OPC UA Client SDK .NET Standard

Develop OPC UA Clients with C# / VB.NET

Tutorial Workshop Client

Document Control

Version	Date	Comment
1.0.8	18-MAY-2019	Initial version based on SDK 1.0.8
1.0.9	31-MAY-2019	Removed several empty pages

Purpose and audience of document

Microsoft's .NET Framework is an application development environment that supports multiple languages and provides a large set of standard programming APIs. This document defines an Application Programming Interface (API) for OPC UA Client and Server development based on the .NET Standard programming model.

The OPC UA specification can be downloaded from the web site of the OPC Foundation. But only [UA Part 1] (Overview and Concepts) is available to the public. All other parts can only be downloaded from OPC Foundation members and may be used only if the user is an active OPC Foundation member. Because of this fact the OPC UA SDK .NET Standard API hides most of the OPC UA specifications to provide the possibility to delevlop OPC UA Clients and OPC UA Servers in the .NET Standard environment without the need to be an OPC Foundation member. The API does support OPC Unified Architecture.

This document is intended as reference material for developers of OPC UA compliant Client and Server applications. It is assumed that the reader is familiar with the Microsoft's .NET Standard and the needs of the Process Control industry.

Summary

This document gives a short overview of the functionality of the client development with the OPC UA Client SDK .NET Standard. The goal of this document is to give an introduction and can be used as base for your own implementations

Referenced OPC Documents

Documents
This document partly uses extracts taken from the OPC UA specifications to be able to give at least a short introduction into the specifications. The specifications itself are available from: http://www.opcfoundation.org/Default.aspx/01_about/UA.asp?MID=AboutOPC#Specifications
OPC Unified Architecture Textbook, written by Wolfgang Mahnke, Stefan-Helmut Leitner and Matthias Damm: http://www.amazon.com/OPC-Unified-Architecture-Wolfgang-Mahnke/dp/3540688986/ref=sr_1_1?ie=UTF8&s=books&qid=1209506074&sr=8-1
[UA Part 1] OPC UA Specification: Part 1 – Concepts http://www.opcfoundation.org/UA/Part1/
[UA Part 2] OPC UA Specification: Part 2 – Security http://www.opcfoundation.org/UA/Part2/
[UA Part 3] OPC UA Specification: Part 3 – Address Space Model http://www.opcfoundation.org/UA/Part3/
[UA Part 4] OPC UA Specification: Part 4 – Services http://www.opcfoundation.org/UA/Part4/
[UA Part 5] OPC UA Specification: Part 5 – Information Model http://www.opcfoundation.org/UA/Part5/
[UA Part 6] OPC UA Specification: Part 6 – Mappings http://www.opcfoundation.org/UA/Part6/
[UA Part 7] OPC UA Specification: Part 7 – Profiles http://www.opcfoundation.org/UA/Part7/
[UA Part 8] OPC UA Specification: Part 8 – Data Access http://www.opcfoundation.org/UA/Part8/
[UA Part 9] OPC UA Specification: Part 9 – Alarm & Conditions http://www.opcfoundation.org/UA/Part9/
[UA Part 10] OPC UA Specification: Part 10 – Programs http://www.opcfoundation.org/UA/Part10/
[UA Part 11] OPC UA Specification: Part 11 – Historical Access http://www.opcfoundation.org/UA/Part11/
[UA Part 12] OPC UA Specification: Part 12 – Discovery and Global Services http://www.opcfoundation.org/UA/Part12/
[UA Part 13] OPC UA Specification: Part 13 – Aggregates http://www.opcfoundation.org/UA/Part13/
[UA Part 14] OPC UA Specification: Part 14 – PubSub https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-14-pubsub/

Other Referenced Documents

SOAP Part 1: SOAP Version 1.2 Part 1: Messaging Framework

http://www.w3.org/TR/soap12-part1/

SOAP Part 2: SOAP Version 1.2 Part 2: Adjuncts

http://www.w3.org/TR/soap12-part2/

XML Encryption: XML Encryption Syntax and Processing

http://www.w3.org/TR/xmlenc-core/

XML Signature:: XML-Signature Syntax and Processing

http://www.w3.org/TR/xmldsig-core/

WS Security: SOAP Message Security 1.1

http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

WS Addressing: Web Services Addressing (WS-Addressing)

http://www.w3.org/Submission/ws-addressing/

WS Trust: Web Services Trust Language (WS-Trust)

http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf

WS Secure Conversation: Web Services Secure Conversation Language (WS-SecureConversation)

http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf

SSL/TLS: RFC 2246: The TLS Protocol Version 1.0

http://www.ietf.org/rfc/rfc2246.txt

X200 : ITU-T X.200 – Open Systems Interconnection – Basic Reference Model

http://www.itu.int/rec/T-REC-X.200-199407-I/en

:X509: X.509 Public Key Certificate Infrastructure

http://www.itu.int/rec/T-REC-X.509-200003-I/e

HTTP: RFC 2616: Hypertext Transfer Protocol - HTTP/1.1

http://www.ietf.org/rfc/rfc2616.txt

HTTPS: RFC 2818: HTTP Over TLS

http://www.ietf.org/rfc/rfc2818.txt

IS Glossary: Internet Security Glossary

http://www.ietf.org/rfc/rfc2828.txt

NIST 800-12: Introduction to Computer Security

http://csrc.nist.gov/publications/nistpubs/800-12/

NIST 800-57: Part 3: Application-Specific Key Management Guidance

http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_PART3_key-management_Dec2009.pdf

NERC CIP: CIP 002-1 through CIP 009-1, by North-American Electric Reliability Council

http://www.nerc.com/page.php?cid=2|20

IEC 62351: Data and Communications Security

http://www.iec.ch/heb/d_mdoc-e050507.htm

SPP-ICS: System Protection Profile
Industrial Control System, by Process Control Security Requirements Forum (PCSRF)

http://www.isd.mel.nist.gov/projects/processcontrol/SPP-ICSv1.0.pdf

SHA-1: Secure Hash Algorithm RFC

http://tools.ietf.org/html/rfc3174

PKI: Public Key Infrastructure article in Wikipedia

http://en.wikipedia.org/wiki/Public_key_infrastructure

X509 PKI: Internet X.509 Public Key Infrastructure

http://www.ietf.org/rfc/rfc3280.txt

EEMUA : 2nd Edition EEMUA 191 - Alarm System - A guide to design, management and procurement (Appendixes 6, 7, 8, 9).

http://www.eemua.co.uk/

Table of Contents

Disclaimer

This document was created strictly for information purposes. No guarantee, contractual specification or condition shall be derived from this document unless agreed to in writing. Technosoftware GmbH reserves the right to make changes in the products and services described in this document at any time without notice and this document does not represent a commitment on the part of Technosoftware GmbH in the future.

While Technosoftware GmbH uses reasonable efforts to ensure that the information and materials contained in this document are current and accurate, Technosoftware GmbH makes no representations or warranties as to the accuracy, reliability or completeness of the information, text, graphics, or other items contained in the document. Technosoftware GmbH expressly disclaims liability for any errors or omissions in the materials contained in the document and would welcome feedback as to any possible errors or inaccuracies contained herein.

Technosoftware GmbH shall not be liable for any special, indirect, incidental, or consequential damages, including without limitation, lost revenues or lost profits, which may result from the use of these materials. All offers are non-binding and without obligation unless agreed to in writing.

