OPC UA

SDK for Java

Client Tutorial

Version 4.0.0

Table of Contents

1. Introduction

Hello and welcome to OPC UA!

This is the Prosys OPC UA SDK for Java tutorial for client application development. With this quick introduction you should be able to grab the basic ideas behind the Prosys OPC UA SDK for Java.

Note that this tutorial assumes that you are already familiar with the basic concepts of OPC UA communications, although you can get started without much prior knowledge.

For a full reference on OPC UA communications, we recommend the book *OPC Unified Architecture* by Mahnke, Leitner and Damm (Springer-Verlag, 2009, ISBN 978-3-540-68898-3).

2. Installation

See the installation instructions in the 'README.txt' file (or the brief version on the download page). The README file also contains notes about the usage and deployment of external libraries used by the SDK.

There is also a basic starting guide with tips on Java development tools and on using the Prosys OPC UA SDK for Java with the Eclipse IDE located in the 'Prosys_OPC_UA_SDK_for_Java_Starting_Guide' PDF file next to this guide in the distribution package.

3. Sample Applications

The SDK contains a sample client application in the SampleConsoleClient Java class. This tutorial will refer to the code in the sample application while explaining the different steps to take in order to accomplish the main tasks of an OPC UA client.

In addition, the samples also contain a SimpleClient Java class that presents a very straightforward example of the most simplified client application you can create with the SDK.

4. UaClient Object

The UaClient class is the main class you will be working with. It encapsulates the connection to the OPC UA server and handles the various details of the actual OPC UA communications, thus providing you with a simple interface to access from your applications. These are the lines in the 'SampleConsoleClient.java' file that create the UaClient object:

```
protected UaClient client;
...
client = new UaClient(serverAddress);
```

5. Server Connection

In the previous example, the serverAddress argument defines the server you are connecting to. Some sample addresses are provided in the following table:

Address	Server
opc.tcp://<hostname>:52520/OPCUA/SampleConsoleServer	Prosys OPC UA SDK for Java Sample Console Server
opc.https://<hostname>:52443/OPCUA/SampleConsoleServer	Prosys OPC UA SDK for Java Sample Console Server
opc.tcp://<hostname>:53530/OPCUA/SimulationServer	Prosys OPC UA Simulation Server
opc.tcp://<hostname>:62541/Quickstarts/DataAccessServer	OPC Foundation QuickStart Data Access Server

<hostname> is the hostname of the computer in which the server is running.

The servers define a list of endpoints that they are listening to. The actual hostname in the endpoint may differ from the one that you use for connection. 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.

You need to define the complete address either in the constructor or with the `setAddress()` method. The first part of the address defines the transport protocol to use. `opc.tcp` refers to UA TCP communication which is usually the preferred protocol. An alternative protocol is `opc.https`, which some servers may support as well.

6. Security Settings

OPC UA applications enable full security that is integrated into the communications. You can decide in the client which kind of security settings you want to use or make available in your application. Usually, the end user should be able to configure the security level of each connection according to his needs.

6.1. Application Identity

All OPC UA applications must define some characteristics of themselves. This information is communicated to other applications via the OPC UA protocol when the applications are connected.

For secure communications, the applications must also define an *Application Instance Certificate*, which they use to authenticate themselves to other applications they are communicating with. Depending on the selected security level, servers may only accept connections from clients that they trust.

6.1.1. Application Description

The characteristics of an OPC UA application is defined in the following structure:

```
static String APP_NAME = 'SampleConsoleClient';
[...]
ApplicationDescription appDescription = new ApplicationDescription();
// 'localhost' (all lower case) in the ApplicationName and
// ApplicationURI is converted to the actual host name of the computer
// in which the application is run
appDescription.setApplicationName(new LocalizedText(APP_NAME +
    "@localhost"));
appDescription.setApplicationUri("urn:localhost:OPCUA:" + APP_NAME);
appDescription.setProductUri("urn:prosysopc.com:OPCUA:" + APP_NAME);
appDescription.setApplicationType(ApplicationType.Client);
```

ApplicationName is used in user interfaces as a name identifier for each application instance.

ApplicationUri is a unique identifier for each running instance.

ProductUri, on the other hand, is used to identify your product and should therefore be the same for all instances. It should refer to your own domain, for example, to ensure that it is globally unique.

