OPC UA SDK
Client + Server Development

OPC UA SDK für .NET

Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

Buch - Die gesamte Anleitung als eBook

Development Guides

Client Development Guide Server Development Guide Anwendungsfälle Häufige Fragen

Download

Das OPC UA .NET SDK kommt mit einer **Testlizenz die je Anwendungsstart 30 Minuten uneingeschränkt zur Client- und Serverentwicklung** verwendet werden kann. Sollte diese Einschränkung ihre Evaluationsmöglichkeiten einschränken, besteht die Möglichkeit eine **alternative Testlizenz** bei uns **kostenlos** zu beantragen. Fragen Sie einfach unseren Support (via support@traeger.de) oder lassen Sie sich gleich direkt von uns beraten und offene Fragen durch unsere Entwickler klären!

OPC UA .NET SDK für Clients – Evaluationspaket[1)]
Download ZIP Archiv von Opc.UaFx.Client (Version: 2.42.0.0 – 2024-06-05)
Download NuGet Paket von Opc.UaFx.Client (Version: 2.42.0.0 – 2024-06-05)

OPC UA .NET SDK für Clients und Server – Evaluationspaket[2)]
Download ZIP Archiv von Opc.UaFx.Advanced (Version: 2.42.1.0 – 2024-07-25)
Download NuGet Paket von Opc.UaFx.Advanced (Version: 2.42.1.0 – 2024-07-25)

OPC UA .NET SDK für LabVIEW Clients – Evaluationspaket[3)]
Download ZIP Archiv von Opc.UaFx.Client.LabView (Version: 1.1.3.0 – 2024-06-14)

OPC UA .NET SDK für Unity Clients – Evaluationspaket[4)]

OPC UA .NET SDK für .NET Framework 3.5 Clients und Server – Evaluationspaket[5)]
Download ZIP Archiv von Opc.UaFx.Advanced (Version: 2.0.1.1 – 2017-06-07)
Download NuGet Paket von Opc.UaFx.Advanced (Version: 2.0.1.1 – 2017-06-07)

OPC Watch (Version: 2.42.1.0 – 2024-07-25)
Ein kostenloser und einfacher, aber professioneller OPC UA Client für den Zugriff auf OPC UA Server.

Versionshistorie - Die Liste der Verbesserungen pro Version

Preview Download

OPC UA .NET SDK für Clients – PREV1 für OPC UA v1.04[6)]

OPC UA .NET SDK für Clients und Server – PREV1 für OPC UA v1.04[7)]

Runtime Download

Zur Anbindung von **OPC Classic** Servern in 64 Bit-Anwendungen und zur Enumeration (= Discovery) von lokalen OPC Classic Servern müssen die Core Components von der OPC Foundation auf dem Zielsystem installiert sein. Diese finden sie unter anderen auf der Seite der OPC Foundation oder auch hier:

OPC Core Components Redistributables[8)]
Download ZIP Archiv von OPC Core Components (Version: 3.00.107 – 2018-01-30)
Download ZIP Archiv von OPC Core Components (Version: 3.00.108 – 2019-12-20)

OPC UA Client

OPC UA Client Development Guide

Beispiel Code: OPC UA Client

- C#
- VB

```
namespace Client
{
    using System;
    using System.Threading;

    using Opc.UaFx.Client;

    public class Program
    {
        public static void Main()
        {
            using (var client = new OpcClient("opc.tcp://localhost:4840")) {
                client.Connect();

                while (true) {
                    var temperature = client.ReadNode("ns=2;s=Temperature");
                    Console.WriteLine("Current Temperature is {0} °C", temperature);

                    Thread.Sleep(1000);
                }
            }
        }
    }
}
```

```
Imports System
Imports System.Threading

Imports Opc.UaFx.Client

Namespace Client
    Public Class Program
        Public Shared Sub Main()
            Using client = New OpcClient("opc.tcp://localhost:4840")
                client.Connect()

                While True
                    Dim temperature = client.ReadNode("ns=2;s=Temperature")
                    Console.WriteLine("Current Temperature is {0} °C", temperature)

                    Thread.Sleep(1000)
                End While
            End Using
        End Sub
    End Class
End Namespace
```