Trademark Notice

Microsoft, MSN, Windows and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

1 Installation

You should download the following products from either https://technosoftware.com/ or the license paper provided to be able to follow this tutorial:

1. **OPC Sample Binaries .NET Standard**
   A suite of OPC Clients and Servers that demonstrate OPC UA, OPC DA, OPC AE and OPC HDA technology and its most popular functionality. All examples are ready to run without any configuration.
   You can donload it from https://technosoftware.com/download/opc-sample-binaries-net-standard/
2. **OPC UA Bundle SDK .NET Standard**
   The OPC UA Bundle SDK .NET Standard offers a fast and easy access to the OPC UA Client & Server technology. Develop OPC compliant UA Clients and Servers with C#/VB.NET targeting the .NET Standard.
   The download includes examples for .NET 4.6.1, .NET 4.7.2 and for .NET Standard 2.0.
   You can download it from https://technosoftware.com/download/opc-ua-bundle-sdk-net-standard-evaluation/

Important:

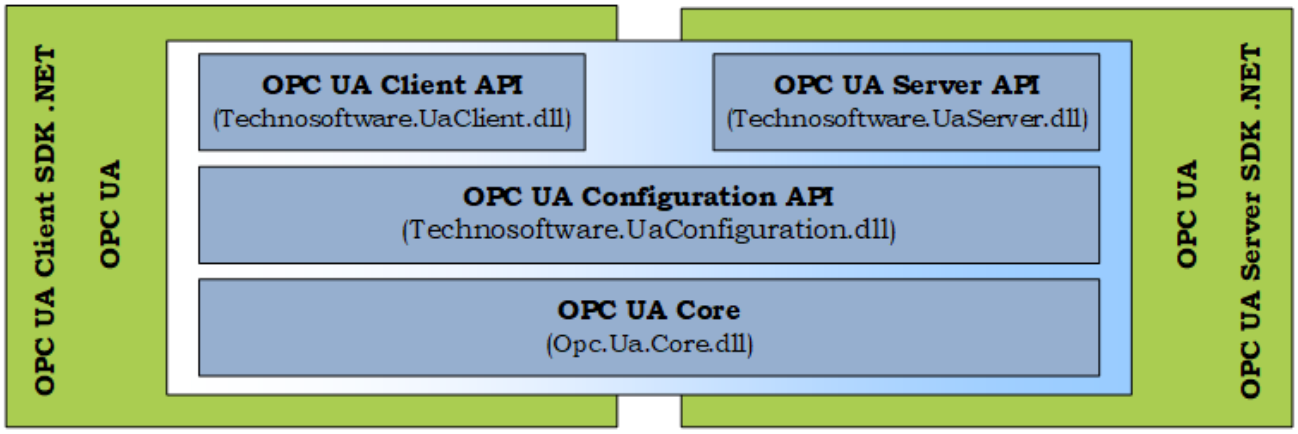
An installation guide is available with the SDK or from https://technosoftware.com/download/opc-ua-net-installation/. Please read that one first and then follow this guide.

2 Sample Applications

The OPC UA Client SDK .NET Standard contains several sample client applications, but we concentrate in this tutorial on the *WorkshopClientForms*, a console base application for testing the server specific features, and the *WorkshopClientConsole*. This tutorial will refer to that code while explaining the different steps to take to accomplish the main tasks of an OPC UA client.

2.1 Required SDK DLLs

The SDK is splitted into several DLL's as shown in the picture below:



The OPC UA Client SDK .NET Standard uses the following DLL's:

Name	Description
Opc.Ua.Core.dll	The OPC UA Stack. Based on the OPC Unified Architecture .NET Standard. We deliver three versions, one for .NET 4.6.1, one for .NET 4.7.2 and one for .NET Standard 2.0.
Technosoftware.UaConfiguration.dll	Contains configuration related classes like, e.g. `ApplicationInstance`. We deliver three versions, one for .NET 4.6.1, one for .NET 4.7.2 and one for .NET Standard 2.0.
Technosoftware.UaClient.dll	The DLL containing the classes and methods usable for OPC UA Client development. We deliver three versions, one for .NET 4.6.1, one for .NET 4.7.2 and one for .NET Standard 2.0.

2.2 Directory Structure

The basic directory layout is as follows:

- **bin/**
  - **net461/**
    Standard SDK Executables and DLL's for the .NET 4.6.1 Framework
  - **net472/**
    Standard SDK Executables and DLL's for the .NET 4.7.2 Framework
  - **netstandard2.0/**
    Standard SDK Executables and DLL's for the .NET Standard 2.0 and .NET Core 2.0 Framework
  - **redist/**
    - **OPC UA Local Discovery Server 1.03/**
      The installer and Merge-Module for the OPC UA Local Discovery Server
- **doc/**
  Additional documentation
- **examples/**
  Sample applications
  - **net/**
    Sample applications using .NET 4.6.1 as target framework
    - **C#/**
      - **AlarmConditions/**
        Client with Windows Forms UI showing the Alarm&Condition features
      - **CommonControls/**
        Windows Forms Controls used by the WorkshopClientSample. Contains the TitleBarControl which allows adapting logo and text of the title
      - **DataAccess/**
        Client with Windows Forms UI showing the Data Access features
      - **HistoricalAccess/**
        Client with Windows Forms UI showing the Historical Access features
      - **Reference/**
        Client used for testing with the OPC Foundation Complianc Test Tool
      - **ClientControls/**
        Windows Forms Controls used by the WorkshopClientForms
      - **Views/**
        Client with Windows Forms UI showing the Views features.
  - **Workshop/**
    - **ClientConsole/**
      Client as Console application used for this introduction. Features .NET 4.6.1, .NET 4.7.2 and .NET Core 2.0 compilation within one solution.
    - **ClientForms/**
      Client with Windows Forms used for this introduction
- **keys/**
  The dummy Key for signing the executables and DLL's
- **scripts/**
  Scripts and executables used for building the applications

2.3 OPC UA Client Solution

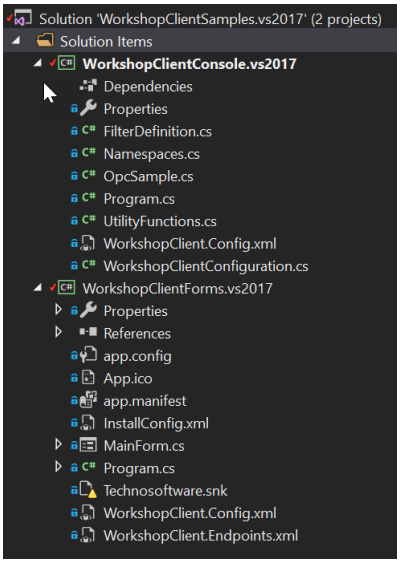
The main OPC UA Client Solutions can be found at \examples\Workshop\ and are named

- Visual Studio 2017:
  WorkshopClientSamples.vs2017.sln
- Visual Studio 2019:
  WorkshopClientSamples.vs2019.sln

and uses in addition the output of the following solutions:

1. Technosoftware.CommonControls
   Contains the ExceptionDialog and the TitleBarControl.
2. Technosoftware.ClientControls
   Contains the Client related controls.

The solution contains two sample clients, one a console-based client and one a Windows Forms based client.



2.3.1 WorkshopClientConsole Application

The main OPC UA Client Solution can be found at \examples\Workshop\ClientConsole.

This client application is a simple console application and is used to test the TechnosoftwareWorkshopServer. The main functionality used and shown here are:

1. Browsing existing UA servers
2. Connecting to the TechnosoftwareWorkshopServer
3. Browsing the address space of the TechnosoftwareWorkshopServer.
4. Reading/Writing of UA variables.
5. Reconnect handling

The documentation of this client can be found mainly in the code, but some of the basics are also explained in the following chapters.

2.3.2 WorkshopClientSample Windows Forms application

The main OPC UA Client Solution can be found at \examples\Workshop\ClientForms.

This client application is a Windows Forms based UI application and is based on the SampleClient delivered as binary with the SDK. Some adaptions where made so that the TechnosoftwareWorkshopServer is used per default. The main functionality used and shown here are:

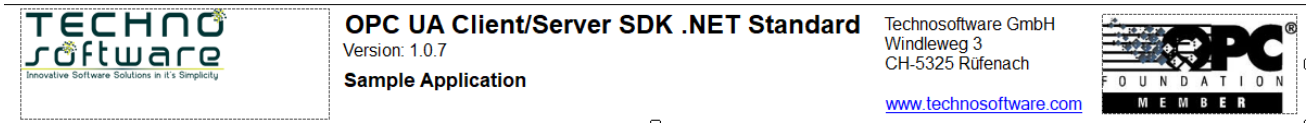
1. Browsing existing UA servers
2. Connecting to the TechnosoftwareWorkshopServer
3. Browsing the address space of the TechnosoftwareWorkshopServer.
4. Reconnect handling.

This client used several common controls provided within different solutions. A short overview of those controls is given in the following chapters.

