OPC UA SDK
Client + Server Development

# Client Development Introduction

## Tested? You want it?

License Model Prices Quotation Order Now

# Connection to the Server

## Connect

This is happening on calling 'Connect':

1. Checking if an **address was set** (ServerAddress property).
2. The Client changes its status (**OpcClient.State** property) to the value ***Connecting***.
3. The Client **checks its configuration** for validity and conclusiveness.
4. Next the Client tries to **find an endpoint**.
   This happens via DiscoveryClient where endpoints with the desired endpoint configuration are compared and the endpoint fulfilling or at least sufficing the configuration is chosen. An endpoint is chosen depending on the used settings of the Client.
5. Next the Client creates the **configuration for a new session**.
6. **Instance certificates** are **checked**:
    1. Client certificate (depending on security configuration)
    2. Server certificate (the certificate provided by the endpoint)
7. A **channel** acting as a connection between Client and Server is **created**.
8. Attempt to **create a session** via the channel.
9. After further exchange and checking of session data the **session** gets **activated**.
10. Finally, **available Namespaces** are **retrieved**
11. And precautions for **connection surveillance** are taken:
    1. “KeepAlive-Tracking” for detecting connection abortions
    2. “Notification-Tracking” for receiving notifications
12. The Client changes its status (**OpcClient.State** property) to the value ***Connected***.

## Disconnect

This is happening on calling 'Disconnect':

1. The Client changes its status (**OpcClient.State** property) to the value ***Disconnecting***.
2. The Client **releases** all **gathered resources** (e.g. File Handles for OPC UA File Nodes)
3. **Ends** the **connection surveillance**
4. The **active session is ended**.
5. The **channel** created during Connect **is closed and disposed of**.
6. The Client changes its status (**OpcClient.State** property) to the value ***Disconnected***.

## BreakDetection

The “BreakDetection” is a mechanism responsible for the detection of connection abortions using “KeepAlive”-Tracking to **detect a timeout of the connection to the Server**. If there is a timeout, the Client **automatically tries to establish a connection to the Server**. In case of a newly created connection a **new session** often happens. While KeepAlive messages are sent between Client and Server in KeepAlive in order to “test” the connection and “hold it up”, it is assumed that the connection was interrupted when response times to a KeepAlive message are too long (= reached timeout?). In that case another KeepAlive message is sent in increasing time intervals. An aborted connection is assumed if these messages also are unanswered and the previously described mechanism to re-establish the connection is introduced. The abortion detection is active by default and can be activated via the

**OpcClient.UseBreakDetection** property.

# Connection Parameters

In order for the Client to connect to the Server the correct parameters have to be set. **Generally** the **address of the Server** (**OpcClient.ServerAddress** property) **is needed**. The Uri (= Uniform Resource Identifier) instance feeds the Client with every primarily necessary information about the Server. The Server-Address “opc.tcp://192.168.0.80:4840” e.g. contains the information of the scheme “opc.tcp” (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) which establishes over which protocol data is exchanged in which way. In general “opc.tcp” is recommended for OPC UA Servers in a local network. Servers outside a local network should use “http” or even better “https”. Furthermore the address defines that the Server is executed on a computer with the IP address “192.168.0.80” and listens to requests via the port numbered “4840” (which is the default port for the OPC UA, custom port numbers are also possible). Instead of a static IP address the DNS name of the computer can be used as well, so instead of “127.0.0.1” also “localhost” can be used.

If the Server does **not define an endpoint** whose policy uses the security mode **“None”** (also possible are “Sign” and “SignAndEncrypt”) for data exchange, **this Endpoint-Policy has to be configured** manually (**OpcClient.Security.EndpointPolicy** property). If, however, an **endpoint with the policy “None”** is provided by the Server, **the Client automatically chooses it**. This **behavior** is activated by default and **can be deactivated** (**OpcClient.Security.UseOnlySecureEndpoints** property). The automatic choice of the endpoint **can be done according to the OPC Foundation** by configuring the Client in a way that **the endpoint defining the highest Policy-Levels per definition** (a number) automatically is the “best” for data exchange. This behavior is deactivated by default but **can be activated** (**OpcClient.Security.UseHighLevelEndpoint** property).