OPC UA Server

OPC UA Server Development Guide

Beispiel Code: OPC UA Server

- C#
- VB

```
namespace Server
{
    using System.Threading;

    using Opc.UaFx;
    using Opc.UaFx.Server;

    public class Program
    {
        public static void Main()
        {
            var temperatureNode = new OpcDataVariableNode<double>("Temperature", 100.0);

            using (var server = new OpcServer("opc.tcp://localhost:4840/", temperatureNode))
{
                server.Start();

                while (true) {
                    if (temperatureNode.Value == 110)
                        temperatureNode.Value = 100;
                    else
                        temperatureNode.Value++;

                    temperatureNode.ApplyChanges(server.SystemContext);
                    Thread.Sleep(1000);
                }
            }
        }
    }
}
```

```
Imports System.Threading

Imports Opc.UaFx
Imports Opc.UaFx.Server

Namespace Server
    Public Class Program
        Public Shared Sub Main()
            Dim temperatureNode = New OpcDataVariableNode(Of Double)("Temperature", 100.0)

            Using server = New OpcServer("opc.tcp://localhost:4840/", temperatureNode)
                server.Start()

                While True
                    If (temperatureNode.Value = 110) Then
                        temperatureNode.Value = 100
                    Else
                        temperatureNode.Value += 1
                    End If

                    temperatureNode.ApplyChanges(server.SystemContext)
                    Thread.Sleep(1000)
                End While
            End Using
        End Sub
    End Class
End Namespace
```

[1], [2], [3], [4], [5] Mit Ihrem „License Code“ wird das Paket zur produktiven Vollversion.

[6], [7] Nicht für den produktiven Einsatz empfohlen.

[8] Bereitgestellt durch die OPC Foundation

OPC UA SDK
Client + Server Development

Client Development Einführung

Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

Die Verbindung zum Server

Connect

Das passiert beim Aufruf von 'Connect':

1. Es wird geprüft, ob eine **Serveradresse festgelegt** wurde (ServerAddress Eigenschaft).
2. Der Client ändert seinen Status (**OpcClient.State** Eigenschaft) auf den Wert ***Connecting***.
3. Der Client **prüft seine Konfiguration** auf Gültigkeit und Schlüssigkeit.
4. Anschließend versucht der Client einen **Endpunkt ausfindig** zu **machen**.
   Dies geschieht mittels DiscoveryClient. Hierbei werden die Endpunkte mit der gewünschten Endpunktkonfiguration verglichen und der Endpunkt ausgewählt, der der Konfiguration entspricht oder zumindest genügt. Nach welchen Kriterien ein Endpunkt ausgewählt wird, hängt von den vorgenommenen Einstellungen des Clients ab.
5. Danach erstellt der Client die **Konfiguration für eine neue Sitzung**.
6. Es werden die **Instanzzertifikate geprüft**:
   1. das Zertifikat des Clients (je nach verwendeter Security Konfiguration)
   2. das Zertifikat des Servers (das Zertifikat, das über den Endpunkt bereitgestellt wird)
7. Schließlich wird ein **Channel erstellt**, welcher als Verbindung zwischen Client und Server dient.
8. Über den Channel wird versucht, eine **Sitzung** zu **erstellen**.
9. Nach weiterem Austausch und Prüfung von Sitzungsdaten wird die **Sitzung aktiviert**.
10. Schlussendlich werden noch die **verfügbaren Namespaces abgerufen**
11. sowie Vorkehrungen für die **Überwachung der Verbindung** getroffen:
    1. „KeepAlive-Tracking“ zur Erkennung von Verbindungsabbrüchen
    2. „Notification-Tracking“ zum Empfangen von Benachrichtigungen
12. Der Client ändert seinen Status (**OpcClient.State** Eigenschaft) auf den Wert ***Connected***.

Disconnect

Das passiert beim Aufruf von 'Disconnect':

1. Der Client ändert seinen Status (**OpcClient.State** Eigenschaft) auf den Wert ***Disconnecting***.
2. Der Client **gibt** alle **erworbenen Ressourcen** wieder **frei** (wie z.B. File Handles zu OPC UA Dateiknoten)
3. **Beendet** die **Überwachung der Verbindung**
4. Die **aktive Sitzung wird beendet**.
5. Der beim Connect erstellte **Channel wird geschlossen und verworfen**.
6. Der Client ändert seinen Status (**OpcClient.State** Eigenschaft) auf den Wert ***Disconnected***.

