* changes active *SecureChannels* are not immediately affected. This ensures the caller of *ApplyChanges* can get a response to the *Method* call. Once

the *Method* response is returned the *Server* shall force existing *SecureChannels* affected by the changes to renegotiate and use the new *Server Certificate* and/or *TrustLists.*

Servers may close *SecureChannels* without discarding any *Sessions* or *Subscriptions.* This will seem like a network interruption from the perspective of the *Client* and the *Client* reconnect logic (see OPC 10000-4) allows them to recover their *Session* and *Subscriptions*. Note that some *Clients* may not be able to reconnect because they are no longer trusted.

Other *Servers* may need to do a complete shutdown. In these cases, the *Server* shall advertise its intent to interrupt connections by setting the *SecondsTillShutdown* and *ShutdownReason Properties* in the *ServerStatus Variable*.

If a *TrustList* change only affects *UserIdentity* associated with a *Session* then *Servers* shall re-evaluate the UserIdentity and if it is no longer valid the *Session* and associated *Subscriptions* are closed.

This *Method* shall be called from an authenticated *SecureChannel* and from the *Session* that created the transaction and has access to the *SecurityAdmin Role* (see 7.2).

Signature

```
ApplyChanges();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.
Bad_NothingToDo	There is no active transaction.
Bad_BadSessionIdInvalid	The session is not valid for the active transaction.

Table 67 specifies the *AddressSpace* representation for the *ApplyChanges Method*.

Table 67 – ApplyChanges Method AddressSpace Definition

Attribute	Value				
BrowseName	0:ApplyChanges				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

7.10.8 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.

Servers shall support one active and one new key pair for each combination of *certificateGroupId* and *certificateTypeId*. If this *Method* is called multiple times with the same *certificateGroupId* and *certificateTypeId* then any previously generated new key pair, that has not been made active, is discarded. If a key pair is made active by a call to *UpdateCertificate* then the previously active key pair is deleted.

If *Certificate* associated with the *certificateGroupId* and *certificateTypeId* is deleted or replaced via *CreateSelfSignedCertificate* (see **Error! Reference source not found.**) or *DeleteCertificate* (see **Error! Reference source not found.**) then the new key pair is discarded.

The new key pair created with *CreateSigningRequest* shall be persisted and shall be available for *UpdateCertificate* even if it is called from a different *Session*.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

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 7.9.4.
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.* If the *CertificateRequest* is for an *ApplicationInstance Certificate* then it shall include all fields required by OPC 10000-6 such as the *subjectAltName*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	One or more of the *certificateTypeId, certificateGroupId,* nonce, or *subjectName* paremeters is not valid.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_TransactionPending	There is already a transaction active for another session.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 68 specifies the *AddressSpace* representation for the *CreateSigningRequest Method*.

Table 68 – CreateSigningRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	0:CreateSigningRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.10.9 CancelChanges

CancelChanges is used to tell the *Server* to discard changes to the *TrustLists* or *Certificates* which were waiting for the *Client* to *ApplyChanges*.

This *Method* shall be called from an authenticated *SecureChannel* and from the *Session* that created the transaction and has access to the *SecurityAdmin Role* (see 7.2).

Signature

```
CancelChanges();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 67 specifies the *AddressSpace* representation for the *CancelChanges Method*.

Table 69 – CancelChanges Method AddressSpace Definition

Attribute	Value				
BrowseName	0:CancelChanges				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

7.10.10 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* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 70 specifies the *AddressSpace* representation for the *GetRejectedList Method*.

Table 70 – GetRejectedList Method AddressSpace Definition

Attribute	Value				
BrowseName	0:GetRejectedList				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

7.10.11 ResetToServerDefaults

The *ResetToServerDefaults Method* resets the *Server* configuration to its default settings.

If the *Server* is running on a *Device* that supports OPC 10000-21 the *Device* is placed in a state where the *Onboarding* process has to restart. If the *Device* does not support OPC 10000-21, the *Server* repeats the Application Setup process described in Annex G.

After this *Method* completes the *Server* shall set the *ServerState* to SHUTDOWN and the *shutdownReason* to a localized message that warns *Clients* that their credentials may not work when the *Server* restarts. The *Server* should set the *secondsTillShutdown* to a time that gives the *Client* a chance to receive the response to this *Method*.

Note that the default configuration for a *Server* is set by configuration and is not necessarily the “factory default”. For example, a machine builder could update the default configuration to ensure that the *Server* can still communicate with other *Servers* within the machine after the reset.

The mechanisms for setting the default configuration is vendor specific.

This *Method* shall be called from an authenticated *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 7.2).