**If the Server uses an access control**, for example via an ACL (= Access Control List), valid **user data for identifying the user has to be given** to the Server before a connection can be established. The user identity can either be varified through a username-password pair (**OpcClientIdentity** class) or through a certificate (**OpcCertificateIdentity** class). Then the identity has to be **mentioned to the Client** (**OpcClient.Security.UserIdentity** property) in order for it to deliver the identity to the Server while connecting.

# Endpoints

Endpoints result from the cross product of the used base addresses of the Server and the security strategies supported by the Server. The results are the base addresses of every scheme-port pair supported, while several schemes (possible are “http”, “https”, “opc.tcp”, “net.tcp” and “net.pipe”) can be determined for data exchange on different ports. The hereby linked policies determine the procedure during the data exchange. Consisting of the Policy Level, the Security-Mode and the Security-Algorithm, every policy determines the kind of secure data exchange.

For example, when two Security-Policies are followed, they can be defined as follows:

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

When furthermore, for example, three Base-Addresses are combined for different schemes as follows:

- 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/"

The results will be the following endpoint descriptions through the cross product:

- 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

Here the address part of the endpoint is always needed by the Client (via constructor or via **ServerAddress** property). While the Client tries to find an endpoint with the Security-Mode “None” by default, the policy of the endpoint has to be configured manually (**OpcServer.Security.EndpointPolicies** property) when none exists.

# Information about Certificates

## Certificates in OPC UA

Certificates are used to **ensure** the **authenticity** and **integrity** of Client and Server applications. Therefore they act as a kind of identity card for Client as well as Server application. This “identification card” has to be stored somewhere as it exists as a form of file. The decision on where certificates are stored is individual. On Windows, every certificate can be passed to the **system** and Windows takes care of the **Store**. As an alternative **custom Stores** (= directories) can be set.

There are different types of Stores for certificates:

- **Store for application certificates**
  The Store also called **Application Certificate Store** exclusively contains certificates of those applications that use this Store as an Application Certificate Store. Here a Client / Server application saves its own certificate.
- **Store for certificates from trustworthy certificate issuers**
  The Store also called **Trusted Issuer Certificate Store** exclusively contains certificates from certificate issuers that issue further certificates. Here a Client / Server application saves all certificates from issuers whose certificates shall be treated as trusted by default.
- **Store for trustworthy certificates**
  The Store also called **Trusted Peer Store** exclusively contains certificates treated as trusted. Here a Client saves the **certificates from trusted Servers** and a Server saves the **certificates from trusted Clients**.
- **Store for rejected certificates**
  The Store also called **Rejected Certificate Store** exclusively contains certificates that are decreed as not trusted. Here a Client saves the **certificates from untrusted Servers** and a Server saves

the **certificates from untrusted Clients**.

Regardless of the Store being located somewhere in the system or in the file system via a directory, generally certificates in the **Trusted Store** are **trusted** and certificates in the **Rejected Store** are **untrusted**. Certificates not belonging to either of the former are automatically saved in the Trusted Store, if the certificate of the certificate issuer mentioned in the certificate is deposited in the Trusted Issuer Store; otherwise it is automatically saved in the Rejected Store. Even if a trustworthy certificate has expired or if its deposited information cannot be successfully verified through the certification center the certificate is graded as not trustworthy and saved in the Rejected Store. During this process it also is removed from the Trusted Peer Store. A certificate can also expire when it is listed in a CRL (=Certificate Revocation List), which can be kept separately in the concerning store.

A certificate that the Client receives from the Server or the other way around is **for the moment always** classified as *unknown* and therefore also treated as **untrusted**. In order for a certificate to be treated as trusted it must be declared as such. This happens by saving the certificate of the Client in the Trusted Store of the Server and the certificate of the Server in the Trusted Store of the Client.

Dealing with a Server certificate at the Client:

1. The Client establishes the certificate of the Server on whose endpoint it shall connect with.
2. The Client verifies the certificate of the Server.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?
    3. Does the certificate exist in the Rejected Store?
3. When the certificate is trusted, the Client establishes a connection to the server.

Dealing with a Client certificate at the Server:

1. The Server receives the Client's certificate from the Client while connecting.
2. The Server verifies the certificate of the Client.
    1. Is the certificate valid?
        1. Has the effective date expired?
        2. Is the issuer's certificate valid?
    2. Does the certificate exist in the Trusted Peer Store?
        1. Is it listed in a CRL?
    3. Does the certificate exist in the Rejected Store?
3. When the certificate is trusted, the Server allows the connection of the Client and operates it.