BreakDetection

Die „BreakDetection“-**Abbrucherkennung** bezeichnet den Mechanismus, der für die Erkennung von Verbindungsabbrüchen zuständig ist. Hierbei kommt das KeepAlive Verfahren zum Einsatz, um so einen **Timeout der Verbindung zum Server** zu **erkennen**. Kommt es zum Timeout, so versucht der Client **automatisch wieder eine Verbindung zum Server herzustellen**. Im Falle einer neu erstellten Verbindung kommt es dabei auch häufig zu einer **neuen Sitzung**. Während beim KeepAlive in regelmäßigen Abständen KeepAlive-Nachrichten zwischen Client und Server ausgetauscht werden, um so die Verbindung „zu testen“ und „aufrecht zu erhalten“, wird bei zu langen Antwortzeiten (= Timeout

erreicht?) auf eine KeepAlive-Nachricht angenommen, dass die Verbindung unterbrochen ist. Ist das der Fall, wird in immer größeren Abständen eine weitere KeepAlive-Nachricht gesendet. Bleiben auch diese unbeantwortet, wird von einer abgebrochenen Verbindung ausgegangen und der zuvor beschriebene Mechanismus zur Wiederaufnahme der Verbindung tritt in Kraft. Aktiviert wird die Abbrucherkennung, welche standardmäßig aktiviert ist, über die **OpcClient.UseBreakDetection** Eigenschaft.

Verbindungsparameter

Damit der Client eine Verbindung zum Server aufbauen kann, müssen die richtigen Parameter festgelegt werden. **Generell** wird die **Adresse des Servers** (**OpcClient.ServerAddress** Eigenschaft) **benötigt**, unter der er zu erreichen ist. Die Uri-Instanz (= Uniform Resource Identifier) liefert dem Client alle primär nötigen Informationen über den Server. So enthält zum Beispiel die Server-Adresse „opc.tcp://192.168.0.80:4840“ die Information des Schemas „opc.tcp“ (möglich sind „http“, „https“, „opc.tcp“, „net.tcp“ und „net.pipe“) welches festlegt, über welches Protokoll die Daten wie ausgetauscht werden sollen. Generell ist bei OPC UA Servern im lokalen Netzwerk „opc.tcp“ zu empfehlen. Server außerhalb des lokalen Netzwerks sollten „http“, besser noch „https“ verwenden. Weiter definiert die Adresse, dass der Server auf dem Rechner mit der IP Adresse „192.168.0.80“ ausgeführt wird und auf Anfragen über den Port mit der Nummer „4840“ lauscht (was der Standardport für die OPC UA ist, benutzerdefinierte Portnummern sind ebenfalls möglich). Anstelle der statischen IP Adresse kann auch der DNS Name des Rechners verwendet werden, so könnte anstelle von „127.0.0.1“ auch „localhost“ verwendet werden.

Definiert der Server **keinen Endpunkt** (engl. Endpoint), dessen Strategie (engl. Policy) den Sicherheitsmodus **„None“** (möglich sind zudem „Sign“ und „SignAndEncrypt“) zum Datenaustausch verwendet, **muss diese Endpoint-Policy** manuell **konfiguriert werden** (**OpcClient.Security.EndpointPolicy** Eigenschaft). Wird hingegen ein **Endpunkt mit der Strategie „None“** vom Server bereitgestellt, **wählt der Client automatisch diesen aus**. Dieses **Verhalten**, welches standardmäßig aktiviert ist, **kann deaktiviert** werden (**OpcClient.Security.UseOnlySecureEndpoints** Eigenschaft). Die automatische Auswahl des Endpunkts **kann** zudem **gemäß der OPC Foundation durchgeführt werden**, indem man den Client so konfiguriert, dass **der Endpunkt, der per Definition das höchste Policy-Level** (eine Zahl) **definiert**, automatisch der „beste“ für den Datenaustausch ist. Dieses Verhalten ist standardmäßig deaktiviert, **kann** aber auf Wunsch **aktiviert werden** (**OpcClient.Security.UseHighLevelEndpoint** Eigenschaft).