2.3.3 CommonControls

2.3.3.1 Customizing the TitleBarControl

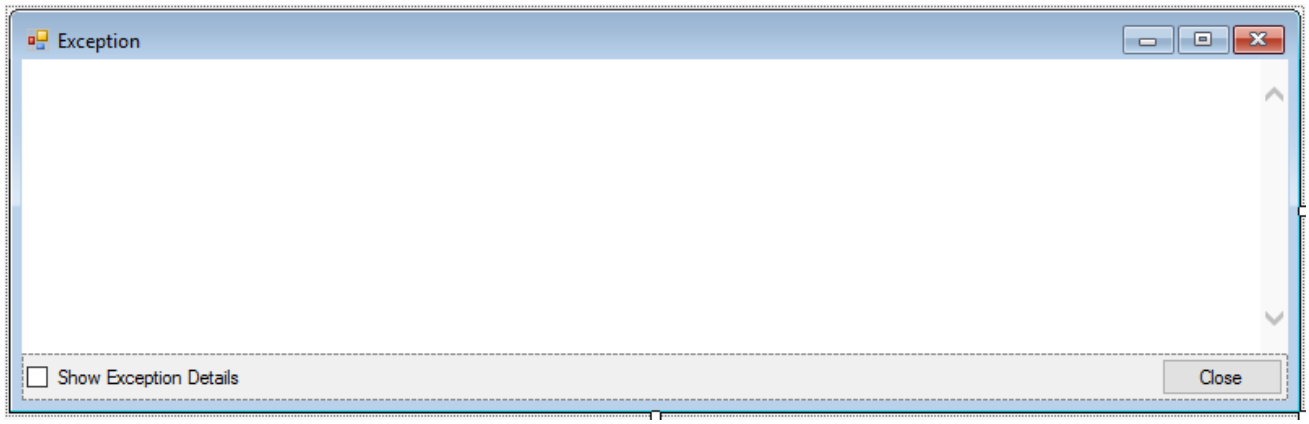
The TitleBarControl contains the header of all Windows Forms based sample server and sample client solutions provided with the SDK, e.g. WorkshopClientForms. The following picture shows how it looks like as default:



By changing this control, you can adapt the layout of the WorkshopClientSample to your needs.

2.3.3.2 Customizing the ExceptionDlg

The ExceptionDlg is used for displaying exceptions. The following picture shows how it looks like as default:



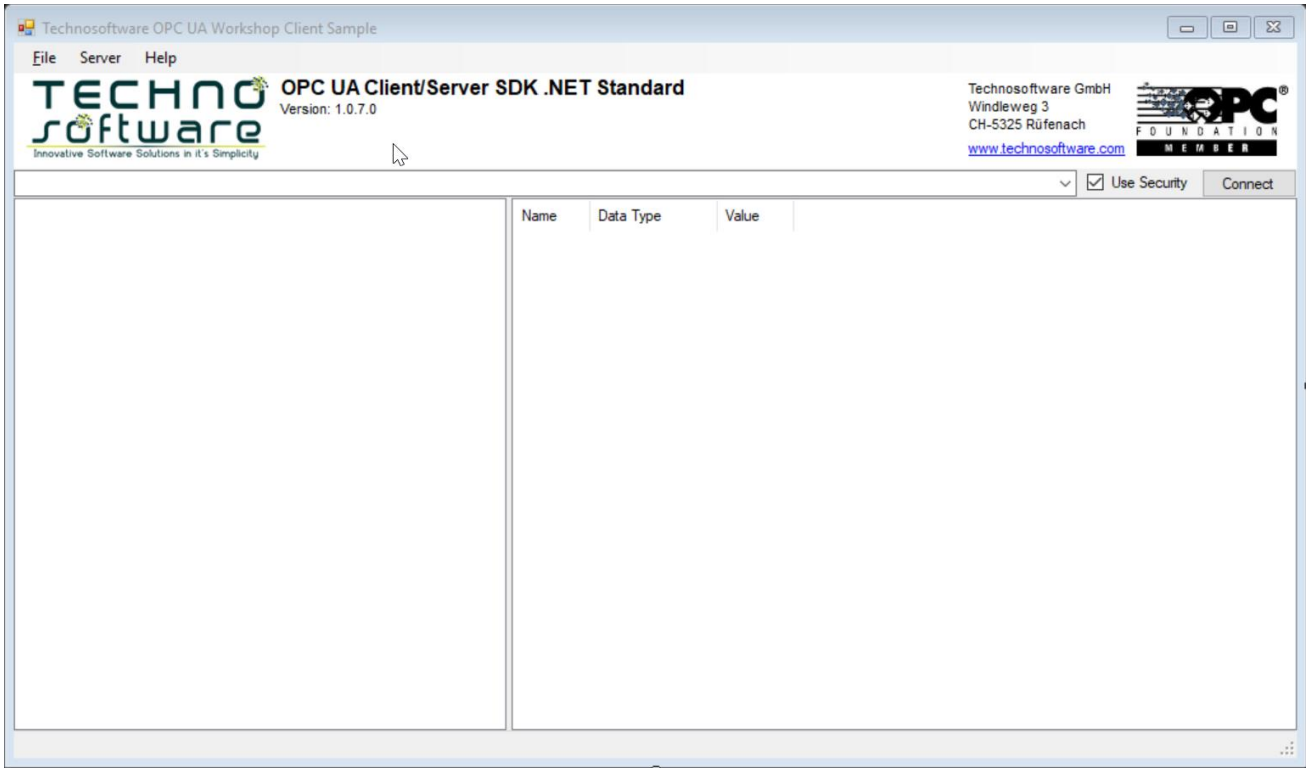
By changing this dialog, you can adapt the layout of the exception dialog for the WorkshopClientSample or the TechnosoftwareSampleServer to your needs.

2.3.4 ClientControls

The ClientControls contains several controls and dialogs used by the WorkshopClientForms.

2.3.4.1 Customizing the ClientForm

The ClientForm in the SampleControls solution combines the different controls to the main dialog used by the WorkshopClientSample. The following picture shows how it looks like as default:



By changing this dialog, you can adapt the layout of this form for the WorkshopClientForms to your needs.

3 Configuration

3.1 Application Configuration

The SDK provides an extensible mechanism for storing the application configuration in an XML file. The class is extensible, so developers can add their own configuration information to it. The table below describes primary elements of the ApplicationConfiguration class.

Name	Type	Description
ApplicationName	String	A human readable name for the application.
ApplicationUri	String	A globally unique name for the application. This should be a URL with which the machine domain name or IP address as the hostname followed by the vendor/product name followed by an instance identifier. For example: http://machine1/OPC/UASampleServer/4853DB1C-776D-4ADA-9188-00CAA737B780
ProductUri	String	A human readable name for the product.
ApplicationType	ApplicationType	The type of application. Possible values: Server_0, Client_1, ClientAndServer_2 or DiscoveryServer_3
SecurityConfiguration	SecurityConfiguration	The security configuration for the application. Specifies the application instance certificate, list of trusted peers and trusted certificate authorities.
TransportConfigurations	TransportConfiguration Collection	Specifies the Bindings to use for each transport protocol used by the application.
TransportQuotas	TransportQuotas	Specifies the default limits to use when initializing WCF channels and endpoints.
ServerConfiguration	ServerConfiguration	Specifies the configuration for Servers
ClientConfiguration	ClientConfiguration	Specifies the configuration for Clients
TraceConfiguration	TraceConfiguration	Specifies the location of the Trace file. Unexpected exceptions that are silently handled are written to the trace file. Developers can add their own trace output with the Utils.Trace(…) functions.
Extensions	XmlElementCollection	Allows developers to add additional information to the file.

The ApplicationConfiguration can be persisted anywhere but the class provides functions that load/save the configuration as an XML file on disk. The location of the XML file can be specified in the app.config file for the application if the ConfigurationLocation is specified as a configuration section.

The declaration for the configuration section in the app.config looks like this:

```
<configSections>
       <section name="WorkshopClient" type="Opc.Ua.ApplicationConfigurationSection,Opc.Ua.Core"/>
</configSections>
```

The name may be any text that is unique within the app.config file. The ConfigurationLocation would look like this:

```
<WorkshopClient>
	<ConfigurationLocation xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd">
		<FilePath>WorkshopClient.Config.xml</FilePath>
	</ConfigurationLocation>
</WorkshopClient>
```

The FilePath can be an absolute path or a relative path. If it is a relative path the current directory is searched followed by the directory where the executable resides. The SDK also supports prefixes which can be replaced with environment variables. The latter functionality requires a token enclosed by '%' symbols at the start of the message. The SDK will first check for a symbol that matches one of the values from the Environment.SpecialFolder enumeration. If not found it will use the environment variable of the same name.