Since the identifiers should be unique for each instance (i.e. installation), it is a good habit to include the hostname of the computer in which the application is running in both the ApplicationName and the ApplicationUri. The SDK supports this by automatically converting `localhost` to the actual hostname of the computer (e.g. 'myhost'). Alternatively, you can use `hostname`, which will be replaced with the full hostname, including the possible domain name part (e.g. 'myhost.mydomain.com').

The URIs must be valid identifiers, i.e. they must begin with a scheme, such as ‘urn:’ and may not contain any space characters. There are some applications in the market, which use invalid URIs and may therefore cause some errors or warnings with your application.

6.1.2. Application Instance Certificate

You can define the Application Instance Certificate for the client by setting an `ApplicationIdentity` for the `UaClient` object. The simplest way to do this is:

```
  final ApplicationIdentity identity = ApplicationIdentity.loadOrCreateCertificate(
        appDescription,
     "Sample Organisation",
     /* Private Key Password, optional */"opcua",
     /* Key File Path */privatePath,
     /* CA certificate & private key, optional */null,
     /* Key Sizes for instance certificates to create, optional */keySizes,
     /* Enable renewing the certificate */true);
```

In this sample we are creating a self-signed certificate using the service of `ApplicationIdentity.loadOrCreateCertificate()`. On the first run, it creates the certificate and the private key and stores them in the files SampleConsoleClient@hostname_keysize.der and SampleConsoleClient@hostname_keysize.pem respectively (hostname is replaced with the actual hostname of the machine where the application is running and keysize is replaced with the size of the public and private key). The private key is used by the client to create a secret token that is sent to the server. The certificate is used by the server to decrypt the token and validate that the client created it.

The fourth parameter in loadOrCreateCertificate() simply defines the path where the certificate files are stored. Do not mind about it at the moment, it will be clarified later.

The last parameter enables automatic certificate renewal when a certificate expires.

As the name refers, the certificate is used to identify each application instance. That means that on every computer, the application has a different certificate. The certificate contains the ApplicationUri, which also identifies the computer in which the application is run, and must match the one defined in the ApplicationDescription. Therefore, we provide the appDescription as a parameter for loadOrCreateCertificate(), which extracts the ApplicationUri from it.

If your application does not use security, you may also create the identity without any certificate by using the default constructor. However, you should always set the ApplicationDescription to the identity.

Note that if some other application gets the same key pair, it can pretend to be the same client application. The private key should be kept safe in order to reliably verify the identity of this application. Additionally, you may secure the usage of the private key with a password that is required to open it for use (but you need to add that in clear text in your application code or prompt it from the user). The certificate is public and can be distributed and stored freely in the servers and anywhere else.

The SDK stores private keys in '.pem' format, which supports password protection. If you get the certificate and private key from an external CA, you may get a '.pfx' file: if such is present (and '.pem' is not present), the application will use it to read the private key.

6.1.3. Issuer Certificate

More ideally, the certificates would be signed by a recognized Certificate Authority (CA) instead of using the self-signed keys as above.

If you wish to create your own KeyPair in your application, you can do that with the SDK like this:

```
KeyPair issuerCertificate =
    ApplicationIdentity.loadOrCreateIssuerCertificate(
        "ProsysSampleCA", privatePath, "opcua", 3650, false);
```

The sample code creates the issuer key pair with a private key password (“opcua”) for “ProsysSampleCA” for 10 years (3650 days). The key pair is stored in the privatePath (which refers to the PKI directory of the validator, as above).

The self-made issuer key does not replace a real CA. In real installations, it is always best to establish a central CA and create all keys for the applications using the CA. In this scenario, you can copy the certificate of the CA to the trust list of each OPC UA application. This will enable the applications to automatically trust all keys created by the CA.

6.1.4. Multiple Application Instance Certificates

OPC UA specification defines different security profiles, which may require different kind of Application Instance Certificates, for example with different key sizes. The SDK enables usage of several certificates by defining an array of keySizes, e.g.:

```
// Use 0 to use the default keySize and default file names (for other
// values the file names will include the key size.
int[] keySizes = new int[] { 0, 4096 };
```

The identity is then initialized as

```
// Define the client application identity, including the security
// certificate
final ApplicationIdentity identity = ApplicationIdentity
    .loadOrCreateCertificate(appDescription, "Sample Organisation",
    /* Private Key Password */"opcua",
    /* Key File Path */privatePath,
    /* CA certificate & private key */issuerCertificate,
    /* Key Sizes for instance certificates to create */keySizes,
    /* Enable renewing the certificate */true);
```

6.1.5. HTTPS Certificate

Starting from 4.0.0 HTTPS Certificate is no longer needed or created on the client side. See Migration Guide and Release Notes for more details.