Verwendet der Server eine Zugriffssteuerung, zum Beispiel über eine ACL (= Access Control List), also eine Zugriffssteuerungsliste, **müssen** dem Server entsprechend gültige **Benutzerdaten zur Feststellung der Identität des Benutzers** des Clients übermittelt werden, bevor eine Verbindung zustande kommen kann. Hierbei besteht die Möglichkeit, die Identität des Benutzers über ein Benutzername-Passwort-Paar (**OpcClientIdentity** Klasse) oder über ein Zertifikat (**OpcCertificateIdentity** Klasse) auszuweisen. Die entsprechende Identität muss dann **dem Client mitgeteilt werden** (**OpcClient.Security.UserIdentity** Eigenschaft), damit dieser beim Verbindungsaufbau die Identität dem Server übermitteln kann.

Endpunkte

Die Endpunkte (engl. Endpoints) ergeben sich aus dem Kreuzprodukt der verwendeten Basis-Adressen des Servers und der vom Server unterstützten Sicherheitsstrategien. Dabei ergeben sich die Basis-Adressen aus jedem unterstützten Schema-Port-Paar und dem Host (IP Adresse oder DNS Name), wobei mehrere

Schemen (möglich sind „http“, „https“, „opc.tcp“, „net.tcp“ und „net.pipe“) zum Datenaustausch auf unterschiedlichen Ports festgelegt werden können. Die so verlinkten Strategien (engl. Policies) legen das Vorgehen beim Datenaustausch fest. Bestehend aus dem Policy-Level, dem Sicherheitsmodus (engl. Security-Mode) und dem Sicherheitsalgorithmus (engl. Security-Algorithm) legt jede Policy die Art des sicheren Datenaustauschs fest.

Werden zum Beispiel zwei Sicherheitsstrategien (engl. Security-Policies) verfolgt, dann könnten diese wie folgt definiert sein:

- Security-Policy A: Level=0, Security-Mode=None, Security-Algorithm=None
- Security-Policy B: Level=1, Security-Mode=Sign, Security-Algorithm=Basic256

Werden weiter zum Beispiel drei Basis-Adressen (engl. Base-Addresses) wie folgt für verschiedene Schemen festgelegt:

- Base-Address A: "https://mydomain.com/"
- Base-Address B: "opc.tcp://192.168.0.123:4840/"
- Base-Address C: "opc.tcp://192.168.0.123:12345/"

So ergeben sich daraus durch das Kreuzprodukt die folgenden Endpunktbeschreibungen:

- Endpoint 1: Address="https://mydomain.com/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 2: Address="https://mydomain.com/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256
- Endpoint 3: Address="opc.tcp://192.168.0.123:4840/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 4: Address="opc.tcp://192.168.0.123:4840/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256
- Endpoint 5: Address="opc.tcp://192.168.0.123:12345/", Level=0, Security-Mode=None, Security-Algorithm=None
- Endpoint 6: Address="opc.tcp://192.168.0.123:12345/", Level=1, Security-Mode=Sign, Security-Algorithm=Basic256

Dabei wird der Addressteil des Endpunktes immer vom Client benötigt (via Konstruktor oder via **ServerAddress** Eigenschaft). Während der Client standardmäßig versucht, einen Endpunkt mit dem Security-Mode „None“ zu finden, muss manuell die Policy des Endpunkts konfiguiert werden (**OpcClient.Security.EndpointPolicy** Eigenschaft), wenn kein solcher existiert.

Aufklärung über Zertifikate

Zertifikate in OPC UA

Zertifikate dienen dazu, die **Authentizität** (einfach: Echtheit) und **Integrität** (einfach: Vertraulichkeit) von Client- und Serveranwendungen **sicherzustellen**. Sie dienen somit einer Client- sowie einer Serveranwendung als eine Art Personalausweis. Da dieser „Personalausweis“ in Form einer Datei vorliegt, muss diese irgendwo gespeichert werden. Wo die Zertifikate gespeichert werden, kann individuell entschieden werden. Unter Windows kann jedes Zertifikat direkt an das **System** übergeben werden und Windows kümmert sich um den **Speicherort**. Alternativ können auch **benutzerdefinierte Speicherorte** (= Verzeichnisse) festgelegt werden.