Note that the same feature exists for all fields that represent file directory paths in the ApplicationConfiguration object.

The Application Configuration file of the WorkshopClientConsole can be found in the file WorkshopClientConsole.Config.xml.

3.1.1 Extensions

The Application Configuration file of the WorkshopClientConsole uses the Extensions feature to make the Excel Configuration configurable.

Name	Type	Description
ConfigurationFile	String	The full path including file name of the Excel file used for the configuration of the address space.

The Extension looks like:

```
<Extensions>
  <ua:XmlElement>
    <WorkshopClientConfiguration xmlns="http://technosoftware.com/WorkshopClient">
      <ConfigurationFile>.\WorkshopServerConfiguration.xlsx</ConfigurationFile>
    </WorkshopClientConfiguration>
  </ua:XmlElement>
</Extensions>
```

To get the configuration value the WorkshopClientConsole uses the following calls:

```
// get the configuration for the extension.
// In case no configuration exists use suitable defaults.
configuration_ =
   application.ApplicationConfig.ParseExtension<WorkshopClientConfiguration>() ??
                                                new WorkshopClientConfiguration();

string ConfigurationFile = configuration_.ConfigurationFile;
```

Important:
This only shows how to use the Extension feature. The Excel based configuration is not implemented at all.

3.1.2 Tracing Output

With the TraceConfiguration UA client and server applications can activate trace information. WorkshopClientConsole and WorkshopClientSample creates the following logfiles:

WorkshopClientConsole:
`%LocalApplicationData%/Logs/Technosoftware.WorkshopClient.log`

WorkshopClientForms:
`%CommonApplicationData%\Technosoftware\Logs\WorkshopClient.log`

where

%CommonApplicationData% typically points to C:\ProgramData

3.2 Installed Application

Important:
Wheras the OPC UA Client SDK .NET up to version 7.5.2 supported this feature, the OPC UA Client SDK .NET Standard does not support this.

4 Certificate Management and Validation

The stack provides several certificate management functions including a custom `CertificateValidator` that implements the validation rules required by the specification. The `CertificateValidator` is created automatically when the ApplicationConfiguration is loaded. Any WCF channels or endpoints that are created with that ApplicationConfiguration will use it.

The `CertificateValidator` uses the trust lists in the ApplicationConfiguration to determine whether a certificate is trusted. A certificate that fails validation is always placed in the Rejected Certificates store. Applications can receive notifications when an invalid certificate is encountered by using the event defined on the `CertificateValidator` class.

The Stack also provides the `CertificateIdentifier` class which can be used to specify the location of a certificate. The Find() method will look up the certificate based on the criteria specified (SubjectName, Thumbprint or DER Encoded Blob).

Each application has a SecurityConfiguration which must be managed carefully by the Administrator since making a mistake could prevent applications from communicating or create security risks. The elements of the SecurityConfiguration are described in the table below:

Name	**Description**
ApplicationCertificate	Specifies where the private key for the Application Instance Certificate is located. Private keys should be in the Personal (My) store for the LocalMachine or the CurrentUser. Private keys installed in the LocalMachine store are only accessible to users that have been explicitly granted permissions.
TrustedIssuerCertificates	Specifies the Certificate Authorities that issue certificates which the application can trust. The structure includes the location of a Certificate Store and a list of individual Certificates.
TrustedPeerCertificates	Specifies the certificates for other applications which the application can trust. The structure includes the location of a Certificate Store and a list of individual Certificates.
InvalidCertificateDirectory	Specifies where rejected Certificates can be placed for later review by the Admistrator (a.k.a. Rejected Certificates Store)

The Administrator needs to create an application instance certificate when applications are installed, when the ApplicationUri or when the hostname changes. The Administrator can use the OPC UA Configuration Tool included in the SDK or use the tools provided by their Public Key Infrastructure (PKI). If the certificate is changed the Application Configuration needs to be updated.

Once the certificate is installed the Administrator needs to ensure that all users who can access the application have permission to access the Certificate's private key.

5 UserIdentity and UserIdentityTokens

The SDK provides the UserIdentity class which converts UA user identity tokens to and from the SecurityTokens used by WCF. The SDK currently supports UserNameSecurityToken, X509SecurityToken, SamlSecurityToken and any other subtype of SecurityToken which is supported by the WCF WSSecurityTokenSerializer class. The UA specification requires that UserIdentityTokens be encrypted or signed before they are sent to the Server. UserIdentityToken class provides several methods that implement these features.

Important: This feature is not supported in the WorkshopClientConsole.

6 Client Startup

The ApplicationInstance class is the main instance used to use services provided by an OPC UA server. The implementation of an OPC UA client application startes with the creation of an ApplicationInstance object. These are the lines in the OpcSample.cs that create the ApplicationInstance:

```
ApplicationInstance application = new ApplicationInstance();
```

Important:
Wheras the OPC UA Client SDK .NET up to version 7.5.2 supported processing the command line, the OPC UA Client SDK .NET Standard does not support this. The ProcessCommandLine() method always returns false.

~~The ApplicationInstance class supports several standard command line parameters, e.g. for installing/uninstalling an application. Just call the ProcessCommandLine() Method to handle these command line parameters.~~

```
// process and command line arguments.
if (application.ProcessCommandLine())
{
    return;
}
```

~~With the above lines added to your code you can now use the following command line parameters to install/uninstall your application:~~

- ~~/install Install's the application and uses the parameters in InstallConfig.xml~~
- ~~/uninstall Uninstall's the application and uses the parameters in InstallConfig.xml~~

An OPC UA Client application can be configured via an application configuration file. This is handled by calling the LoadConfiguration() method. This loads the ApplicationConfiguration from the configuration file:

```
// Load the Application Configuration and use the specified config section
// "WorkshopClientConsole"
application.LoadConfiguration("WorkshopClientConsole");
```

The other possibility is to use the following lines:

```
application.ApplicationType   = ApplicationType.Client;
application.ConfigSectionName = "WorkshopClientSample";
// load the application configuration.
application.LoadApplicationConfigurationAsync(false).Wait();
```

The SDK supports transport security within the communication of server and clients. A client application can choose the security configuration to use, depending on the security implemented in the server. All clients must have an application certificate, which is used for the validation of trusted clients.

A connection can only be established if the security settings are correct. The security principals are those of a PKI (Public Key Infrastructure).

A client application can automatically accept certificates from an OPC UA Server. If this is not used a callback can be used to get informed about new server certificates. These are the lines in the OpcSample.cs that uses the certificate valkidator callback:

```
// Install a certificate validator callback.
if (!application.Configuration.SecurityConfiguration.AutoAcceptUntrustedCertificates)
{
    application.Configuration.CertificateValidator.CertificateValidation +=
                                                                OnCertificateValidation;
}


private void OnCertificateValidation(CertificateValidator validator,
                                     CertificateValidationEventArgs e)
{
    e.Accept = true;
}
```

The last step in using the ApplicationInstance object is to check the application instance certificate by calling the CheckCertificate() method.

```
// Check the Application Certificate.
application.CheckCertificate();
```

You can also use

```
// check the application certificate.
application.CheckApplicationInstanceCertificateAsync(false, 0).Wait();
```

6.1.1 Application Configuration Extensions

As described in chapter Extension [9.1] you can use the section Extension in the configuration file to add some configuration options. With the following code you load that extension and maps it to the `WorkshopClientConfiguration:`

```
// get the configuration for the extension.
// In case no configuration exists use suitable defaults.

configuration_ =
       application.ApplicationConfig.ParseExtension<WorkshopClientConfiguration>() ??
                                                    new WorkshopClientConfiguration();
```

You can now use those entries like shown below:

```
string ConfigurationFile = configuration_.ConfigurationFile;
```

7 Server Connection

To be able to connect to an OPC UA Server a server Url is required:

URI	Server
opc.tcp://<hostname>:55533/TechnosoftwareSampleServer	Technosoftware Sample Server
opc.tcp://localhost:55552/WorkshopSampleServer	Technosoftware Workshop UA Sample Server
opc.tcp://<hostname>:52520/OPCUA/SampleConsoleServer	Prosys OPC UA Java SDK Sample Console Server
opc.tcp://<hostname>:4841	Unified Automation Demo Server
opc.tcp://<hostname>:62541/Quickstarts/DataAccessServer	OPC Foundation QuickStart Data Access Server

where <hostname> is the host name of the computer in which the server is running.[1]

Instead of using the complete URI like this, you can alternatively define the connection in parts using the properties Protocol2, Host, Port and ServerName. These make up the Url as follows:

<Protocol>[2]://<Host>:<Port><ServerName>

The WorkshopClientConsole uses the following Uri to connect to the Technosoftware OPC UA Sample Server:

```
        private const string TechnosoftwareSampleServer Uri = "
opc.tcp://localhost:55533/TechnosoftwareSampleServer ";
```

[1] Note that 'localhost' may also work. The servers define a list of endpoints that they are listening to. The client can only connect to the server using an Url that matches one of these endpoints. But the SDK will convert it to the actual hostname, if the server does not define 'localhost' in its endpoints.