In case the verification of the certificate from the respective counterpart fails, the verification can be extended by custom mechanisms and still decided on user scale, if the certificate gets accepted or not.

# Types of Certificates

General: Self-Signed Certificates vs. Signed Certificates

A certificate is comparable to a document. A document can be issued by everybody and can also be signed by everybody. However, the main difference here is, if the signee of a document really vouches for its correctness (like a notary) or if the signee is the owner of the document itself. Especially documents of the latter are not really inspiring confidence because no (legally) recognized instance as e.g. a notary vouches

for the owner of the document.

As certificates are comparable to documents and also have to show a (digital) signature, the situation here is the same. The signature of a certificate has to tell the recipient of the certificate copy, who vouches for this certificate. Herefore it always applies that the issuer of a certificate also signs it. When the **issuer of a certificate equals the subject** of the certificate, you call this a **self-signed certificate** (subject equals issuer). When the **issuer of a certificate does not equal the subject** of the certificate, you call this a **(simple / normal / signed) certificate** (subject does not equal issuer).

As certificates are used especially in the context of the OPC UA authentication of an identity (of a certain Client or Server application), signed certificates should be used as application certificates for the own application. If, however, the issuer of the certificate also its owner, this self-signed certificate should only be trusted when the owner is rated as trusted. Such certificates were, as described, signed by the issuer of the certificate. Therefore, the issuer certificate has to be located in the **Trusted Issuer Store** of the application. When the issuer certificate cannot be found there , the certificate chain is declared incomplete and the certificate is not accepted by the counterpart. Yet, if the issuer certificate of the issuer of the application certificate is not a self-signed certificate, the certificate of its issuer has to be available in the **Trusted Issuer Store**.

# User Identification

User Identification through Certificates

Next to the use of a certificate as an *identification card* for Client / Server applications, a certificate can also be used to identify a user. A Client application is always operated by a certain user by whom it operates with the Server. Depending on the Server configuration a Server can request additional information about the identity of the Client's user from the Client. The user has the possibility to prove his identity through a certificate. How thoroughly a Server is examining the certificate on validity, authenticity and confidentiality depends on the Server. The Server provided by the Framework exclusively checks, if the Thumbprint information of the user identity can be found in its ACL (=Access Control List) for certificate-based user identities.

# Aspects of Security

# Productive use

The primary goal of the Framework is to make getting the grips of the OPC UA as easy as possible. This basic thought sadly also leads to the fact that without secondary configuration of the Server a completely save connection / communication between Client and Server does not occur. Yet, if the final Spike has been implemented and tested, second thought should be given to the *aspects of security*.

Even if one is dependant on the security mechanisms provided by the Server while developing a Client, one should always make the best choice possible. In general, the choice should always be the endpoint (**OpcClient.ServerAddress** property and **OpcClient.Security.EndpointPolicy** property) that provides the most secure connection (e.g. “https” instead of “http” as a scheme). This includes endpoints that follow the best Security-Policy possible. Keep an eye on the Security-Mode and the Security-Algorithm. According to the OPC Foundation, if you want to have the savest endpoint, refer to the endpoint with the highest Security-Level (**OpcClient.Security.UseHighLevelEndpoint** property).

For simplified handling of certificates the Client accepts every certificate by default

(**OpcClient.Security.AutoAcceptUntrustedCertificates** property), also those it should deny under productive conditions because only certificates known to the Client (located in the Trusted Peer Store) apply as truly trusted. Apart from that the validity of a certificate should always be verified, including the “expiration date” of the certificate, for example. Furthermore it is advisable to check the domains referenced in the certificate (**OpcClient.Security.VerifyServersCertificateDomains** property). Other properties of the certificate or looser rules for the validity and trustworthiness of a Server certificate can be furthermore carried out manually (**OpcClient.CertificateValidationFailed** event).

If the Server uses a security process which controls the access via user identities, a concrete user identity should always be chosen (**OpcClient.Security.UserIdentity** property). Apart from the fact that anonymous identities almost always have only limited access, the Client can be granted access to more sensitive data when a concrete identity (e.g. a certificate or a username-password pair) is used. At the same time security is higher at e.g. a signed data transmission using a Certificate Identity.

# Table of Contents