6.1.6. Assigning the Application Identity

Now, we can finally just assign the created identity to the UaClient object:

```
client.setApplicationIdentity(identity);
```

6.2. Security Modes

6.2.1. SecurityMode for UA TCP

Once the certificates are defined, you may decide the level of security that is used in the UA TCP binary communications by setting the SecurityMode:

```
client.setSecurityMode(SecurityMode.NONE);
```

This setting enables access without any security features. It is usually easier to verify that the communication can be established without security, however please note that this SecurityMode should only be used in isolated networks. If you only need to ensure that the messages cannot be tampered you can set the SecurityMode to signed mode, for example:

```
client.setSecurityMode(SecurityMode.BASIC128RSA15_SIGN);
```

This will add digital signatures to every communication message, ensuring that no one can alter the communication on the wire. However the messages wont be encrypted. In order to that, you can encrypt all messages and ensure that third parties cannot listen to the communication either by using:

```
client.setSecurityMode(SecurityMode.BASIC128RSA15_SIGN_ENCRYPT);
```

Only SecurityModes having encryption is recommended for public networks.

BASIC128RSA15 is the most lightweight security profile and supported by most OPC UA applications.

Other alternative security profiles defined in OPC UA 1.03 are BASIC256 and BASIC256SHA256.

> BASIC128RSA15 and BASIC256 are deprecated in OPC UA 1.04, due to some security issues and they are no longer recommended.
>
> OPC UA 1.04 also defines new security policies, AES128_SHA256_RSAOAEP, AES256_SHA256_RSAPSS to replace the deprecated ones. Since different applications will support different policies in practice, you may still need to use the deprecated policies sometimes.
>
> It is also recommended to enable the users of your application to configure the security policy to use.

In practice, you can use only security modes that are enabled in the server that you are connecting to. If you don't know which they are, call

```
client.getSupportedSecurityModes();
```

6.2.2. HTTPSSecurityPolicy

If you use the HTTPS protocol for server communication the UaClient will negotiate a usable TLS security policy with the server application. You can define which policies your application supports with

```
// The TLS security policies to use for HTTPS
Set<HttpsSecurityPolicy> supportedHttpsModes = new HashSet<HttpsSecurityPolicy>();
// HTTPS was added in UA 1.02
supportedHttpsModes.addAll(HttpsSecurityPolicy.ALL_102);
supportedHttpsModes.addAll(HttpsSecurityPolicy.ALL_103);
supportedHttpsModes.addAll(HttpsSecurityPolicy.ALL_104);
client.getHttpsSettings().setHttpsSecurityPolicies(supportedHttpsModes);
```

The constants ALL_102, ALL_103 and ALL_104 define which (HTTPS) security policies were considered safe in which OPC UA specification versions.

In order to be able to make a connection with HTTPS, you must also be able to validate the certificates properly. See Validating HTTPS Certificates for details about that.

In general, HTTPS is quite tricky in practice, and you should really consider twice if you really want to use that. Check how your servers support it and use it if you need it. But usually you should do better with the UA TCP protocol instead.

6.3. User Identity

In addition to verifying the identity of the applications, OPC UA also enables verification of user identities. In UaClient, you can define the identity of the user with the UserIdentity class. The SampleConsoleClient does not do that by default, as each server defines what kind of user identities it supports. You can define a user identity that uses a standard username and password combination

```
client.setUserIdentity(new UserIdentity("my_name", "my_password"));
```

Another alternative is to use a certificate and private key, similar to the application instance identity, or a WS-SecurityToken provided by an external security system (e.g. SAML or Kerberos). To find out which user token types are supported by the server, call

```
client.getSupportedUserIdentityTokens();
```

6.4. Validating Server Certificates

An integral part of all OPC UA applications, in addition to defining their own security information, is of course, to validate the security information of the other party.