Also IP number can only be used, if the server also defines the respective endpoint using the IP number.

For Windows hostname resolution, see http://technet.microsoft.com/en-us/library/bb727005.aspx. If you are using the client in Linux, you cannot use NetBIOS computer names to access Windows servers. In general, it is best to use TCP/IP DNS names from all clients. Alternatively, you can always use the IP address of the computer, if you make sure that the server also initializes an endpoint using the IP address, in addition to the hostname.

[2] Note that not all servers support all different protocols, e.g. the OPC Foundation Java stack only supports the binary (opc.tcp) protocol at the moment.

8 Discover Servers

For UA Server discovering you can use the GetUaServers() methods. To be able to find an UA server, all UA Servers running on a machine should register with the UA Local Discovery Server using the Stack API.

If a UA Server running on a machine is registered with the UA Local Discovery Server a client can discover it using the following code:

```
// Discover all local UA servers
List<string> servers = Discover.GetUaServers(application.Configuration);

Console.WriteLine("Found local OPC UA Servers:");
foreach (var server in servers)
{
    Console.WriteLine(String.Format("{0}", server));
}
```

Remote servers can be discovered by specifiying a Uri object like shown below:

```
// Discover all remote UA servers
Uri discoveryUri = new Uri("opc.tcp://technosoftware:4840/");
servers = Discover.GetUaServers(application.Configuration, discoveryUri);

Console.WriteLine("Found remote OPC UA Servers:");
foreach (var server in servers)
{
   Console.WriteLine(String.Format("{0}", server));
}
```

9 Accessing an OPC UA Server

There are only a few classes required by an UA client to handle operations with an UA server. In general, an UA client

- creates one or more Sessions by using the Session [9.1] class
- creates one or more Subscriptions within a Session [9.1] by using the Subscription [9.9] class
- adding one or more MonitoredItems within a Subscription [9.9] by using the MonitoredItem [9.6] class

9.1 Session

The Session class inherits from the SessionClient which means all the UA services are in general accessible as methods on the Session object.

The Session object provides several helper methods including a `Session.Create()` method which Creates and Opens the Session. The process required when establishing a session with a Server is as follows:

- The Client application must choose the EndpointDescription to use. This can be done manually or by getting a list of available EndpointDescriptions by using the `Discover.GetEndpointDescriptions()` method.
- The Client takes the EndpointDescription and uses it to Create the Session object by using the `Session.Create()` method. If `Session.Create()` succeeds the client application will be able to call other methods.

Example from the WorkshopClientConsole:

```
    List<EndpointDescription> endpointDescriptions =
        Discover.GetEndpointDescriptions(application.Configuration, TechnosoftwareSampleServer
Uri);

    ConfiguredEndpoint endpoint = new ConfiguredEndpoint(null, endpointDescriptions[0]);

    mySessionSampleServer_ = Session.Create(application.Configuration,
                                            endpoint,
                                            false,
                                            application.Configuration.ApplicationName,
                                            60000,
                                            new UserIdentity(),
                                            null);
```

9.1.1 Keep Alive

After creating the session, the Session object starts periodically reading the current state from the Server at a rate specified by the KeepAliveInterval (default is 5s). Each time a response is received the state and latest timestamp is reported as a SessionKeepAliveEvent event. If the response does not arrive after 2 KeepAliveIntervals have elapsed a SessionKeepAliveEvent event with an error is raised. The KeepAliveStopped property will be set to true. If communication resumes the normal SessionKeepAliveEvent events will be reported and the KeepAliveStopped property will go back to false.

The client application uses the SessionKeepAliveEvent event and KeepAliveStopped property to detect communication problems with the server. In some cases, these communication problems will be temporary but while they are going on the client application may choose not to invoke any services because they would likely timeout. If the channel does not come back on its own the client application will execute whatever error recovery logic it has.

Client applications need to ensure that the SessionTimeout is not set too low. If a call times out the WCF channel is closed automatically and the client application will need to create a new one. Creating a new channel will take time. The KeepAliveStopped property allows applications to detect failures even if they are using a long SessionTimeout.

The following sample is taken from the WorkshopClientConsole and shows how to use the KeepAlive and Reconnect handling. After creating the session [9.1] the client can add a keep alive event handler:

```
mySessionSampleServer_.SessionKeepAliveEvent += OnSessionKeepAliveEvent;
```

Now the client gets updated with the keep alive events and can easily add a reconnect feature:

```
private void OnSessionKeepAliveEvent(object sender, SessionKeepAliveEventArgs e)
{
    Session session = (Session)sender;
    if (sender != null && session.Endpoint != null)
    {
        Console.WriteLine(Utils.Format(
            "{0} ({1}) {2}",
            session.Endpoint.EndpointUrl,
            session.Endpoint.SecurityMode,
            (session.EndpointConfiguration.UseBinaryEncoding) ? "UABinary" : "XML"));
    }
    if (e != null && mySessionSampleServer_ != null)
    {
        if (ServiceResult.IsGood(e.Status))
        {
            Console.WriteLine(Utils.Format(
                "Server Status: {0} {1:yyyy-MM-dd HH:mm:ss} {2}/{3}",
                e.CurrentState,
                e.CurrentTime.ToLocalTime(),
                mySessionSampleServer_.OutstandingRequestCount,
                mySessionSampleServer_.DefunctRequestCount));
        }
        else
        {
            Console.WriteLine(String.Format(
                "{0} {1}/{2}", e.Status,
                mySessionSampleServer_.OutstandingRequestCount,
                mySessionSampleServer_.DefunctRequestCount));
            if (reconnectPeriod_ <= 0)
            {
                return;
            }
```

```
            if (reconnectHandler_ == null && reconnectPeriod_ > 0)
            {
                reconnectHandler_ = new SessionReconnectHandler();
                reconnectHandler_.BeginReconnect(mySessionSampleServer_,
                                                 reconnectPeriod_ * 1000,
                                                 OnServerReconnectComplete);
            }
        }
    }
}
```

As soon as the session keep alive event handler (`OnSessionKeepAliveEvent`) detects that a reconnect must be done a reconnect handler is created. In the above sample the following lines are doing this:

```
if (reconnectHandler_ == null && reconnectPeriod_ > 0)
{
    reconnectHandler_ = new SessionReconnectHandler();
    reconnectHandler_.BeginReconnect(mySessionSampleServer_,
                                     reconnectPeriod_ * 1000,
                                     OnServerReconnectComplete);
}
```

As soon as the OPC UA stack reconnected to the OPC UA Server the `OnServerReconnectComplete` handler is called and can then finish the client-side actions.

The following sample is taken from the WorkshopClientConsole and shows how to implement the `OnServerReconnectComplete` handler:

```
private void OnServerReconnectComplete(object sender, EventArgs e)
{
    try
    {
        // ignore callbacks from discarded objects.
        if (!Object.ReferenceEquals(sender, reconnectHandler_))
        {
            return;
        }
        mySessionSampleServer_ = reconnectHandler_.Session;
        reconnectHandler_.Dispose();
        reconnectHandler_ = null;
        OnSessionKeepAliveEvent(mySessionSampleServer_, null);
    }
    catch (Exception exception)
    {
        //GuiUtils.HandleException(this.Text, MethodBase.GetCurrentMethod(), exception);
    }
}
```

Important in the `OnServerReconnectComplete` handler are the following lines:

1. `mySessionSampleServer_ = reconnectHandler_.Session;`
   The session used up to now must be replaced with the new session provided by the reconnect handler. The client itself does not need to create a new session, subscreiptions or MonitoredItems. That's all done by the OPC UA stack. So with taking the session provided by the reconnect handler all subriptions and MonitoredItems are then still valid and functional.
2. `reconnectHandler_.Dispose();` and `reconnectHandler_ = null;`
   This ensures that the keep alive event handler doesn't start a new reconnect again.
3. `OnSessionKeepAliveEvent(mySessionSampleServer_, null);`
   This ensures that the keep alive event handler is immediately called and status information can be updated.