Es gibt verschiedene Typen von Speicherorten für Zertifikate:

- **Speicherort für Anwendungszertifikate**
  Der auch als **Application Certificate Store** bezeichnete Speicherort enthält ausschließlich die Zertifikate der Anwendungen, die diesen Store als Application Certificate Store verwenden. Hier speichert eine Client- beziehungsweise Serveranwendung ihr eigenes Zertifikat.
- **Speicherort für Zertifikate vertrauenswürdiger Zertifikataussteller**
  Der auch als **Trusted Issuer Certificate Store** bezeichnete Speicherort enthält ausschließlich Zertifikate von Zertifizierungsstellen, welche weitere Zertifikate ausstellen. Hier speichert eine Client- beziehungsweise Serveranwendung alle Zertifikate der Aussteller (engl. issuer), deren Zertifikate standardmäßig als vertrauenswürdig (engl. trusted) eingestuft werden sollen.
- **Speicherort für vertrauenswürdige Zertifikate**
  Der auch als **Trusted Peer Store** bezeichnete Speicherort enthält ausschließlich Zertifikate, die als vertrauenswürdig eingestuft werden. Hier speichert ein Client die **Zertifikate vertrauenswürdiger Server** beziehungsweise ein Server die **Zertifikate vertrauenswürdiger Clients**.
- **Speicherort für verweigerte Zertifikate**
  Der auch als **Rejected Certificate Store** bezeichnete Speicherort enthält ausschließlich Zertifikate, die als nicht vertrauenswürdig eingestuft werden. Hier speichert ein Client die **Zertifikate nicht vertrauenswürdiger Server** beziehungsweise ein Server die **Zertifikate nicht vertrauenswürdiger Clients**.

Unabhängig ob der Speicherort irgendwo im System liegt oder im Dateisystem über ein Verzeichnis, es gilt generell, dass Zertifikaten, die im **Trusted Store** liegen, **vertraut** wird und Zertifikaten, die im **Rejected Store** liegen, **nicht vertraut** wird. Zertifikate, die in keinem von beiden enthalten sind, werden automatisch in den Trusted Store gespeichert, wenn das Zertifikat des im Zertifikat hinterlegten Zertifikatsausstellers im Trusted Issuer Store existiert; andernfalls wird es automatisch in den Rejected Store gespeichert. Ist selbst ein vertrauenswürdiges Zertifikat abgelaufen oder können dessen hinterlegte Informationen nicht erfolgreich durch die Zertifizierungsstelle geprüft werden, dann wird das Zertifikat als nicht vertrauenswürdig eingestuft und in den Rejected Store gespeichert. Dabei wird es auch aus dem Trusted Peer Store wieder entfernt. Ebenso kann ein Zertifikat ungültig werden, wenn es in einer Zertifikatsperrliste (engl. Certificate Revocation List - CRL) gelistet ist, welche im jeweiligen Store separat geführt werden kann.

Ein Zertifikat, das der Client vom Server beziehungsweise das der Server vom Client erhält, wird **vorerst immer** als *unbekannt* eingestuft und somit auch als **nicht vertrauenswürdig** (engl. untrusted) behandelt. Damit ein Zertifikat als vertrauenswürdig behandelt wird, muss es als solches deklariert werden. Dies geschieht, indem das Zertifikat des Clients im Trusted Store des Servers beziehungsweise das Zertifikat des Servers im Trusted Store des Clients gespeichert wird.

Abhandlung eines Serverzertifikats beim Client:

1. Der Client ermittelt das Zertifikat des Servers, auf dessen Endpunkt er sich verbinden soll.
2. Der Client prüft das Zertifikat des Servers.
    1. Ist das Zertifikat gültig?
        1. Ist das Gültigkeitsdatum überschritten?
        2. Ist das Zertifikat des Ausstellers gültig?
    2. Existiert das Zertifikat im Trusted Peer Store?
        1. Ist es in eine Zertifikatsperrliste (CRL) eingetragen?
    3. Existiert das Zertifikat im Rejected Store?
3. Ist das Zertifikat vertrauenswürdig, dann stellt der Client eine Verbindung zum Server her.