Signature

```
ResetToServerDefaults ();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not authenticated.

Table 71 specifies the *AddressSpace* representation for the *ResetToServerDefaults Method*.

Table 71 – ResetToServerDefaults Method AddressSpace Definition

Attribute	Value
BrowseName	0:ResetToServerDefaults

7.10.12 ApplicationConfigurationType

The *ApplicationConfigurationType ObjectType* defines a model which represents the configuration of a *Client or Server*. A *Server* acting as a proxy will add the *Objects* that represent the *Applications* it manages to the *ManagedApplications Object* (see 7.10.14).

Table 72 – ApplicationConfigurationType Definition

Attribute	Value				
BrowseName	0:ApplicationConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**Type Definition**	**Modelling Rule**
Subtype of the *ServerConfigurationType* defined in 7.10.4.					
0:HasProperty	Variable	0:Enabled	0:Boolean	0:PropertyType	Mandatory
0:HasProperty	Variable	0:ProductUri	0:UriString	0:PropertyType	Mandatory
0:HasProperty	Variable	0:ApplicationUri	0:UriString	0:PropertyType	Mandatory
0:HasProperty	Variable	0:ApplicationType	0:ApplicationType	0:PropertyType	Mandatory
Conformance Units					
Managed Application Configuration					

The *Enabled Property* indicates whether the *Application* is enabled. If FALSE the *Application* will not run. If TRUE the *Application* runs.

The *ProductUri Property* is the unique identifier for the product. *Applications* running on different *Devices* with the same *ProductUri* are based on the same software.

The *ApplicationUri Property* is the unique identifier for the *Application* which is not the same as the *ProductInstanceUri* which identifiers the *Device* that is executing the *Application*.

The *ApplicationType Property* specifies whether the *Application* is a *Client*, a *Server* or both *Applications* which do not support OPC UA specify an *ApplicationType* of *Client*.

An LDS is exposed an *ApplicationConfiguration Object* for it with *ApplicationType* set to *DiscoveryServer*. *Applications* which do not support OPC UA specify an *ApplicationType* of *Client*.

The *Application* may require software updates. In this case, the software update model described in OPC 10000-100 specifies an instance of the *SoftwareUpdateType* that may be added to the *ApplicationConfiguration* instance.

7.10.13 ApplicationConfigurationFolderType

A *Folder* for *ApplicationConfiguration Objects* which a *Server* exposes in its *AddressSpace*.

Table 73 – ApplicationConfigurationFolderType Definition

Attribute	Value

BrowseName	0:ApplicationConfigurationFolderType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**Type Definition**	**Modelling Rule**
Subtype of the *FolderType* defined in OPC 10000-5.					
0:Organizes	Object	0:<ApplicationName>		0:ApplicationConfigurationType	OptionalPlaceholder
Conformance Units					
Managed Application Configuration					

7.10.14 ManagedApplications

This *Object* allows access to the application configurations and it is the target of an *Organizes* reference from the *Resources Object* defined in OPC 10000-5.

Its representation in the *AddressSpace* is formally defined in Table 74.

Table 74 – ManagedApplications Object Definition

Attribute	**Value**				
BrowseName	0:ManagedApplications				
TypeDefinition	0:ApplicationConfigurationFolderType defined in 7.10.13.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Conformance Units					
Managed Application Configuration					

7.10.15 TransactionDiagnosticsType

This type defines an *ObjectType* which represents the diagnostics for the last transaction (see 7.10.1. If no transaction has started the values of all *Variables* have a status of *Bad_OutOfService.* All existing results are discarded when a new transaction starts.

Table 75 – TransactionDiagnosticsType Definition

Attribute	**Value**				
BrowseName	0:TransactionDiagnosticsType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**Type Definition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:StartTime	0:UtcTime	0:PropertyType	Mandatory
0:HasProperty	Variable	0:EndTime	0:UtcTime	0:PropertyType	Mandatory
0:HasProperty	Variable	0:Result	0:StatusCode	0:PropertyType	Mandatory
0:HasProperty	Variable	0:AffectedTrustLists	0:NodeId []	0:PropertyType	Mandatory
0:HasProperty	Variable	0:AffectedCertificateGroups	0:NodeId []	0:PropertyType	Mandatory
0:HasProperty	Variable	0:Errors	0:TransactionErrorType []	0:PropertyType	Mandatory
Conformance Units					
Push Model for Global Certificate and TrustList Management					

The *StartTime Property* indicates when transaction started. It has a status of *Bad_OutOfService* if a transaction has not started

The *EndTime Property* indicates when transaction ended. It has a value of *DateTime.MinValue* if the transaction has not completed.

The *Result Property* indicates the overall transaction result. It has a status of Bad_InvalidState if a transaction has started but not completed. If the transaction has completed the status is *Good* and the value is the *StatusCode* that was returned from the *ApplyChanges Method*. If the *CancelChanges Method* was called the value is Bad_RequestCancelledByClient.

The *AffectedTrustLists Property* specifies the *NodeIds* of the *TrustLists* that are included in the transaction. It is updated each time as soon as a *TrustList* is added to the transaction.

The *AffectedCertificateGroups Property* specifies the *NodeIds* of the *CertificateGroups* are included in the transaction. It is updated each time as soon as a *CertificateGroup* is added to the transaction.The *Errors Property* has a list of errors that occurred when the changes were applied. Empty if no errors occurred. The *TransactionErrorType* is defined in 7.10.16.

7.10.16 TransactionErrorType

This type defines a *DataType* which stores an error that occurred when processing a transaction. Its values are defined in Table 76.

Table 76 – TransactionErrorType Structure

Name	Type	Description
TransactionErrorType	Structure	Subtype of the *Structure DataType* defined in OPC 10000-5
targetId	NodeId	The *NodeId* of the Object that had the error. It is either a *TrustListId* or a *CertificateGroupId*.
error	StatusCode	The code describing the error.
message	LocalizedText	A description of the error. It should include enough information to allow the *Client* to understand which *Certificate(s)* and/or *CRL(s)* are the source of the problem.

Its representation in the *AddressSpace* is defined in Table 77.

Table 77 – TransactionErrorType Definition

Attribute	Value				
BrowseName	0:TransactionErrorType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of the 0:*Structure DataType* defined in OPC 10000-5.					
Conformance Units					
Push Model for Global Certificate and TrustList Management					

7.10.17 CertificateUpdateRequestedAuditEventType

This event is raised when the *UpdateCertificate Method* is called

Its representation in the *AddressSpace* is formally defined in Table 78.

Table 78 – CertificateUpdateRequestedAuditEventType Definition

Attribute	Value				
BrowseName	0:CertificateUpdateRequestedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
Conformance Units					
Push Model for Global Certificate and TrustList Management					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

7.10.18 CertificateUpdatedAuditEventType

This event is raised when a *Certificate* is actually changed.

This is the result of a sucessful call to *UpdateCertificate* or *ApplyChanges* on a *ServerConfigurationType Object*.

Its representation in the *AddressSpace* is formally defined in Table 79.

Table 79 – CertificateUpdatedAuditEventType Definition

Attribute	Value				
BrowseName	0:CertificateUpdatedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:CertificateGroup	0:NodeId	0:PropertyType	Mandatory
0:HasProperty	Variable	0:CertificateType	0:NodeId	0:PropertyType	Mandatory
Conformance Units					
Push Model for Global Certificate and TrustList Management					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

The *SourceNode Property* for *Events* of this type shall be assigned to the *NodeId* of the *Object* with the *Method* that triggered the *Event*.

The *CertificateGroup Property* specifies the *CertificateGroup* 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 *AuthorizationServices* 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 *PushManagement*. In *PullManagement*, 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 *PushManagement* 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 *AuthorizationServices* and/or *Brokers* and are not *Certificates* with private keys. *Certificate* distribution is managed by the *Certificate* management model described in 7. For example, *AuthorizationService*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 Roles and Privileges

KeyCredentialServices restrict access to many of the features they provide. These restrictions are described either by referring to well-known *Roles* which a *Session* must have access to or by referring to *Privileges* which are assigned to *Sessions* using mechanisms other than the well-known *Roles*. The well-known *Roles* used for a *KeyCredentialService* are listed in Table 80.

Table 80 – Well-known Roles for a KeyCredentialService

Name	Description
KeyCredentialAdmin	This *Role* grants rights to request or revoke any *KeyCredential*.
SecurityAdmin	This *Role* grants the right to change the security configuration of a *KeyCredentialService.*

The well-known *Roles* for *Server* managed by a *KeyCredentialService* are listed in Table 81.

Table 81 – Well-known Roles for Server managed by a KeyCredentialService

Name	Description
SecurityAdmin	For *PushManagement*, this *Role* grants the right to change the security configuration of a *Server* managed by a *KeyCredentialService.*

The *Privileges* used for a *KeyCredentialService* are listed in Table 82.

Table 82 – Privileges for a KeyCredentialService

Name	Description
ApplicationSelfAdmin	This *Privilege* grants an *OPC UA Application* the right to request its own *KeyCredentials*. The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application.*
ApplicationAdmin	This *Privilege* grants rights to request *KeyCredentials* for one or more *OPC UA Applications.* The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application* and the set of *OPC UA Applications* that it is authorized to manage.

8.3 Pull Management

Pull management is performed by using a *KeyCredentialManagement Object* (see 8.5.4). It allows *Clients* to request credentials for *AuthorizationServices* or *Brokers* which are supported by the *KeyCredentialService*. The interactions between the *Client* and the *KeyCredentialService* during *PullManagement* are illustrated in Figure 23.

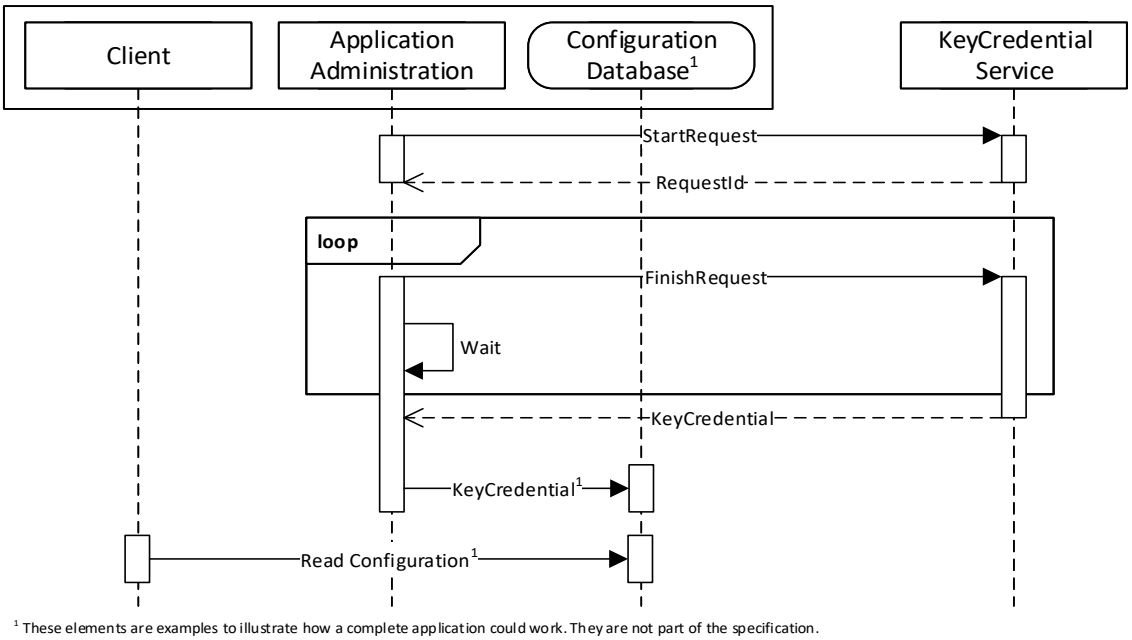

Figure 23 – The Pull Model for KeyCredential Management

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 *FinishRequest 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 shall only be returned to applications which are authorized by the *KeyCredentialService*.

Security in *PullManagement* requires an encrypted channel and *Clients* with access to the *KeyCredentialAdmin Role,* the *ApplicationAdmin Priviledge* or the *ApplicationSelfAdmin Priviledge*.

8.4 Push Management

Push management is performed by using a *KeyCredentialConfiguration Object* (see 8.6.5) which is a component of the *KeyCredentialConfigurationFolder Object* which, in turn, is component of the *ServerConfiguration Object* in a *Server*. The interactions between the Administration application and the *KeyCredentialService* during *PushManagement* are illustrated in Figure 24.

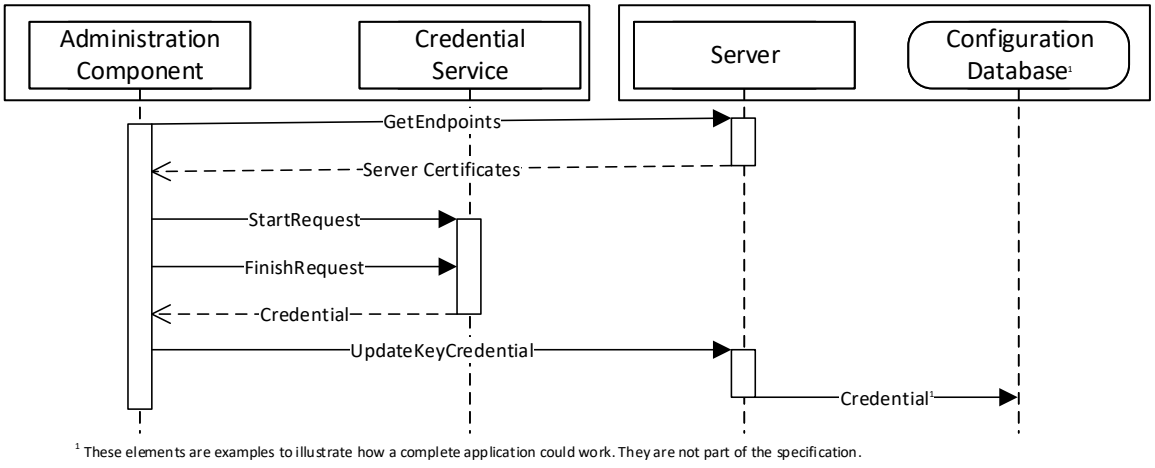

Figure 24 – The Push Model for KeyCredential Management

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 8.3). 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 *PushManagement* model, requires an encrypted channel and *Clients* with acccess to the *SecurityAdmin Role*.

8.5 Information Model for Pull Management

8.5.1 Overview

The *AddressSpace* used for *PullManagement* is shown in Figure 25. *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.

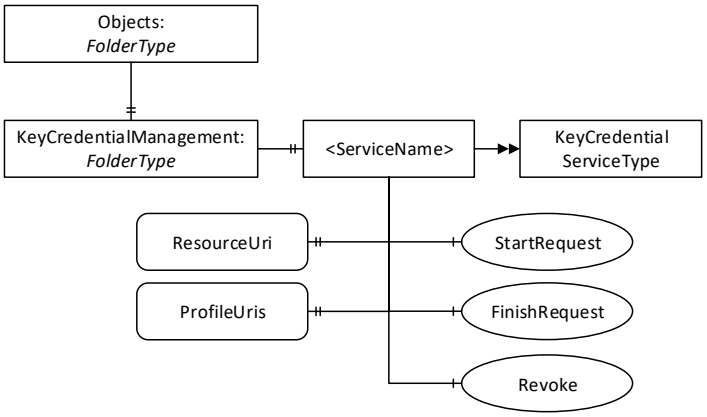

Figure 25 – The Address Space used for Pull KeyCredential Management

8.5.2 KeyCredentialManagementFolderType

This *ObjectType* represents a *Folder* that contains *KeyCredentialService Objects* which may be accessed via the *Server*. It is defined in Table 83.

Table 83 – KeyCredentialManagementFolderType Definition

Attribute	**Value**			
BrowseName	2:KeyCredentialManagementFolderType			
IsAbstract	False			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*FolderType* defined in OPC 10000-5.				
0:HasComponent	Object	2:<ServiceName>	2:KeyCredentialServiceType	OptionalPlaceholder
Conformance Units				
GDS Key Credential Service Pull Model				

8.5.3 KeyCredentialManagement

This *Object* 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 OPC 10000-5. It is defined in Table 84.

Table 84 – KeyCredentialManagement Object Definition

Attribute	**Value**			
BrowseName	2:KeyCredentialManagement			
TypeDefinition	2:*KeyCredentialManagementFolderType* defined in 8.5.2.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Conformance Units				
GDS Key Credential Service Pull Model				

8.5.4 KeyCredentialServiceType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows the management of *KeyCredentials*. It is defined in Table 85.

Table 85 – KeyCredentialServiceType Definition

Attribute	**Value**				
BrowseName	2:KeyCredentialServiceType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	2:ResourceUri	0:String	0:PropertyType	Mandatory
0:HasProperty	Variable	2:ProfileUris	0:String[]	0:PropertyType	Mandatory
0:HasProperty	Variable	2:SecurityPolicyUris	0:String[]	0:PropertyType	Optional
0:HasComponent	Method	2:StartRequest		Defined in 8.5.5.	Mandatory

0:HasComponent	Method	2:FinishRequest		Defined in 8.5.6.	Mandatory
0:HasComponent	Method	2:Revoke		Defined in 8.5.7.	Optional
Conformance Units					
GDS Key Credential Service Pull Model					

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

The *ProfileUris Property* specifies URIs assigned in OPC 10000-7 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 *SecurityPolicyUris Property* is the list of *SecurityPolicies* that may be used when encrypting the *KeyCredentials*. One of these URIs is passed in the *StartRequest Method*. If not present, The *Server* shall support all of the *SecurityPoliciesUris* returned by *GetEndpoints*,

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*.

8.5.5 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* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *KeyCredentialAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 8.2).

Signature