To validate the certificates of OPC UA servers, you need to define a CertificateValidator in the UaClient. This validator is used to validate the certificates received from the servers automatically.

To provide a standard certificate validation mechanism, the OPC UA Java Stack contains a specific implementation of the CertificateValidator, the DefaultCertificateValidator. You can create the validator as follows:

```
    // Use PKI files to keep track of the trusted and rejected server
    // certificates...
    final PkiDirectoryCertificateStore certStore = new PkiDirectoryCertificateStore();
    final DefaultCertificateValidator validator = new DefaultCertificateValidator(certStore);
    client.setCertificateValidator(validator);
```

The way this validator stores the received certificates is defined by the certStore, which in the example is an instance of PkiDirectoryCertificateStore. It keeps the certificates in a file directory structure, such as 'PKI/CA/certs' and 'PKI/CA/rejected'. The trusted certificates are stored in the 'certs' folder and the untrusted in 'rejected'. By default, the certificates are not trusted so they are stored in 'rejected'. You can then manually move the trusted certificates to the 'certs' directory.

Additionally, you can plug a custom handler to the Validator by defining a ValidationListener:

```
validator.setValidationListener(validationListener);

private static DefaultCertificateValidatorListener validationListener = new MyCertificateValidationListener();
```

where

```
/**
 * A sampler listener for certificate validation results.
 */
public class MyCertificateValidationListener implements DefaultCertificateValidatorListener {

  @Override
  public ValidationResult onValidate(Cert certificate, ApplicationDescription applicationDescription,
      EnumSet<CertificateCheck> passedChecks) {
    // Called whenever the PkiFileBasedCertificateValidator has
    // validated a certificate {
        [...]
}
```

The SampleConsoleClient application uses the listener to prompt the user whether to accept the server certificate. The user can accept the certificate permanently, just once or reject it completely. In the first case, the certificate is placed in the 'certs' folder automatically, and in the latter cases, it is placed in the 'rejected' folder. In the last case, connection to the server is cancelled due to the certificate rejection.

You are, of course, free to use the listener to define any custom logic, but in principle, you should only trust certificates for which passedChecks equals CertificateCheck.COMPULSORY. Normally you can trust self-signed certificates, so that check is not included in the COMPULSORY definition. Most OPC UA certificates will probably be self-signed, because they are easy to generate automatically. A proper Certificate Authority should be preferred in real systems to enable a properly managed system for certificate management.

6.4.1. Validating HTTPS Certificates

HTTPS connection is established with special HTTPS certificates that differ from the Application Instance Certificates. Only client needs to validate the server’s HTTPS Certificate, the server does not validate client’s HTTPS Certificate. By default MessageSecurityMode is None with HTTPS, in which case the servers will not authenticate the client applications. In order to enable application authentication, MessageSecurityMode Sign must be used. In this case the applications are authenticated using the Application Instance Certificates similar to the UA TCP communication.

In order to validate the HTTPS Certificates of servers, you can plug a validator (which can also be the same that you use for validating Application Instance Certificates) to HttpsSettings like this:

```
client.getHttpsSettings().setHttpsCertificateValidator(validator);
```

Typically, HTTPS certificates are signed with CA certificates, so in order to trust a HTTPS certificate of a server in your application, you must copy the respective CA certificate to the certificate store as trusted.

6.5. Teach Yourself the Security Details

OPC UA uses security heavily to guarantee that the applications can be safely used in real production environments. The security only works when configured properly, so you should make yourself familiar with the concepts and learn to configure these systems.

Read the *OPC Unified Architecture* book by Mahnke et al. for more details on the OPC UA security settings and how they should be applied. The security technology follows standard PKI (Public Key Infrastructure) principles, so all material related to that can also be used to understand the basics.

Also, try different settings in different environments so that you know more than you guess.

7. Connecting and Disconnecting

Once you have managed to get over the first compulsory hurdles of defining where and how to connect, you can simply connect to the server with

```
client.connect();
```

If that fails, you will get an exception. If the actual connection cannot be made, you will get a ServerConnectionException. If you get a connection, but something goes wrong in the server, the UaClient typically throws a ServiceException. You may also see a ServiceFaultException, ServiceResultException or some other runtime exception, which are thrown from the actual OPC UA Stack.

> Often the original exception from the Stack is also available as the Cause from the ServiceException.