9.1.2 Cache

The Session object provides a cache that can be used to store Nodes that are accessed frequently. The cache is particularly useful for storing the types defined by the server because the client will often need to check if one type is a subtype of another. The cache can be accessed via the NodeCache property of the Session object. The type hierarchies stored in the cache can be searched using the TypeTree property of the NodeCache or Session object (the both return a reference to the same object).

The NodeCache is populated with the FetchNode() method which will read all of the attributes for the Node and the fetch all of its references. The Find() method on the NodeCache looks for a previously cached version of the Node and calls the FetchNode() method if it does not exist.

Client applications that wish to use the NodeCache must pre-fetch all the ReferenceType hierarchy supported by the Server by calling FetchTypeTree() method on the Session object.

The Find() method is used during Browse of the address space [9.1].

9.1.3 Events

The Session object is responsible for sending and processing the Publish requests. Client applications can receive events whenever a new NotificationMessage is received by subscribing to the SessionNotificationEvent event.

- The SessionPublishErrorEvent event is raised whenever a Publish response reports an error.
- The SubscriptionsChangedEvent event indicates when a Subscription is added or removed.
- The SessionClosingEvent event indicates that the Session is about to be closed.

Important: The WorkshopClientConsole doesn't show the usage of these features.

9.1.4 Multi-Threading

The Session is designed for multi-threaded operation because client application frequently need to make multiple simultaneous calls to the Server. However, this is only guaranteed for calls using the Session class. Client applications should avoid calling services directly which update the Session state, e.g. CreateSession or ActivateSession.

9.2 Browse the address space

The first thing to do is typically to find the server items you wish to read or write. The OPC UA address space is a bit more complex structure than you might expect to, but nevertheless, you can explore it by browsing.

In the SDK, the address space is accessed through the Browser class. You can call browse to request nodes from the server. You start from any known node, typically the root folder and follow references between the nodes. In a first step, you create a browser object as follows:

```
// Create the browser
var browser = new Browser(mySessionSampleServer_)
{
    BrowseDirection = BrowseDirection.Forward,
    ReferenceTypeId = ReferenceTypeIds.HierarchicalReferences,
    IncludeSubtypes = true,
    NodeClassMask = 0,
    ContinueUntilDone = false,
};
```

The `Objects.ObjectsFolder` node represents the root folder, so starting from the root folder can be done with the following call:

```
// Browse from the RootFolder
ReferenceDescriptionCollection references = browser.Browse(Objects.ObjectsFolder);

GetElements(mySessionSampleServer_, browser, 0, references);
```

The `GetElements` method can be implemented like this:

```
private static void GetElements(Session session, Browser browser, uint level,
                                ReferenceDescriptionCollection references)
{
    var spaces = "";
    for (int i = 0; i <= level; i++)
    {
        spaces += "   ";
    }
    // Iterate through the references and print the variables
    foreach (ReferenceDescription reference in references)
    {
        // make sure the type definition is in the cache.
        session.NodeCache.Find(reference.ReferenceTypeId);

        switch (reference.NodeClass)
        {
            case NodeClass.Object:
                Console.WriteLine(spaces + "+ " + reference.DisplayName);
                break;

            default:
                Console.WriteLine(spaces + "- " + reference.DisplayName);
                break;
        }
        var subReferences = browser.Browse((NodeId)reference.NodeId);
        level += 1;
        GetElements(session, browser, level, subReferences);
        level -= 1;
    }
}
```

9.3 Read Value

Once you have a node selected, you can read the attributes of the node. There are actually several alternative read-calls that you can make in the Session class. In the WorkshopClientConsole this is used with

```
DataValue simulatedDataValue = mySessionSampleServer_.ReadValue(simulatedDataNodeId_);
```

where the `simulatedDataNodeId_` is defined as

```
private readonly NodeId simulatedDataNodeId_ = new NodeId("ns=2;s=Scalar_Simulation_Number");
```

This reads the value of the node `"ns=2;s=Scalar_Simulation_Number"` from the server.

In general, you should avoid calling the read methods for individual items. If you need to read several items at the same time, you should consider using `mySessionTechnosoftwareSampleServer .ReadValues()` [9.4]. It is a bit more complicated to use, but it will only make a single call to the server to read any number of values. Or if you want to monitor variables that are changing in the server, you had better use the Subscription, as described in chapter [0].

9.4 Read Values

As already mentioned above you can also read attributes of multiple nodes at the same time. This is more efficient then calling `mySessionTechnosoftwareSampleServer .ReadValue()` [9.3] several times for each of the nodes you want to get attributes from. In the WorkshopClientConsole this is used with

```
// The input parameters of the ReadValues() method
List<NodeId> variableIds = new List<NodeId>();
List<Type> expectedTypes = new List<Type>();

// The output parameters of the ReadValues() method
List<object> values;
List<ServiceResult> errors;

// Add a node to the list
variableIds.Add(simulatedDataNodeId_);
// Add an expected type to the list (null means we get the original type from the server)
expectedTypes.Add(null);

// Add another node to the list
variableIds.Add(staticDataNodeId1_);
// Add an expected type to the list (null means we get the original type from the server)
expectedTypes.Add(null);

// Add another node to the list
variableIds.Add(staticDataNodeId2_);
// Add an expected type to the list (null means we get the original type from the server)
expectedTypes.Add(null);

mySessionSampleServer_.ReadValues(variableIds, expectedTypes, out values, out errors);
```

where the following NodeId's:

```
private readonly NodeId simulatedDataNodeId_ = new NodeId("ns=2;s=Scalar_Simulation_Number");
private readonly NodeId staticDataNodeId1_ = new NodeId("ns=2;s=Scalar_Static_Integer");
private readonly NodeId staticDataNodeId2_ = new NodeId("ns=2;s=Scalar_Static_Double");
```

This reads the value of the 3 nodes from the server.

9.5 Write Value

Like reading, you can also write values to the server. For example:

```
short writeInt = 1234;

Console.WriteLine("Write Value: " + writeInt);
StatusCode result = mySessionSampleServer_.WriteValue(staticDataNodeId1_,
                                                      new DataValue(writeInt));

// read it again to check the new value
Console.WriteLine("Node Value (should be {0}): {1}",
                  mySessionSampleServer_.ReadValue(staticDataNodeId1_).Value, writeInt);
```

As a response, you get a status code – indicating if the write was successful or not.

If the operation fails, e.g. because of a connection loss, you will get an exception. For service call errors, such that the server could not handle the service request at all, you can expect `ServiceResultException`.

9.6 Write Values

Like reading several values at once, you can also write values of multiple nodes to the server. For example:

```
writeInt = 5678;
Double writeDouble = 1234.1234;

List<NodeId> nodeIds = new List<NodeId>();
List<DataValue> dataValues = new List<DataValue>();

nodeIds.Add(staticDataNodeId1_);
nodeIds.Add(staticDataNodeId2_);

dataValues.Add(new DataValue(writeInt));
dataValues.Add(new DataValue(writeDouble));

Console.WriteLine("Write Values: {0} and {1}", writeInt, writeDouble);
result = mySessionSampleServer_.WriteValues(nodeIds, dataValues);

// read it again to check the new value
Console.WriteLine("Node Value (should be {0}): {1}",
                  mySessionSampleServer_.ReadValue(staticDataNodeId1_).Value,
                  writeInt);
Console.WriteLine("Node Value (should be {0}): {1}",
                  mySessionSampleServer_.ReadValue(staticDataNodeId2_).Value,
                  writeDouble);
```

As a response, you get a status code – indicating if the write was successful or not.

If the operation fails, e.g. because of a connection loss, you will get an exception. For service call errors, such that the server could not handle the service request at all, you can expect `ServiceResultException`.