```
StartRequest (
    [in]  String      applicationUri
    [in]  ByteString  publicKey
    [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 6.6.6). If the requestor is not the same as the application used to create the *Secure Channel* then a *Certificate* should be provided.
publicKey	A *Public Key* used to encrypt the returned *KeyCredential* secret. For RSA *SecurityPolicies* this is the DER encoded form of an X.509 v3 *Certificate* as described in OPC 10000-6. For ECC *SecurityPolicies* this is an ephemeral key created by the owner of the *KeyCredentials*. 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 8.5.6).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 86 specifies the *AddressSpace* representation for the *StartRequest Method*.

Table 86 – StartRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	2:StartRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

8.5.6 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 *credentialId* is the identifier, such as a user name, which often needs to be presented when using the *credentialSecret*.

The *credentialSecret* is a UA Binary encoded form of one of the *EncryptedSecret DataTypes* defined in OPC 10000-4. If the *securityPolicyUri* requires an RSA *Certificate* then the *RsaEncryptedSecret* DataType is used. If the *securityPolicyUri* requires an ECC *Certificate* then the *EccEncryptedSecret* DataType is used.

The *Signing Certificate* is owned by the source of the *KeyCredential.* The *KeyCredentialService* determines the most appropriate *Certificate* to use.

If the return code is *Bad_RequestNotComplete* then the request has not been processed and the *Client* should call again. It is expected that a *Client* will periodically call this *Method* until the *KeyCredentialService* has completed the request.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *KeyCredentialAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 8.2)*. In addition,* this *Method* shall only be called *SecureChannel* using that same *Certificate* that *Client* used to call *StartRequest*.

Signature

```
FinishRequest (
    [in]  NodeId     requestId
    [in]  Boolean    cancelRequest
    [out] String     credentialId
    [out] ByteString credentialSecret
    [out] String certificateThumbprint
    [out] String securityPolicyUri
    [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* used to encrypt the secret for RSA *SecurityPolicies*. Set to NULL for ECC *SecurityPolicies*.
securityPolicyUri	The *SecurityPolicy* used to create the *credentialSecret*.
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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 87 specifies the *AddressSpace* representation for the *FinishRequest Method*.

Table 87 – FinishRequest Method AddressSpace Definition

Attribute	Value				
BrowseName	2:FinishRequest				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

8.5.7 Revoke

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

KeyCredentials shall be deleted when revoked.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *KeyCredentialAdmin Role,* the *ApplicationAdmin Privilege*, or the *ApplicationSelfAdmin Privilege* (see 8.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 88 specifies the *AddressSpace* representation for the *RevokeCredential Method*.

Table 88 – Revoke Method AddressSpace Definition

Attribute	Value				
BrowseName	2:Revoke				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	0:InputArguments	Argument[]	0:PropertyType	Mandatory

8.5.8 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 90.

Table 89 – KeyCredentialAuditEventType Definition

Attribute	Value				
BrowseName	0:KeyCredentialAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*AuditUpdateMethodEventType* defined in OPC 10000-5.					
0:HasProperty	Variable	ResourceUri	String	0:PropertyType	Mandatory
Conformance Units					
GDS Key Credential Service Pull Model					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

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

8.5.9 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 *StartRequest Method* call.

Its representation in the *AddressSpace* is formally defined in Table 90.

Table 90 – KeyCredentialRequestedAuditEventType Definition

Attribute	Value				
BrowseName	2:KeyCredentialRequestedAuditEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*KeyCredentialAuditEventType* defined in 8.5.8.					
Conformance Units					
GDS Key Credential Service Pull Model					

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

8.5.10 KeyCredentialDeliveredAuditEventType

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

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

Its representation in the *AddressSpace* is formally defined in Table 91.

Table 91 – KeyCredentialDeliveredAuditEventType Definition

Attribute	Value				
BrowseName	2:KeyCredentialDeliveredAuditEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*KeyCredentialAuditEventType* defined in 8.5.8.					
Conformance Units					
GDS Key Credential Service Pull Model					

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

8.5.11 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 92.

Table 92 – KeyCredentialRevokedAuditEventType Definition

Attribute	Value				
BrowseName	2:KeyCredentialRevokedAuditEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*KeyCredentialAuditEventType* defined in 8.5.8.					
Conformance Units					
GDS Key Credential Service Pull Model					

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

8.6 Information Model for Push Management

8.6.1 Overview

The *AddressSpace* used for *PushManagement* is shown in Figure 26. *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 *KeyCredentialConfiguration Folder* is a well-known *Object* that appears in the *AddressSpace* of any *Server* which supports *KeyCredential* management.

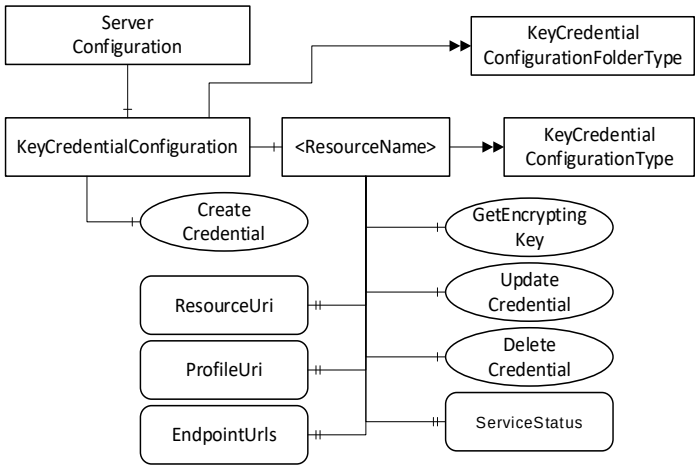

Figure 26 – The Address Space used for Push KeyCredential Management

8.6.2 KeyCredentialConfigurationFolderType

This *ObjectType* is the *TypeDefinition* for an *Folder Object* that contains the *KeyCredentialConfiguration Objects* which may be accessed via the *Server*.

Table 93 – KeyCredentialConfigurationFolderType Definition

Attribute	Value
BrowseName	0:KeyCredentialConfigurationFolderType
IsAbstract	False

References	NodeClass	BrowseName	TypeDefinition	Modelling Rule
Subtype of the 0:*FolderType* defined in OPC 10000-5.				
0:HasComponent	Object	0:<ServiceName>	0:KeyCredentialConfigurationType	Optional Placeholder
0:HasComponent	Method	0:CreateCredential	Defined in 8.6.3.	Optional
Conformance Units				
GDS Key Credential Service Push Model				

8.6.3 CreateCredential

CreateCredential is used to add a new *KeyCredentialConfiguration Object*.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 8.2).

Signature