Abhandlung eines Clientzertifikats beim Server:

1. Der Server erhält beim Verbindungsaufbau durch den Client das Zertifikat des Clients.
2. Der Server prüft das Zertifikat des Clients.
    1. Ist das Zertifikat gültig?
        1. Ist das Gültigkeitsdatum überschritten?
        2. Ist das Zertifikat des Ausstellers gültig?
    2. Existiert das Zertifikat im Trusted Peer Store?
        1. Ist es in eine Zertifikatsperrliste (CRL) eingetragen?
    3. Existiert das Zertifikat im Rejected Store?
3. Ist das Zertifikat vertrauenswürdig, dann lässt der Server die Verbindung des Clients zu und bedient ihn.

Im Falle, dass die Prüfung des Zertifikats der jeweiligen Gegenstelle fehlschlägt, kann über benutzerdefinierte Mechanismen die Prüfung erweitert werden und selbst auf Benutzerebene noch entschieden werden, ob das Zertifikat akzeptiert wird oder nicht.

Arten von Zertifikaten

Allgemein: Selbstsignierte Zertifikate vs. signierte Zertifikate

Ein Zertifikat ist vergleichbar mit einer Urkunde. Eine Urkunde kann von jedem ausgestellt und auch von jedem unterzeichnet werden. Hierbei besteht aber ein wesentlicher Unterschied darin, ob der Unterzeichner einer Urkunde auch wirklich für dessen Korrektheit bürgt (wie ein Notar), oder ob der Unterzeichner der Inhaber der Urkunde selbst ist. Insbesondere Urkunden der letzteren Art sind nicht besonders vertrauenerweckend, da keine anerkannte (gesetzliche) Instanz wie zum Beispiel ein Notar sich für den Inhaber der Urkunde verbürgt.

Da Zertifikate mit Urkunden vergleichbar sind und ebenfalls eine (digitale) Unterschrift (= Signierung) aufweisen müssen, verhält es sich hier genauso. Die Signatur eines Zertifikats muss dem Empfänger einer Zertifikatskopie Auskunft darüber geben, wer sich für dieses Zertifikat verbürgt. Dabei gilt immer, dass der Aussteller (engl. issuer) eines Zertifikats zugleich dieses auch signiert. Wenn der **Aussteller eines Zertifikats gleich der Zielperson** (engl. subject) des Zertifikats ist, dann spricht man von einem **selbstsignierten Zertifikat** (Subject ist gleich Issuer). Wenn der **Aussteller eines Zertifikats nicht gleich der Zielperson** des Zertifikats ist, dann spricht man von einem **(einfachen / normalen / signierten) Zertifikat** (Subject ist nicht gleich Issuer).

Da Zertifikate insbesondere im Kontext der OPC UA zur Authentifizierung einer Identität (einer bestimmten Client- oder Serveranwendung) eingesetzt werden, sollten signierte Zertifikate als Anwendungszertifikate für die eigene Anwendung verwendet werden. Ist hingegen der Aussteller zugleich auch Inhaber des Zertifikats, sollte dessen selbstsigniertem Zertifikat nur dann vertraut werden, wenn man den Inhaber als vertrauenswürdig einstuft. Solche Zertifikate wurden, wie eben beschrieben, durch den Aussteller des Zertifikats signiert. Das hat wiederum zur Folge, dass das Zertifikat des Ausstellers (engl. issuer certificate) im **Trusted Issuer Store** der Anwendung liegen muss. Ist das Zertifikat des Ausstellers dort nicht auffindbar, gilt die Zertifikatskette (engl. certificate chain) als unvollständig, woraus folgt, dass das Zertifikat der Gegenstelle nicht akzeptiert wird. Ist hingegen das Zertifikat vom Aussteller des Anwendungszertifikats wiederum kein selbstsigniertes Zertifikat, dann muss das Zertifikat von dessen Aussteller im **Trusted Issuer Store** verfügbar sein.