9.7 Create a MonitoredItem

The `MonitoredItem` class stores the client-side state for a `MonitoredItem` belonging to a `Subscription` on a Server. It maintains two sets of properties:

1. The values requested when the MonitoredItem is/was created
2. The current values based on the revised values returned by the Server.

The requested properties are what is saved when then `MonitoredItem` is serialized.

The requested properties are saved when then `MonitoredItem` is serialized. Please keep in mind that the server may change (revise) some values requested by the client. The revised properties are returned in the Status property, which is of type `MonitoredItemStatus`.

The `NodeId` for the `MonitoredItem` can be specified as an absolute `NodeId` or as a starting `NodeId` followed by RelativePath string which conforms to the syntax defined in the OPC Unified Architecture Specification Part 4. The RelativePath class included in the Stack can parse these strings and produce the structures required by the UA services.

Changes to any of the properties which affect the state of the MonitoredItem on the Server are not applied immediately. Instead the ParametersModified flag is set and the changes will only be applied when the ApplyChanges method on the `Subscription` is called. Note that changes to parameters which can only be specified when the `MonitoredItem` was created are ignored if the `MonitoredItem` has already been created. Client applications that wish to change these parameters must delete the monitored item and then re-create it.

The current values for monitoring parameters are stored in the Status property. Client application must use the Status. Error property to check an error occurs while creating or modifying the item. MonitoredItems that specify a RelativePath string may have encountered an error parsing or translating the RelativePath. When such an error occurs the Error property is set and the MonitoredItem is not created.

The `MonitoredItem` maintains a local queue for data changes or events received from the Server. This means the client application does not need to explicitly process NotificationMessages and can simply read the latest value from the MonitoredItem whenever it is required. The length of the local queue is controlled by the CacheQueueSize property.

The `MonitoredItem` provides a MonitoredItemNotification event which can be used by the client application to receive events whenever a new notification is received from the Server. It is always called after it is added to the cache.

The `MonitoredItem` is designed for multi-threaded operation because the Publish requests may arrive on any thread. However, data which is accessed while updating the cache is protected with a separate synchronization lock from data that is used while updating the `MonitoredItem` parameters. This means notifications can continue to arrive while other threads update the `MonitoredItem` parameters.

Client applications must be careful when update `MonitoredItem` parameters while another thread has called ApplyChanges on the Subscription because it could lead to situation where the state of the MonitoredItem on the Server does not match the state of the MonitoredItem on the client.

The WorkshopClientConsole uses the following code to create a MonitoredItem:

```
// Create a MonitoredItem
MonitoredItem monitoredItem = new MonitoredItem
{
    StartNodeId = new NodeId(simulatedDataNodeId_),
    AttributeId = Attributes.Value,
    DisplayName = "Simulated Data Value",
    MonitoringMode = MonitoringMode.Reporting,
    SamplingInterval = 1000,
    QueueSize = 0,
    DiscardOldest = true
};
```

9.8 Create a Subscription

The `Subscription` class stores the client-side state for a Subscription with a Server. It maintains two sets of properties:

- the values requested when the Subscription is/was created and
- the current values based on the revised values returned by the Server.

The `Subscription` object is designed for batch operations. This means the subscription parameters and the `MonitoredItem` can be updated several times but the changes to the `Subscription` on the Server do not happen until the ApplyChanges() method is called. After the changes are complete the SubscriptionStatusChangedEvent event is reported with a bit mask indicating what was updated.

In normal operation, the important settings for the `Subscription` are the `PublishingEnabled` and `PublishingInterval`. The following example shows how the WorkshopClientConsole creates a subscription:

```
// Create a new subscription
Subscription mySubscription = new Subscription
{
    DisplayName = "My Subscription",
    PublishingEnabled = true,
    PublishingInterval = 500,
    KeepAliveCount = 10,
    LifetimeCount = 100,
    MaxNotificationsPerPublish = 1000,
    TimestampsToReturn = TimestampsToReturn.Both
};
```

The settings `KeepAliveCount`, `LifetimeCount`, `MaxNotificationsPerPublish` and the `Priority` of the `Subscription` can also be omitted to use the default values.

The **`KeepAliveCount`** defines how many times the PublishingInterval needs to expire without having notifications available before the server sends an empty message to the client indicating that the server is still alive but no notifications are available.

The **`LifetimeCount`** defines how many times the PublishingInterval expires without having a connection to the client to deliver data. If the server is not able to deliver notification messages after this time, it deletes the Subsction to clear the resources. The LifetimeCount must be at minimum three times the KeepAliveCount. Both values are negotiated between the client and the server.

The **`MaxNotificationsPerPublish`** is used to limit the size of the notification message sent from the server to the client. The number of notifications is set by the client but the server can send fewer notifications in one message if his limit is smaller than the client-side limit. If not all available notifications can be sent with one notification message, another notification message is sent.

The **Priority** setting defines the priority of the Subscription relative to the other Subscriptions created by the Client. This allows the server to handle Subscriptions with higher priorities first in high-load scenarios.

The `Subscription` class provides several helper methods including a Constructor with default values for several. The process required when using a subscription is as follows:

1. The `Subscription` object must be created. This can be done by using the default constructor and using one or more of the properties available.
2. Items (MonitoredItem) must be added to the subscription.
3. The subscription must be added to the session.
4. The subscription must be created for the session.
5. The subscription changes must be applied, because of the above-mentioned batch functionality.

When a `Subscription` is created, it must start sending Publish requests. It starts off the process by telling the Session object to send one request. Additional Publish requests can be send by calling the Republish() method. Applications can use additional Publish requests to compensate for high network latencies because once the pipeline is filled the Server will be able to return a steady stream of notifications.

Once the `Subscription` has primed the pump the `Session` object keeps it going by sending a new Publish whenever it receives a successful response. If an error occurs the Session raises a SessionPublishErrorEvent event and does not send another Publish.

If everything is working properly the `Session` save the message in cache at least once per keep alive interval. If a NotificationMessage does not arrive it means there are network problems, bugs in the Server or high priority Subscriptions are taking precedence. The keep alive timer is designed to detect these problems and to automatically send additional Publish requests. When the keepalive timer expires, it checks the time elapsed since the last notification message. If publishing appears to have stopped the PublishingStopped property will be true and the Subscription will raise a PublishStatusChangedEvent event and send another Publish request. Client applications must assume that any cache data values are out of date when they receive the PublishStatusChangedEvent event (e.g. the StatusCode should be set to UncertainLastKnownValue). However, client applications do not need to do anything else since the interruption may be temporary. It is up to the client application to decide when to give up on a Session and to try again with a new Session.

The `Subscription` object checks for missing sequence numbers when it receives a NotificationMessage. If there is a gap it starts a timer that will call Republish() in 1s if the gap still exists. This delay is necessary because the multi-threaded stack on the client side may process responses out of order even if they are received in order.

The `Subscription` maintains a cache of messages received. The size of this cache is controlled by the MaxMessageCount property. When a new message is received, the Subscription adds it to the cache and removes any extras. It then extracts the notifications and pushes them to the MonitoredItem identified by the ClientHandle in the notification.

The Subscription is designed for multi-threaded operation because the Publish requests may arrive on any thread. However, data which is accessed while processing an incoming message is protected with a separate synchronization lock from data that is used while updating the Subscription parameters. This means notifications can continue to arrive while network operations to update the Subscription state on the server are in progress. However, no more than one operation to update the Subscription state can proceed at one time. Closing the Session will interrupt any outstanding operations. Any synchronization locks held by the subscription are released before any events are raised.