```
CreateCredential (
   [in]  String    name
   [in]  String    resourceUri
   [in]  String profileUri
   [in]  String[]  endpointUrls
   [out] NodeId credentialNodeId
);
```

Argument	Description
name	This the *BrowseName* of the new *Object*.
resourceUri	The *resourceUri* uniquely identifies the resource that accepts the *KeyCredentials*. A valid URI shall be provided.
profileUri	The specified URI assigned in OPC 10000-7 to the protocol used to communicate with the resource identified by the *resourceUri*. A valid URI shall be provided.
endpointUrls	The specifies URLs used by the *Server* to communicate with the resource identified by the *resourceUri*. Valid URLs shall be provided.
credentialNodeId	A unique identifier for the new *KeyCredentialConfiguration Object Node*.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The *resourceUri, profileUri,* or one or more *endpointUrls* are not valid.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 94 specifies the *AddressSpace* representation for the *CreateCredential Method*.

Table 94 – CreateCredential Method AddressSpace Definition

Attribute	Value				
BrowseName	0:CreateCredential				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	Argument[]	0:PropertyType	Mandatory

8.6.4 KeyCredentialConfiguration

This *Object* is an instance of *FolderType.* It contains The *Objects* which may be accessed via the *Server*. It is the target of an *HasComponent* reference from the *ServerConfiguration Object* defined in 7.10.3. It is defined in Table 95.

Table 95 – KeyCredentialConfiguration Object Definition

Attribute	Value				
BrowseName	0:KeyCredentialConfiguration				
TypeDefinition	0:*KeyCredentialConfigurationFolderType* defined in 8.6.2.				
References	**NodeClass**	**BrowseName**	**TypeDefinition**		**Modelling Rule**

Conformance Units
GDS Key Credential Service Push Model

8.6.5 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 96.

Table 96 – KeyCredentialConfigurationType Definition

Attribute	Value				
BrowseName	0:KeyCredentialConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:ResourceUri	0:String	0:PropertyType	Mandatory
0:HasProperty	Variable	0:ProfileUri	0:String	0:PropertyType	Mandatory
0:HasProperty	Variable	0:EndpointUrls	0:String[]	0:PropertyType	Optional
0:HasProperty	Variable	0:CredentialId	0:String	0:PropertyType	Optional
0:HasProperty	Variable	0:ServiceStatus	0:StatusCode	0:PropertyType	Optional
0:HasComponent	Method	0:GetEncryptingKey		Defined in 8.6.6.	Optional
0:HasComponent	Method	0:UpdateCredential		Defined in 8.6.7.	Optional
0:HasComponent	Method	0:DeleteCredential		Defined in 8.6.8.	Optional
Conformance Units					
GDS Key Credential Service Push Model					

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 *CredentialId Property* is the identifier, such as a user name, which often needs to be presented when using the credentialSecret.

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 OPC 10000-4 for a complete list of standard *StatusCodes*.

The *GetEncryptingKey Method* is used request a *Public Key* that can be used to encrypt the *KeyCredentials*.

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*.

8.6.6 GetEncryptingKey

GetEncryptingKey is used to request a key that can be used to encrypt a *KeyCredential*.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 8.2).

Signature

```
GetEncryptingKey(
    [in]  String    credentialId
    [in]  String    requestedSecurityPolicyUri
    [out] ByteString publicKey
```

```
    [out] String    revisedSecurityPolicyUri
  );
```

Argument	Description
credentialId	The unique identifier associated with the *KeyCredential*.
requestedSecurityPolicyUri	The *SecurityPolicy* used to encrypt the secret. If not specified the *Server* chooses a suitable default.
publicKey	The Public Key used to encrypt the secret. The format depends on the *SecurityPolicyUri*.
revisedSecurityPolicyUri	The *SecurityPolicy* used to encrypt the secret. It also specifies the contents of the *publicKey*. This may be different from the *requestedSecurityPolicyUri.*

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidArgument	The credentialId is not valid.
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 97 specifies the *AddressSpace* representation for the *GetEncryptingKey Method*.

Table 97 – GetEncryptingKey Method AddressSpace Definition

Attribute	Value				
BrowseName	0:GetEncryptingKey				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

8.6.7 UpdateCredential

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

The *KeyCredential* secret may be encrypted using the key returned by *GetEncryptingKey*. The *SecurityPolicyUri* species the algorithm used for encryption. The format of the encrypted data is described in 8.5.6.

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 8.2).

Signature

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

Argument	Description
credentialId	The *credentialId* is the identifier, such as a user name, which often needs to be presented when using the credentialSecret.
credentialSecret	The secret associated with the *KeyCredential*.
certificateThumbprint	The thumbprint of the *Certificate* used to encrypt the secret. For RSA *SecurityPolicies* this shall be one of the *Application Instance Certificates* assigned to the *Server*. For ECC *SecurityPolicies* this field is not specified.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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 99 specifies the *AddressSpace* representation for the *UpdateKeyCredential Method*.

Table 98 – UpdateCredential Method AddressSpace Definition

Attribute	Value				
BrowseName	0:UpdateCredential				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory

8.6.8 DeleteCredential

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

This *Method* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *SecurityAdmin Role* (see 8.2).

Signature

```
DeleteCredential();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_UserAccessDenied	The current user does not have the rights required.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 98 specifies the *AddressSpace* representation for the *DeleteKeyCredential Method*.

Table 99 – DeleteCredential Method AddressSpace Definition

Attribute	Value				
BrowseName	0:DeleteCredential				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

8.6.9 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 100.

Table 100 – KeyCredentialUpdatedAuditEventType Definition

Attribute	Value				
BrowseName	0:KeyCredentialUpdatedAuditEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*KeyCredentialAuditEventType* defined in 8.5.8.					
Conformance Units					
Push Model for KeyCredential Service					

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

8.6.10 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 101.

Table 101 – KeyCredentialDeletedAuditEventType Definition

Attribute	**Value**				
BrowseName	0:KeyCredentialDeletedAuditEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the 0:*KeyCredentialAuditEventType* defined in 8.5.8.					
Conformance Units					
GDS Key Credential Service Push Model					

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

9 AuthorizationServices

9.1 Overview

AuthorizationServices provide *Access Tokens* to *Clients* that may use them to access resources. A *Server,* such as a GDS*,* with *AuthorizationService* capabilities may support one or more *AuthorizationService Objects* (see 9.6.2) which may represent an internal *AuthorizationService* or be an API to an external *AuthorizationService*. The *AuthorizationService* is best used in conjunction with the *Role* model defined in OPC 10000-5. 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 27.

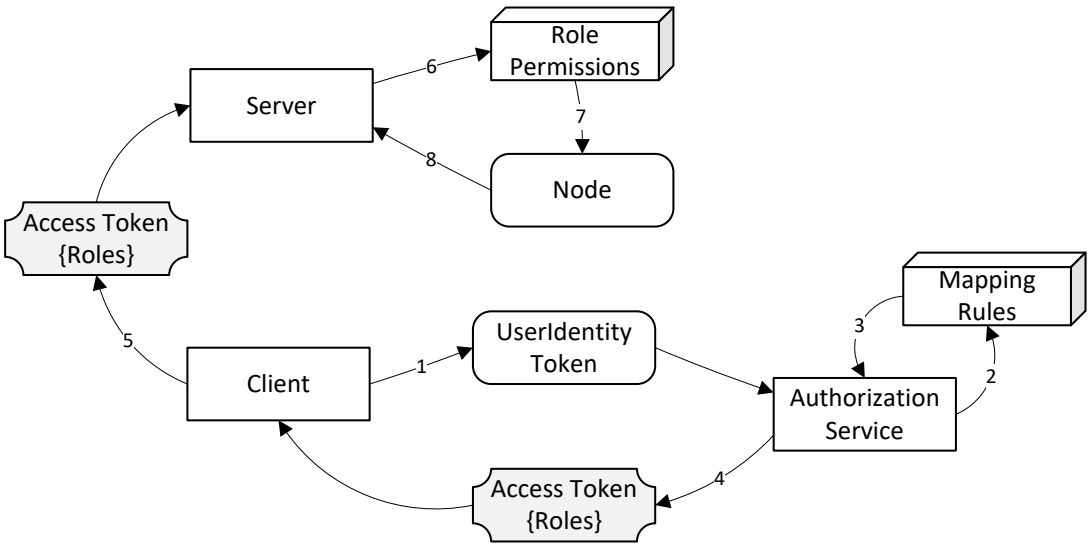

Figure 27 – Roles and AuthorizationServices

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 *AuthorizationServices* described in OPC 10000-4. The Chained use case is an implementation of the ‘Direct’ model.

9.2 Roles and Privileges

AuthorizationServices restrict access to many of the features they provide. These restrictions are described either by referring to well-known *Roles* which a *Session* must have access to or

by referring to *Privileges* which are assigned to *Sessions* using mechanisms other than the well-known *Roles*. The well-known *Roles* for an *AuthorizationService* are listed in Table 102.

Table 102 – Well-known Roles for an AuthorizationService

Name	Description
AuthorizationServiceAdmin	This *Role* grants the right to manage the configuration of an *AuthorizationService*.
SecurityAdmin	This *Role* grants the right to change the security configuration of an *AuthorizationService*.

The *Privileges* for an *AuthorizationService* are listed in Table 103.

Table 103 – Privileges for an AuthorizationService

Name	Description
AccessTokenRequestor	This *Privilege* grants an *OPC UA Application* the right to request *AccessTokens*. The *Certificate* used to create the *SecureChannel* is used to determine the identity of the *OPC UA Application.* A *KeyCredential* (see 0) provided as a *UserIdentityToken* may also be used to determine if the *Client* has access to this *Privilege*.

9.3 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 28.