Benutzeridentifizierung

Benutzeridentifizierung mit Zertifikate

Neben dem Einsatz eines Zertifikats als *Personalausweis* für Client- beziehungsweise Serveranwendungen, kann ein Zertifikat auch zur Identifizierung eines Benutzers verwendet werden. Eine Clientanwendung wird stets durch einen bestimmten Benutzer bedient, durch die er mit dem Server operiert. Je nach Serverkonfiguration kann ein Server vom Client zusätzlich Informationen über die Identität des Benutzers des Clients anfordern. Hier besteht die Möglichkeit, dass der Benutzer seine Identität in Form eines Zertifikats ausweist. In wieweit ein Server das Zertifikat auf seine Gültigkeit, Echtheit und Vertraulichkeit prüft, hängt vom jeweiligen Server ab. Der vom Framework bereitgestellte Server prüft dabei ausschließlich, ob die Thumbprintinformationen der Benutzeridentität in seiner Zugriffskontrollliste (engl. Access Control List - ACL) für zertifikatbasierte Benutzeridentitäten auffindbar ist.

Aspekte der Sicherheit

Produktiver Einsatz

Das primäre Ziel des Frameworks ist es, den Einstieg in OPC UA so einfach wie möglich zu gestalten. Dieser Grundgedanke führt leider auch dazu, dass ohne weiterführende Konfiguration des Clients keine völlig sichere Verbindung / Kommunikation zwischen Client und Server stattfindet. Wurde jedoch der finale Spike implementiert und getestet, sollte über die *Aspekte der Sicherheit* nachgedacht werden.

Auch wenn man bei der Entwicklung des Clients von den vom Server bereitgestellten Sicherheitsmechanismen abhängig ist, sollte stets die möglichst beste Wahl getroffen werden. So sollte generell der Endpunkt (**OpcClient.ServerAddress** Eigenschaft und **OpcClient.Security.EndpointPolicy** Eigenschaft) verwendet werden, welcher die bestmögliche sichere Verbindung bereitstellt (z.B. „https“ anstelle von „http“ als Schema). Dazu gehören Endpunkte, welche die bestmögliche Sicherheitsstrategie (engl. Security-Policy) verfolgen. Dabei muss auf den Sicherheitsmodus (engl. Security-Mode) und den Sicherheitsalgorithmus (engl. Security-Algorithm) geachtet werden. Will man gemäß der OPC Foundation den quasi „per se“ sichersten Endpunkt wählen, kann man sich auch auf den Endpunkt mit dem höchsten Security-Level berufen (**OpcClient.Security.UseHighLevelEndpoint** Eigenschaft).

Zum vereinfachten Handling von Zertifikaten akzeptiert der Client standardmäßig jedes Zertifikat (**OpcClient.Security.AutoAcceptUntrustedCertificates** Eigenschaft), auch die, die er unter produktiven Bedingungen verweigern sollte. Denn nur Zertifikate, die dem Client bekannt sind (diese befinden sich im TrustedPeer Zertifikatspeicher), gelten als wirklich vertrauenswürdig. Davon abgesehen sollte stets die Gültigkeit der Zertifikate geprüft werden, dazu zählt unter anderem das „Verfallsdatum“ des Zertifikats. Weiter ist es ratsam, die im Zertifikat referenzierten Domänen zu prüfen (**OpcClient.Security.VerifyServersCertificateDomains** Eigenschaft). Andere Eigenschaften des Zertifikats oder lockerere Regeln für die Gültigkeit und Vertrauenswürdigkeit eines Serverzertifikats können zudem manuell durchgeführt werden (**OpcClient.CertificateValidationFailed** Ereignis).

Verwendet der Server ein Sicherheitsverfahren, welches den Zugriff über Benutzeridentitäten (engl. User Identities) regelt, sollte stets eine konkrete User Identity gewählt werden (**OpcClient.Security.UserIdentity** Eigenschaft). Davon abgesehen, dass der Zugriff über eine anonyme Identität meist eingeschränkt ist, kann durch den Einsatz einer konkreten Identität wie einem Zertifikat (engl. Certificate) oder einem Benutzername-Passwort-Paar dem Client gegebenenfalls auch der Zugriff zu sensibleren Daten gestattet werden. Zugleich erhöht zum Beispiel eine Certificate Identity die Sicherheit bei der signierten Datenübertragung.

Inhaltsverzeichnis