Once you have the connection, you can start playing with the server. When running the

SampleConsoleClient application, you are presented with a menu in the console output where you can pick up the tasks you may want to try out.

Once you are finished testing, you can disconnect the client by simply calling

```
client.disconnect();
```

7.1. Connection Monitoring

Each service call that you make to the server can fail, for example, if the connection is lost due to network problems or the server is simply shut down.

7.1.1. ServiceException

The service calls (described in the following sections) raise a ServiceException in case of communication or other service errors.

7.1.2. Timeout

The OPC UA Java Stack handles temporary communication errors by retrying to establish a lost connection in certain cases. It also takes care of timeout handling, i.e. the synchronous service calls are monitored for a response until the timeout delay occurs without a response. You can define the default timeout (in milliseconds) to use in the UaClient with:

```
client.setTimeout(30000);
```

7.1.3. Server Status Monitoring

When connected to a server, the UaClient periodically monitors the value of ServerStatus, which is a compulsory Object in the OPC UA server address space. It will perform a check every StatusCheckInterval, which is 1 second by default. The client uses a specific timeout setting, StatusCheckTimeout (default value is 10 seconds) to detect communication breaks.

You can listen to changes in the status by defining your own ServerStatusListener, for example as follows:

```
/**
 * A sampler listener for server status changes.
 */
public class MyServerStatusListener implements ServerStatusListener {
  @Override
  public void onShutdown(UaClient uaClient, long secondsTillShutdown, LocalizedText shutdownReason) {
    // Called when the server state changes to Shutdown
    SampleConsoleClient.printf("Server shutdown in %d seconds. Reason: %s\n", secondsTillShutdown,
        shutdownReason.getText());
  }

  @Override
  public void onStateChange(UaClient uaClient, ServerState oldState, ServerState newState) {
    // Called whenever the server state changes
    SampleConsoleClient.printf("ServerState changed from %s to %s\n", oldState, newState);
    if (newState.equals(ServerState.Unknown)) {
      SampleConsoleClient.println("ServerStatusError: " + uaClient.getServerStatusError());
    }
  }

  @Override
  public void onStatusChange(UaClient uaClient, ServerStatusDataType status, StatusCode code) {
    // Called whenever the server status changes, typically every
    // StatusCheckInterval defined in the UaClient.
    // SampleConsoleClient.println("ServerStatus: " + status + ", code: " + code);
  }
}
```

You can then set the client to use the new listener with:

```
protected ServerStatusListener serverStatusListener = new MyServerStatusListener();
...
client.addServerStatusListener(serverStatusListener);
```

7.1.4. Automatic Reconnect

The UaClient enables automatic reconnections in case the communication fails. Whenever the status read fails due to a connection or timeout error or if the server notifies about a shutdown, the UaClient will start to perform reconnect attempts every second in accordance to the procedure suggested in the OPC UA specifications.

If you wish to disable the automatic reconnect feature, call UaClient.setAutoReconnect(false). In this case, you can try to reconnect yourself by calling UaClient.reconnect() until it succeeds.

8. OPC UA Server Address Space

OPC UA server applications provide all the data that they have available in their address space which is constructed in a standardised way according to the OPC UA Address Space Model. This helps client applications locate all relevant data from the server even if they don't have any prior knowledge about it.

The OPC UA client applications identify data in the OPC UA server using Node Identifiers (NodeIds). NodeIds are used to uniquely identify all information in the address space which consists of Nodes that can be Objects, Variables or various Types. Nodes have Attributes that define their properties and, for example, contain the data that the Node provides (in the Value Attribute). Nodes are connected to each by References that signify their relationship, for example, the HasComponent Reference connects an Object and a Variable inside it. NodeIds are used when the client sends read or write requests to the server, for example. If the client applications don't have the NodeId of a certain Node available, they can *browse* the server address space to find it.

You can use the Prosys OPC UA Client application to explore the address space of any OPC UA server visually and access the information inside it.

8.1. Browse the Address Space

Typically, the first thing to do is 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, but nevertheless, you can explore it by browsing.

In the UaClient, the address space is accessed through the AddressSpace property. You can call browse() to request Nodes from the server.