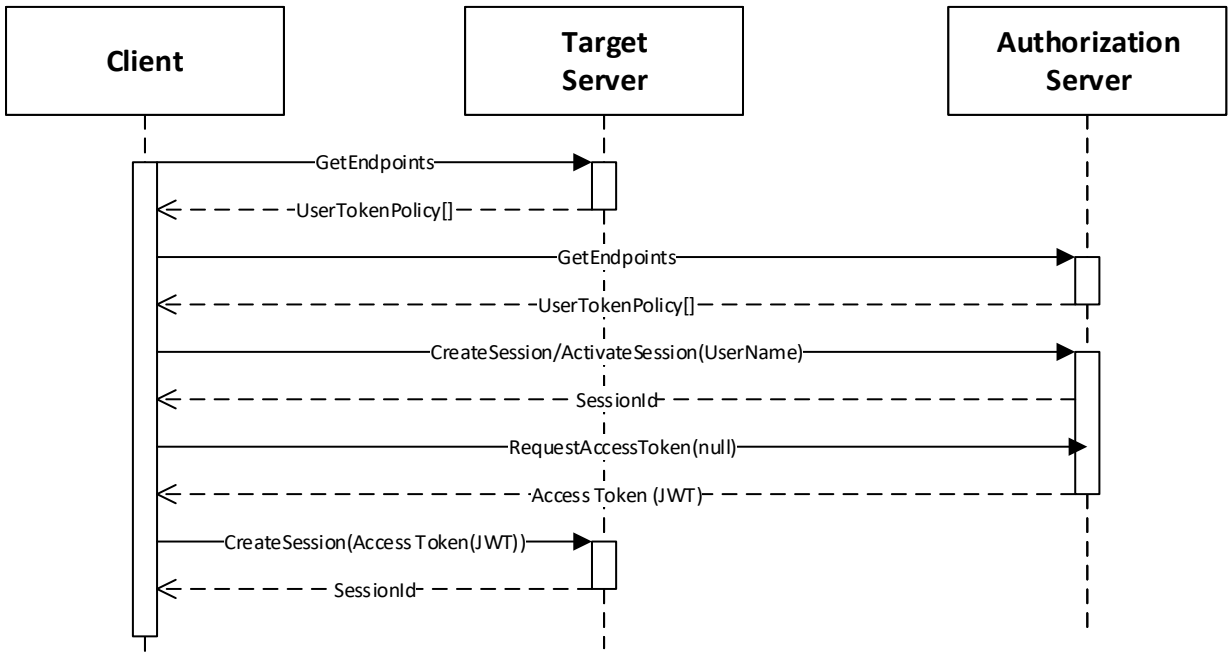

Figure 28 – Implicit Authorization

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 then 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 user credentials used can be any type of user credential including X.509 and JWT.

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 OPC 10000-3).

Once the *Client* has the *Access Token,* it passes the *Access Token* to the Target *Server* which validates the *Access Token*, as described in OPC 10000-4. The Target *Server* is configured out-of-band with the *Certificate* needed to validate the *Access Tokens* issued by the Authorization *Server*.

9.4 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 29.

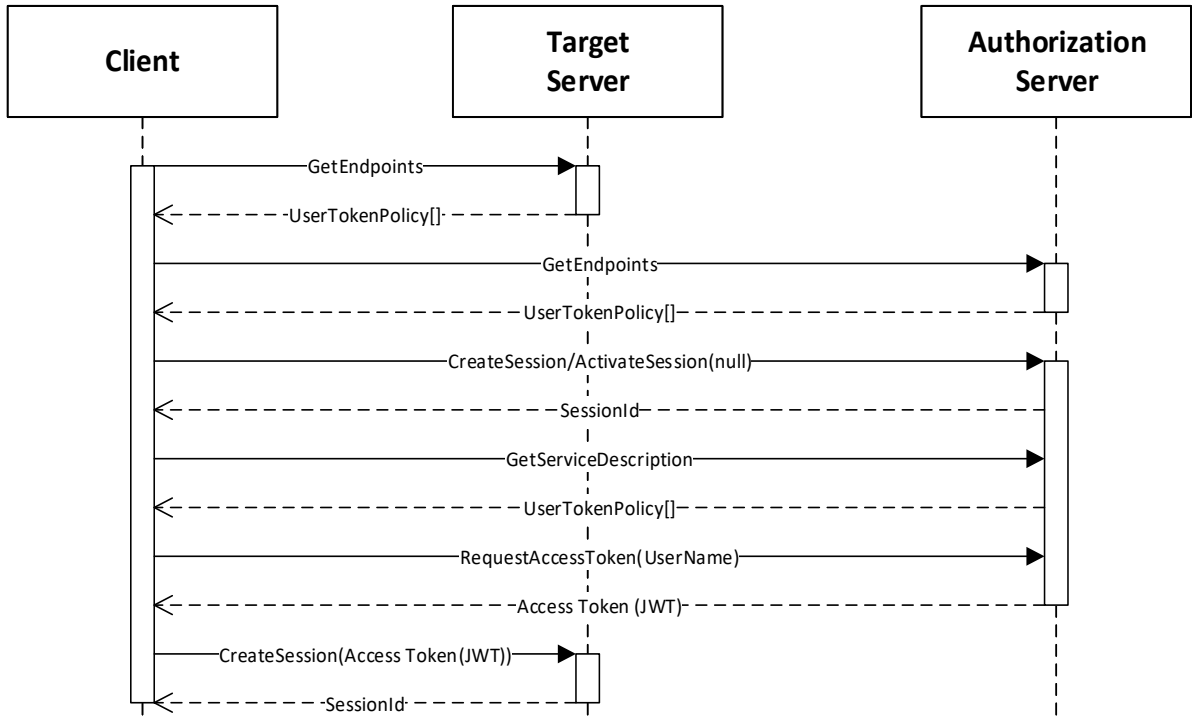

Figure 29 – Explicit Authorization

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 9.3.

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 9.3.

9.5 Chained

The chained use case means the *Client* provides an *Access Token* issued by another *AuthorizationService* acting as an *Identity Provider*. This use case is illustrated in Figure 30.

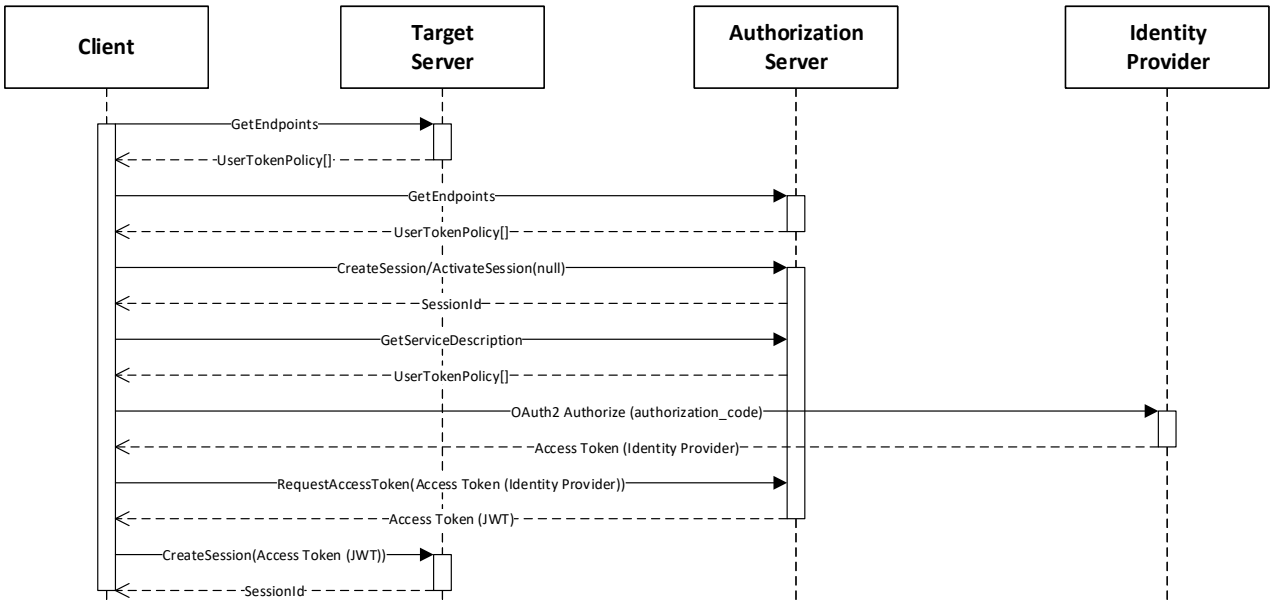

Figure 30 – Chained Authorization

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 9.3.

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 9.3.

9.6 Information Model for Requesting Access Tokens

9.6.1 Overview

The information model for *AuthorizationServices* which allow *Clients* to request *Access Tokens* from a *Server* is shown in Figure 31.

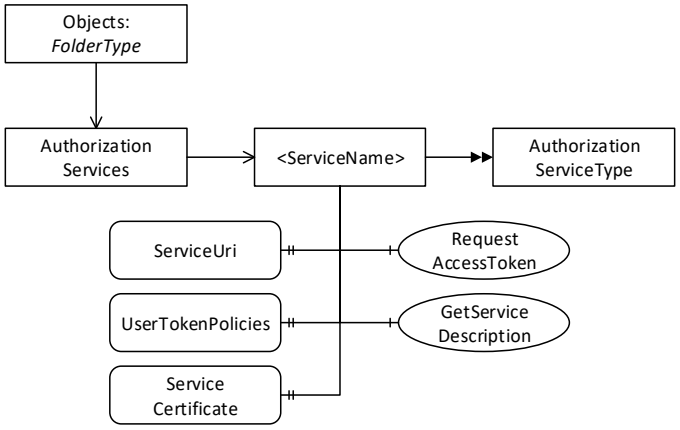

Figure 31 – The Model for Requesting Access Tokens from AuthorizationServices

9.6.2 AuthorizationServicesFolderType

This *ObjectType* represents a folder that contains *AuthorizationService Objects* which may be accessed via the *Server*. It is defined in Table 104.