9.9 Subscribe to data changes

In order to monitor data changes, you have to subscribe to the `MonitoredItemNotificationEvent` as shown below:

```
// Establish the notification event to get changes on the MonitoredItem
monitoredItem.MonitoredItemNotificationEvent += OnMonitoredItemNotificationEvent;
```

You also must add the MonitoredItem to the subscription

```
// Add the item to the subscription
mySubscription.AddItem(monitoredItem);
```

If you are finished with adding MonitoredItems to the subscription you have to add the subscription to the session:

```
// Add the subscription to the session
mySessionSampleServer_.AddSubscription(mySubscription);
```

Now you can finish creating the subscription and apply the changes to the session by using the following code:

```
// Create the subscription. Must be done after adding the subscription to a session
mySubscription.Create();

// Apply all changes on the subscription
mySubscription.ApplyChanges();
```

The specified event callback `OnMonitoredItemNotificationEvent` of the WorkshopClientConsole looks like:

```
private void OnMonitoredItemNotificationEvent(object sender,
                                              MonitoredItemNotificationEventArgs e)
{
    var notification = e.NotificationValue as MonitoredItemNotification;
    if (notification == null)
    {
        return;
    }
    var monitoredItem = sender as MonitoredItem;
    if (monitoredItem != null)
    {
        var message = String.Format("Event called for Variable \"{0}\" with Value = {1}.",
                                    monitoredItem.DisplayName, notification.Value);
        Console.WriteLine(message);
    }
}
```

9.10 Subscribe to events

In addition to subscribing to data changes in the server variables, you may also listen to events from event notifiers. You can use the same subscriptions, but additionally, you must also define the event filter, which defines the events that you are interested in and the event fields you wish to monitor. To make handling of the filters a bit easier the WorkshopClientConsole uses a utility class `FilterDefinition`. The following code creates a filter:

```
// the filter to use.
filterDefinition = new FilterDefinition();
```

The default constructor subscribes to all events coming from the RootFolder of the Server object (`ObjectIds.Server`) with a Severity of `EventSeverity.Min` and all events of type `ObjectTypeIds.BaseEventType`.

The `FilterDefinition` class also has a helper method to create the select clause:

```
// must specify the fields that the client is interested in.
filterDefinition.SelectClauses = filterDefinition.ConstructSelectClauses(
                    mySessionSampleServer_,
                    ObjectIds.Server,
                    ObjectTypeIds.BaseEventType
                    );
```

The code above creates a select claus which includes all fields of the BaseEventType. Another helper method of the `FilterDefinition` class creates the MonitoredItem:

```
// create a monitored item based on the current filter settings.
MonitoredItem monitoredEventItem =
                    filterDefinition.CreateMonitoredItem(mySessionSampleServer_);
```

Now we can subscribe to the event changes with:

```
// set up callback for notifications.
monitoredEventItem.MonitoredItemNotificationEvent += OnMonitoredEventItemNotification;
```

See the WorkshopClientConsole for the code of the callback `OnMonitoredEventItemNotification()`. After creating the MonitoredItem it must be added to the subscription and the changes must be applied:

```
mySubscription.AddItem(monitoredEventItem);
mySubscription.ApplyChanges();
mySubscription.ConditionRefresh();
```

9.11 Calling Methods

OPC UA also defines a mechanism to call methods in the server objects. To find out if an object defines methods, you can call `ReadNode()` of the session and use as parameter the NodeId you want to call a method from:

```
private readonly NodeId methodsNodeId_ = new NodeId("ns=2;s=Methods");
private readonly NodeId callHelloMethodNodeId_ = new NodeId("ns=2;s=Methods_Hello");

INode node = mySessionSampleServer_.ReadNode(callHelloMethodNodeId_);

MethodNode methodNode = node as MethodNode;

if (methodNode != null)
{
    // Node supports methods
}
```

OPC UA Methods have a variable list of Input and Output Arguments. To make this example simple we have choosen a method with one input and one output argument. To be able to call a method you need to know the node of the method, in our example `callHelloMethodNodeId_` but also the parent node, in our example `methodsNodeId_`. Calling the method then done by

```
NodeId methodId = callHelloMethodNodeId_;
NodeId objectId = methodsNodeId_;

VariantCollection inputArguments = new VariantCollection();
Argument argument = new Argument();
inputArguments.Add(new Variant("from Technosoftware"));

var request = new CallMethodRequest { ObjectId = objectId,
                                      MethodId = methodId,
                                      InputArguments = inputArguments };

var requests = new CallMethodRequestCollection { request };

CallMethodResultCollection results;
DiagnosticInfoCollection diagnosticInfos;

ResponseHeader responseHeader = mySessionSampleServer_.Call(
                        null,
                        requests,
                        out results,
                        out diagnosticInfos);

if (StatusCode.IsBad(results[0].StatusCode))
{
    throw new ServiceResultException(new ServiceResult(
                    results[0].StatusCode,
                    0, diagnosticInfos,
                    responseHeader.StringTable));
}

Console.WriteLine(String.Format("{0}", results[0].OutputArguments[0]));
```

9.12 History Access

The UA Servers may also provide history information for the nodes. You can read the Historizing attribute of a Variable node to see whether history is supported. For this example we use the Historical Access Sample Server with the Endpoint Uri opc.tcp://<localhost>:55551/TechnosoftwareHistoricalAccessServer.

9.12.1 Check if a Node supports historizing

You can get information about a node by reading the Attribute `Attributes.AccessLevel` and check whether the node supports HistoricalAccess. The code we use for this is shown below:

```
ReadValueId nodeToRead = new ReadValueId();
nodeToRead.NodeId = dynamicHistoricalAccessNodeId_;
nodeToRead.AttributeId = Attributes.AccessLevel;
nodesToRead.Add(nodeToRead);

// Get Information about the node object
mySessionHistoricalAccessServer_.Read(
                    null,
                    0,
                    TimestampsToReturn.Neither,
                    nodesToRead,
                    out values,
                    out diagnosticInfos);

ClientBase.ValidateResponse(values, nodesToRead);
ClientBase.ValidateDiagnosticInfos(diagnosticInfos, nodesToRead);


for (int ii = 0; ii < nodesToRead.Count; ii++)
{
    byte accessLevel = values[ii].GetValue<byte>(0);

    // Check if node supports HistoricalAccess
    if ((accessLevel & AccessLevels.HistoryRead) != 0)
    {
        // Node supports HistoricalAccess
    }
}
```

9.12.2 Reading History

To actually read history data you use the `HistoryRead()` method of the Session. The example below reads a complete history for a single node (specified by nodeId):

```
HistoryReadResultCollection results = null;

// do it the hard way (may take a long time with some servers).
ReadRawModifiedDetails details = new ReadRawModifiedDetails();
details.StartTime = DateTime.UtcNow.AddDays(-1);
details.EndTime = DateTime.MinValue;
details.NumValuesPerNode = 10;
details.IsReadModified = false;
details.ReturnBounds = false;

HistoryReadValueId nodeToReadHistory = new HistoryReadValueId();
nodeToReadHistory.NodeId = dynamicHistoricalAccessNodeId_;

HistoryReadValueIdCollection nodesToReadHistory = new HistoryReadValueIdCollection();
nodesToReadHistory.Add(nodeToReadHistory);

// Read the historical data
mySessionHistoricalAccessServer_.HistoryRead(
    null,
    new ExtensionObject(details),
    TimestampsToReturn.Both,
    false,
    nodesToReadHistory,
    out results,
    out diagnosticInfos);

ClientBase.ValidateResponse(results, nodesToRead);
ClientBase.ValidateDiagnosticInfos(diagnosticInfos, nodesToRead);

if (StatusCode.IsBad(results[0].StatusCode))
{
    throw new ServiceResultException(results[0].StatusCode);
}

// Get the historical data
HistoryData historyData = ExtensionObject.ToEncodeable(results[0].HistoryData) as HistoryData;
```

What you need to be aware of is that there are several “methods” that the historyRead supports, depending on which HistoryReadDetails you use. For example, in the above example we used ReadRawModifiedDetails, to read a raw history (the same structure is used to read Modified history as well, therefore the name).

Why Technosoftware GmbH?...

- Professionalism
  Technosoftware GmbH is, measured by the number of employees, truly not a big company. However, when it comes to flexibility, service quality, and adherence to schedules and reliability, we are surely a great company which can compete against the so called leaders in the industry. And this is THE crucial point for our customers.

- Continuous progress
  Lifelong learning and continuing education is, especially in the information technology, essential for future success. Concerning our customers, we will constantly be accepting new challenges and exceeding their requirements again and again. We will continue to do everything to fulfill the needs of our customers and to meet our own standards.

- High Quality of Work
  We reach this by a small, competent and dynamic team of coworkers, which apart from the satisfaction of the customer; take care of a high quality of work. We concern the steps necessary for it together with consideration of the customers' requirements.

- Support
  We support you in all phases – consultation, direction of the project, analysis, architecture & design, implementation, test and maintenance. You decide on the integration of our coworkers in your project; for an entire project or for selected phases.

Technosoftware GmbH
Windleweg 3, CH-5235 Rüfenach

sales@technosoftware.com
www.technosoftware.com