Table 104 – AuthorizationServicesFolderType Definition

Attribute	**Value**			
BrowseName	2:AuthorizationServicesFolderType			
IsAbstract	False			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FolderType* defined in OPC 10000-5.				
0:Organizes	Object	2:<ServiceName>	2:AuthorizationServiceType	OptionalPlaceholder
Conformance Units				
GDS Authorization Service Server				

9.6.3 AuthorizationServices

This *Object* is an instance of *AuthorizationServicesFolderType* 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 OPC 10000-5. It is defined in Table 105.

Table 105 – AuthorizationServices Object Definition

Attribute	**Value**			
BrowseName	2:AuthorizationServices			
TypeDefinition	*2:AuthorizationServicesFolderType* defined in 9.6.2.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Conformance Units				
GDS Authorization Service Server				

9.6.4 AuthorizationServiceType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows access to an *AuthorizationService*. It is defined in Table 106.

Table 106 – AuthorizationServiceType Definition

Attribute	**Value**				
BrowseName	2:AuthorizationServiceType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	2:ServiceUri	0:String	0:PropertyType	Mandatory
0:HasProperty	Variable	2:ServiceCertificate	0:ByteString	0:PropertyType	Mandatory

0:HasProperty	Variable	2:UserTokenPolicies	0:UserToken Policy []	0:PropertyType	Optional
0:HasComponent	Method	2:GetServiceDescription	Defined in 9.6.6.		Mandatory
0:HasComponent	Method	2:RequestAccessToken	Defined in 9.6.5.		Optional
Conformance Units					
GDS Authorization Service Server					

The *ServiceUri* is a globally unique identifier that allows a *Client* to correlate an instance of *AuthorizationServiceType* with instances of *AuthorizationServiceConfigurationType* (see 9.7.4).

The *ServiceCertificate* is the *Certificate* required to check any *Signature* that is included with the *Access Tokens.* The *ServiceCertificate* may be a complete chain (see OPC 10000-6 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 *AuthorizationService.*

9.6.5 RequestAccessToken

RequestAccessToken is used to request an *Access Token* from an *AuthorizationService*. The scenarios where this *Method* is used are described fully in 9.3, 9.4 and 9.5.

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 OPC 10000-4.

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 v3 IdentityTokens* 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* shall be called from an encrypted *SecureChannel* and from a *Client* that has access to the *AccessTokenRequestor Privilege* (see 9.2).

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.
Bad_SecurityModeInsufficient	The SecureChannel is not encrypted.

Table 107 specifies the *AddressSpace* representation for the *RequestAccessToken Method*.

Table 107 – RequestAccessToken Method AddressSpace Definition

Attribute	Value				
BrowseName	2:RequestAccessToken				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:InputArguments	0:Argument[]	0:PropertyType	Mandatory
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

9.6.6 GetServiceDescription

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

Signature

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

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

Method Result Codes (defined in Call Service)

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

Table 108 specifies the *AddressSpace* representation for the *GetServiceDescription Method*.

Table 108 – GetServiceDescription Method AddressSpace Definition

Attribute	Value				
BrowseName	2:GetServiceDescription				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
0:HasProperty	Variable	0:OutputArguments	0:Argument[]	0:PropertyType	Mandatory

9.6.7 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 109.

Table 109 – AccessTokenIssuedAuditEventType Definition

Attribute	Value				
BrowseName	2:AccessTokenIssuedAuditEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *0:AuditUpdateMethodEventType* defined in OPC 10000-5.					
Conformance Units					
GDS Authorization Service Server					

This *EventType* inherits all *Properties* of the *AuditUpdateMethodEventType*. Their semantic is defined in OPC 10000-5.

9.7 Information Model for Configuring Servers

9.7.1 Overview

The information model used to provide *Servers* with the information needed to accept *Access Tokens* from *AuthorizationServices* in Figure 32.

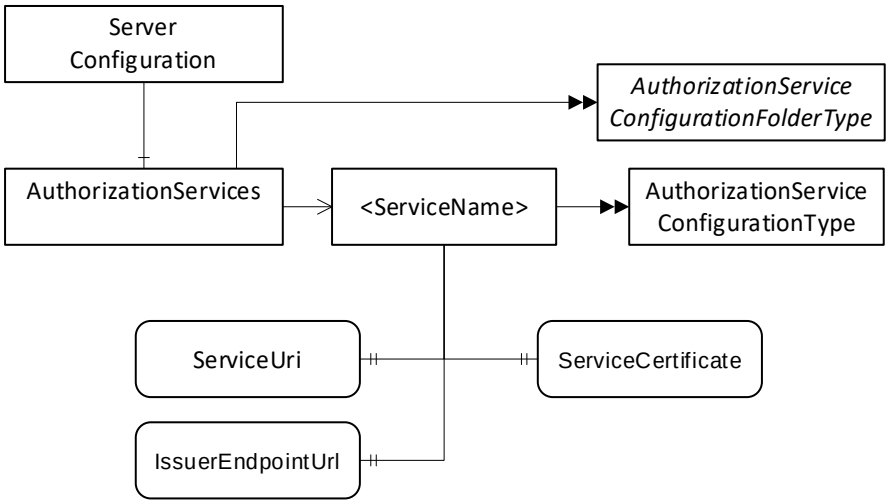

Figure 32 – The Model for Configuring Servers to use AuthorizationServices

If a *Server* is also a *Client* that needs to access the *AuthorizationService,* the necessary *KeyCredentials* can be provided with the push configuration management model (see 8.4).

9.7.2 AuthorizationServiceConfigurationFolderType

This *ObjectType* represents a folder that contains *AuthorizationServiceConfiguration Objects* which may be accessed via the *Server*. It is defined in Table 110.

Table 110 – AuthorizationServicesFolderType Definition

Attribute	Value			
BrowseName	0:AuthorizationServicesConfigurationFolderType			
IsAbstract	False			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Subtype of the *0:FolderType* defined in OPC 10000-5.				
0:HasComponent	Object	0:<ServiceName>	0:AuthorizationServiceConfigurationType	OptionalPlaceholder
Conformance Units				
Authorization Service Configuration Server				

9.7.3 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 7.10.3. It is defined in Table 111.

Table 111 – AuthorizationServices Object Definition

Attribute	**Value**			
BrowseName	0:AuthorizationServices			
TypeDefinition	*0:AuthorizationServicesConfigurationFolderType* defined in 9.6.2.			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**Modelling Rule**
Conformance Units				
Authorization Service Configuration Server				

9.7.4 AuthorizationServiceConfigurationType

This *ObjectType* is the *TypeDefinition* for an *Object* that allows the configuration of an *AuthorizationService* used by a *Server*. It is defined in Table 112.

Table 112 – AuthorizationServiceConfigurationType Definition

Attribute	**Value**				
BrowseName	0:AuthorizationServiceConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *0:BaseObjectType* defined in OPC 10000-5.					
0:HasProperty	Variable	0:ServiceUri	0:String	0:PropertyType	Mandatory
0:HasProperty	Variable	0:ServiceCertificate	0:ByteString	0:PropertyType	Mandatory
0:HasProperty	Variable	0:IssuerEndpointUrl	0:String	0:PropertyType	Mandatory
Conformance Units					
Authorization Service Configuration Server					

The *ServiceUri Property* uniquely identifies the *AuthorizationService*.

The *ServiceCertificate Property* has the *Certificate(s)* needed to verify *Access Tokens* issued by the *AuthorizationService*. The value is the complete chain of Certificate needed for verification (see OPC 10000-6 for information on encoding chains).

The *IssuerEndpointUrl* is the value of the *IssuerEndpointUrl* in *UserTokenPolicies* which require the use of the AuthorizationService. This contents of the field depend on the AuthorizationService and are described in OPC 10000-6.

10 Namespaces

10.1 Namespace Metadata

Table 113 defines the namespace metadata for this document. The *Object* is used to provide version information for the namespace and an indication about static *Nodes*. Static *Nodes* are identical for all *Attributes* in all *Servers*, including the *Value Attribute*. See OPC 10000-5 for more details.

The information is provided as *Object* of type *NamespaceMetadataType*. This *Object* is a component of the *Namespaces Object* that is part of the *Server Object*. The *NamespaceMetadataType ObjectType* and its *Properties* are defined in OPC 10000-5.

The version information is also provided as part of the ModelTableEntry in the UANodeSet XML file. The UANodeSet XML schema is defined in OPC 10000-6.

Table 113 – NamespaceMetadata Object for this Document

Attribute	Value	
BrowseName	2:http://opcfoundation.org/UA/GDS/	
Property	**DataType**	**Value**
0:NamespaceUri	0:String	http://opcfoundation.org/UA/GDS/
0:NamespaceVersion	0:String	1.05.04
0:NamespacePublicationDate	0:DateTime	2024-07-01
0:IsNamespaceSubset	0:Boolean	False
0:StaticNodeIdTypes	0:IdType []	0
0:StaticNumericNodeIdRange	0:NumericRange []	
0:StaticStringNodeIdPattern	0:String	

10.2 Handling of OPC UA Namespaces

Namespaces are used by OPC UA to create unique identifiers across different naming authorities. The *Attributes NodeId* and *BrowseName* are identifiers. A *Node* in the UA *AddressSpace* is unambiguously identified using a *NodeId*. Unlike *NodeIds*, the *BrowseName* cannot be used to unambiguously identify a *Node*. Different *Nodes* may have the same *BrowseName*. They are used to build a browse path between two *Nodes* or to define a standard *Property*.

Servers may often choose to use the same namespace for the *NodeId* and the *BrowseName*. However, if they want to provide a standard *Property*, its *BrowseName* shall have the namespace of the standards body although the namespace of the *NodeId* reflects something else, for example the *EngineeringUnits Property*. All *NodeIds* of *Nodes* not defined in this document shall not use the standard namespaces.

Table 114 provides a list of namespaces and their index used for *BrowseNames* in this document. The default namespace of this document is not listed since all *BrowseNames* without prefix use this default namespace.

Table 114 – Namespaces used in this document

NamespaceURI	Namespace Index	Example
http://opcfoundation.org/UA/	0	0:EngineeringUnits
http://opcfoundation.org/UA/GDS/	2	2:ApplicationRecordDataType

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 33.

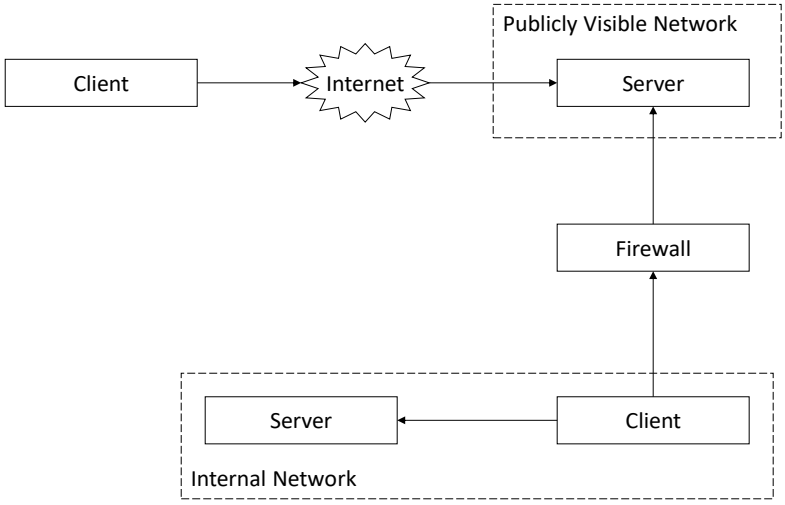

Figure 33 – Discovering Servers Outside a Firewall

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 *DiscoveryServers* 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 34.

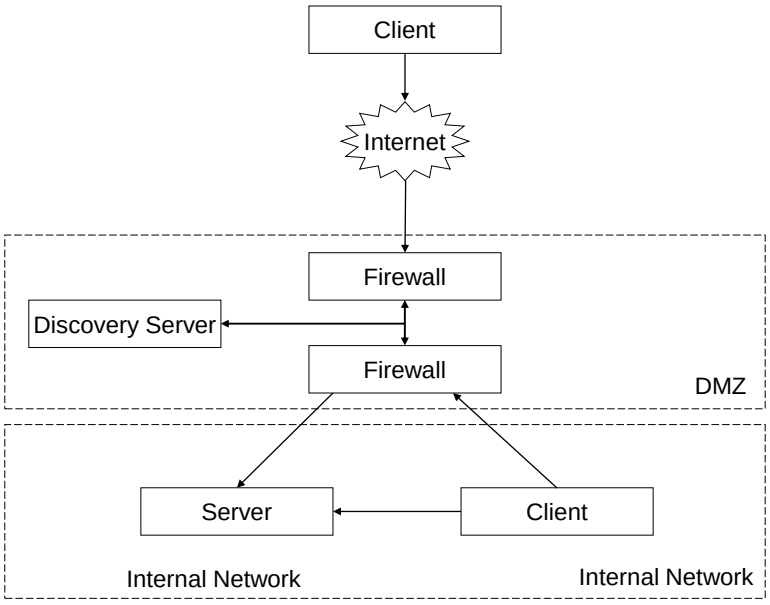

Figure 34 – Discovering Servers Behind a Firewall

In this configuration the address of the *Server* that the Internet *Client* sees will be different from the address that the internal *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 OPC 10000-4.

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 35. At this point the *Client* would have all the information it needs to establish a secure channel with the *Server* behind the firewall.

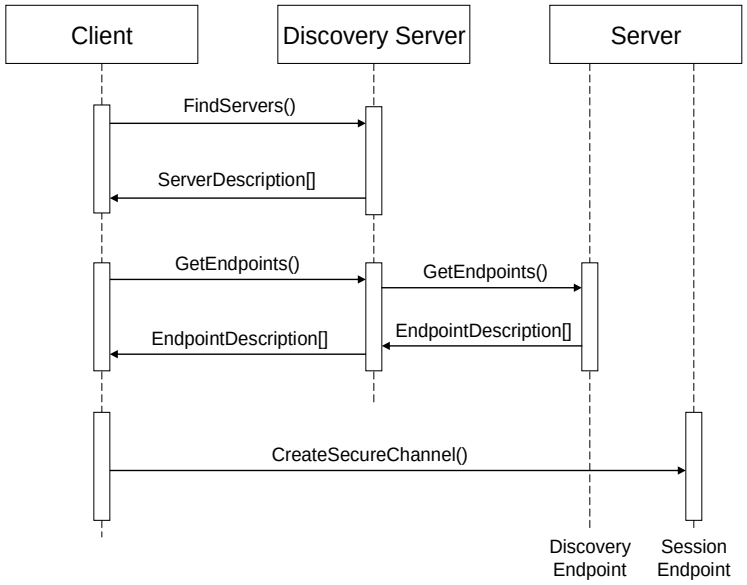

Figure 35 – Using a Discovery Server with a Firewall

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 36.

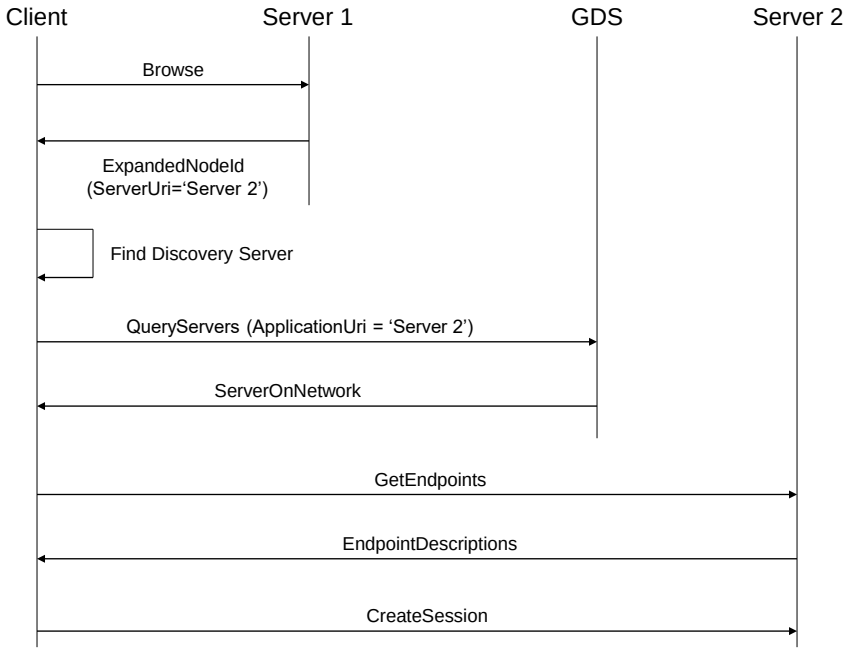

Figure 36 – Following References to Remote Servers

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

Annex B
(normative)

NodeSet and Constants

B.1 NodeSet

The OPC UA NodeSet includes the complete Information Model defined in this document. It follows the XML Information Model schema syntax defined in OPC 10000-6 and can thus be read and processed by a computer program.

The complete Information Model Schema for this version of this document (including any amendments and errata) can be found here:

http://www.opcfoundation.org/UA/schemas/1.05/Opc.Ua.Gds.NodeSet2.xml

NOTE The latest Information Model schema that is compatible with this version of this document can be found here:
http://www.opcfoundation.org/UA/schemas/Opc.Ua.Gds.NodeSet2.xml

The complete Information Model Schema includes many types which are only used in *Service Requests* and *Responses* and should not be used by *Servers* to populate their *Address Space*.

B.2 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 OPC 10000-6.

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/1.05/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 C describes 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 115.

Table 115 – Allowed mDNS Service Names

Service Name	Description
_opcua-tcp	The *DiscoveryUrl* supports the OPC UA TCP mapping (see OPC 10000-6). This name is assigned by IANA.
_opcua-tls	The *DiscoveryUrl* supports the OPC UA WebSockets mapping (see OPC 10000-6). 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 specification adopts the key-value syntax for TXT records described in DNS-SD.

Table 116 defines the syntax for strings that may in the TXT record.

Table 116 – 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 117 – 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 116).

Suitable default values should be chosen for fields in a SRV record that do not have a mapping specified in Table 117. 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.

CapabilityIdentifiers 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 118 is a non-normative list of possible identifiers.

Table 118 – Examples of CapabilityIdentifiers

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	The *ApplicationType* is *DiscoveryServer*. Only used by a standalone LDS implementation.
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	Accepts reverse connect requests as defined in OPC 10000-6.
PUB	Supports the *Publisher* capabilities defined in OPC 10000-14.
PSC	Supports the *PubSub Configuration* model defined in OPC 10000-14.
ALIAS	Supports the *Alias Names* capabilities defined in OPC 10000-17.
SKS	Supports the Security Key Server (SKS) capabilities defined in OPC 10000-14.
REGISTRAR	Supports the Registrar model defined in OPC 10000-21.
DCA	Supports the Device Configuration Application (DCA) model defined in OPC 10000-21.

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

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

This CSV will be changed to meet the needs of companion specifications and will not trigger an update to this document. Application developers shall always use the linked CSV.

Applications that support the PUB capability can send *PubSub Messages* but may 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 *DirectoryServices* 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 *QueryApplications* method will return the same information as the *DirectoryService* API. The discovery process for this scenario is illustrated in Figure 37.

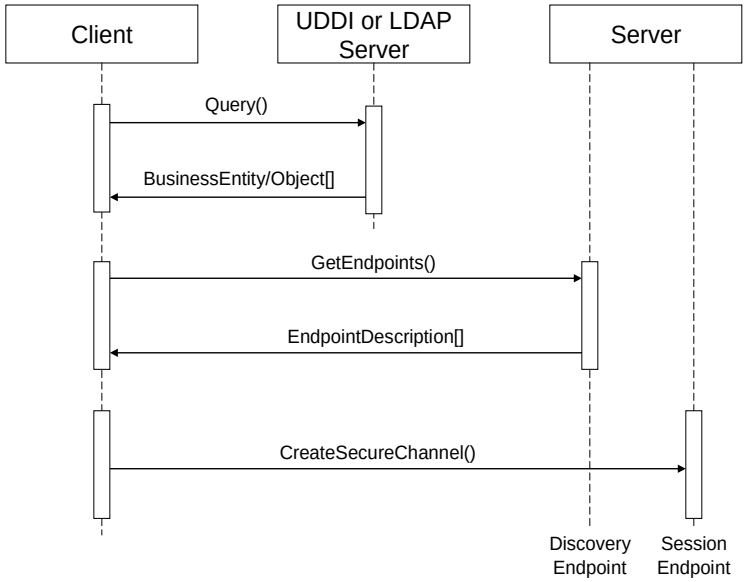

Figure 37 – The UDDI or LDAP Discovery Process

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 38 illustrates the relationships between the UDDI registry elements.

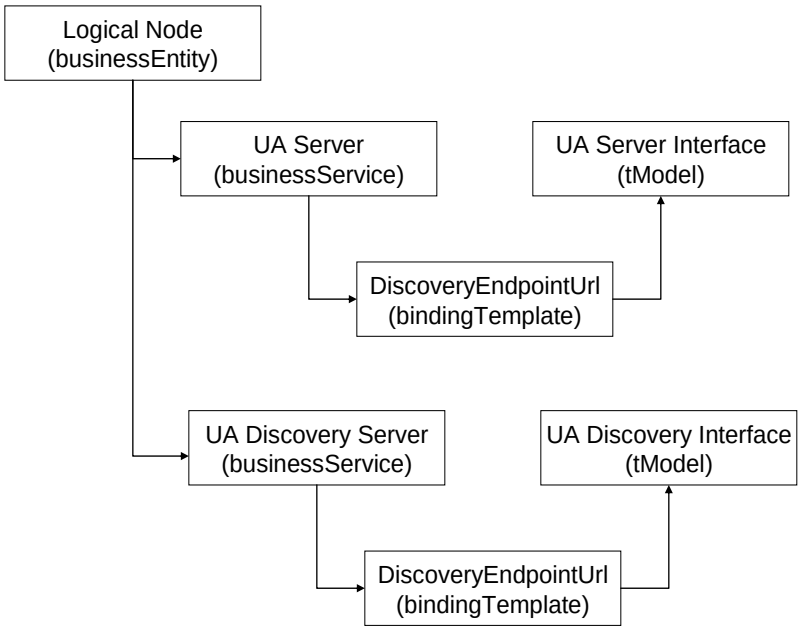

Figure 38 – UDDI Registry Structure

This specification defines standard tModels which shall be referenced by businessServices that support UA. The standard UA tModels shown in Table 119.

Table 119 – 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 39 illustrates a sample LDAP hierarchy which contains entries describing UA servers.

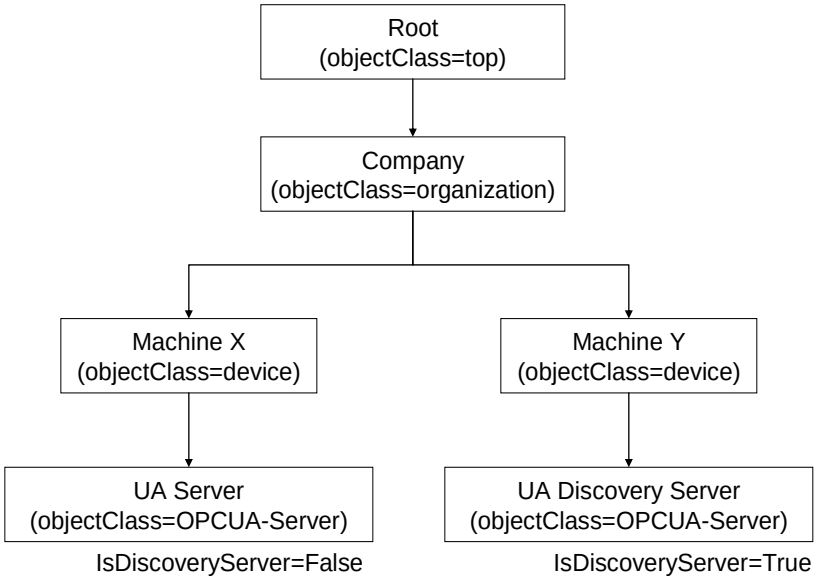

Figure 39 – Sample LDAP Hierarchy

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 120.

Table 120 – 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 121. The Local Discovery Server shall use this structure.

This structure is based on the rules defined in OPC 10000-6.

Table 121 – Application Certificate Store Directory Layout

Path	Description
<root>	A descriptive name for the TrustList.
<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>-[<Algorithm>-<Thumbprint>].(der | pem | pfx)

Where the CommonName is the CommonName of the Certificate, the Algorithm is the key-pair algorithm and the Thumbprint is the *CertificateDigest* of the certificate formatted as a hexadecimal string.

The currently supported key-pair algorithms are: RSA, nistP256, nistP384, brainpoolP256r1, brainpoolP384r1, curve25519 and curve448.

F.2 Installation Directories on Windows

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

```
%CommonProgramFiles%\OPC Foundation\UA\Discovery
```

where %CommonProgramFiles% is the value of the *CommonProgramFiles* environment variable on 32-bit systems. On 64-bit systems the value of the *CommonProgramFiles(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 Setup

G.1 Application Setup with PullManagement

Applications that use *PullManagement* (see 7.3) to setup their configuration need to know the location of the *CertificateManager* which they can use to request *Certificates* and download *TrustLists*. 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 *SecureChannel*. The Application may choose to connect even if it has not been pre-configured to trust the CertificateManager, however, Applications should not provide any secret information to a *CertificateManager* that is not trusted.

After establishing a *SecureChannel* with the *CertificateManager*, the *Application* needs demonstrate that it has permission to request *Certificates* and *TrustLists*. This permission may be granted if the *CertificateManager* is pre-configured with CAs and/or *Certificates* used by *Applications* on the network (see OPC 10000-21).

Permissions can also be granted if the *Application* provides user credentials that give it *ApplicationAdmin* rights (see 7.2). If the *CertificateManager* is not pre-configured to be trusted by the *Application* then the *Application* shall not provide any secrets, such as passwords, to the *CertificateManager*. It may use *UserIdentityTokens*, such as *X509IdentityTokens*, that do not require a secret to be sent to a potentially malicious *CertificateManager*.

If an *Application* prompts the user to enter the credentials to use it shall not persist these credentials for use in the future.

A *CertificateManager* may accept a *CertificateRequest* from unknown *Applications* that provide anonymous credentials if there is a process for manual review by a *CertificateManager* administrator. The *Certificate* is not issued until the *CertificateRequest* is approved.

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 *TrustList*.

G.2 Application setup with the PushManagement

Servers that use *PushManagement* (see 7.4) to initialize their configuration shall have a default *Certificate* assigned before the *PushManagement* process can start.

In addition, *Servers* shall go into an application setup state (for example, see OPC 10000-21) that makes it possible for remote *Clients* to update the security configuration via the *ServerConfiguration Object* (see 7.10.3). When a *Server* is in the application setup state it shall limit the available functionality. The value of the *ServerState Property* shall be *NoConfiguration*.

It is good practice for a *Client* to always check the *ServerState* after creating a *Session*. If the *ServerState* is *NoConfiguration* then the *Client* should check the *InApplicationSetup Property* on the *ServerConfiguration Object* to confirm that the *Server* is in the application setup state.

In some cases, cached user credentials will no longer work because of *Server* has been reset. *Clients* can determine that the Server is in the Application Setup state by reconnecting using Anonymous user credentials and checking the *ServerState Property*.

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

A possible workflow for implementing the Application Setup state is:

1. A flag in the configuration file that defaults to ON;

2. Always allow *Clients* to connect securely and assign the *SecurityAdmin Role* to *Anonymous* user if the *TrustList* is empty;

3. Connect to the *Server* after toggling a physical switch on the device which enables access for a short period.

4. Add *Client ApplicationUri* to *SecurityAdmin Role*, remove Anonymous from *SecurityAdmin Role*;

5. Provide a new *Certificate* and *TrustList*;

6. Set the configuration flag to OFF.

Subsequent updates to *TrustLists* or *Certificates* can be allowed if the *Client* has a trusted *Certificate* and has access to the *SecurityAdmin Role*. During the setup state the *Client* shall configure the *SecurityAdmin Role*. If the *Client* fails to do this Server shall stay in application setup state.

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 *TrustList* 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 onboarding of the device. OPC 10000-21 formally defines mechanisms for onboarding new devices when they are connected to the network.

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 *TrustList*. This can also be done by setting operating system permissions on the directory where the *TrustList* 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 their *TrustList* 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 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 *TrustList* assigned to the application from the *CertificateManager*. Prior to these operations the *Client* should verify that the server is authorized to provide CAs. Table 122 compares how EST clients verify the EST server with how *CertificateManager* clients verify a *CertificateManager*.

Table 122 – 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 *TrustList*.
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 123 compares how EST servers verify the EST client with how a *CertificateManager* verifies a *CertificateManager* client.

Table 123 – 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 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 *AddressSpace*. 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.

_____