The SampleConsoleClient uses an internal variable, called nodeId to keep track of the "current Node" that is the target of all client operations. The nodeId is initialized to the value Identifiers.RootFolder, which is a standard NodeId defined in the OPC UA specification. It corresponds to the root of the address space, which all servers must support. In addition, the servers must provide three standard subfolders under the RootFolder: ObjectsFolder, TypesFolder and ViewsFolder. You can start browsing from one of these to dynamically explore the available data and metadata (such as types) available from the server.

The Identifiers class is generated in the OPC UA Java Stack to contain the NodeId of every Node that is defined in the OPC UA specification.

So, in order to browse the address space with the SampleConsoleClient, you start from the RootFolder and follow References between the Nodes. There may be a huge number of References from a Node, so you can define some communication limits to the server. You can set these with the different properties of the AddressSpace, e.g.:

```
client.getAddressSpace().setMaxReferencesPerNode(1000);
client.getAddressSpace().setReferenceTypeId(Identifiers.HierarchicalReferences);
```

by which you define a limit of 1000 References per call to the server and that you only wish to receive the hierarchical References between the Nodes.

The AdressSpace will in fact use the MaxReferencesPerNode internally while communicating with the server. In practice, it will make subsequent calls to the server until it receives all the References, even in the case that the current Node has more References than defined by the limit. The limit is just necessary to avoid timeouts and too large messages in the communication.

Now, if you call

```
List<ReferenceDescription> references = client.getAddressSpace().browse(nodeId);
```

you will get a list of ReferenceDescription entries from the server. From these, you can find the target Nodes, which you can browse next. In the SampleConsoleClient, you may choose which Node to browse next, or to end browsing and stay at the Node you are at that point. Check the sample code to see the specifics of the methods that are used to let you browse around the address space.

8.2. Browsing Through the Nodes

An alternative way to browsing using specific NodeIds is to follow the references between Node objects. You can access the References simply with node.getReferences(), for example. See the section Using Node Objects for more about that.

9. Read Values

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 UaClient. In SampleConsoleClient we use the basic

```
DataValue value = client.readAttribute(nodeId, attributeId);
```

which reads the value of a single Attribute from a Node in the server. The Attribute to read is defined by the attributeId. Valid IDs are defined in the Attributes class. Note that different Node types (or NodeClasses according to the OPC UA terminology) support different Attributes. For example, the Attributes.Value attribute is only supported by the Variable and VariableType Nodes.

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 use readAttributes() (to read several Attributes from one Node), readValues() (to read the Value Attribute for several Variables) or consider using read(). The read() method is a bit more complicated to use, but it will only make a single call to the server to read any number of Attributes of any Nodes.

If you actually want to monitor Variables that are changing on the server, you had better use the Subscriptions, as described below in Subscribe to Data Changes.

The method will throw ServiceException or StatusException if the call does fail (see Exceptions When Operations Fail for more information).

10. Write Values

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

```
boolean status = client.writeAttribute(nodeId, attributeId, value);
```

The method will throw ServiceException or StatusException if the call does fail (see Exceptions When Operations Fail for more information).

The return value will indicate if the write operation completes successfully and synchronously (true) or completes asynchronously (false).

Similar to the read methods, you also have better options for writing several values at the same time: writeAttributes(), writeValues() and the generic write().

11. Exceptions When Operations Fail

If any service call or operation fails, you will get an Exception. For service call errors, such as when the server could not handle the service request at all, you can expect a ServiceException. When performing a single operation, any failure (for example calling readAttribute() with an invalid nodeId or attributeId or calling writeValue() to a variable that does not permit changes) will produce a StatusException.

If you perform several operations inside a single call (such as readValues()), you can only expect a ServiceException. For each operation you will get a StatusCode that indicates which individual operation succeeded and which failed. Use StatusCode.isBad() and .isGood() to check whether the operation failed or not. The StatusCode provides a complete status code, which you can check against all status codes defined in the OPC UA specification. In case of failure, you may also get additional information in a DiagnosticInfo structure. These fields are present in the exceptions. You can also examine the result codes of the last service call from client.getLastServiceDiagnostics() and getLastOperationDiagnostics().

StatusCode defines actually three different severities: Good, Bad and Uncertain. Good usually comes without additional information, whereas Bad and Uncertain will also include a more detailed error code. The status codes defined in the OPC UA specification are available from the OPC UA Java Stack class StatusCodes. The actual values are UnsignedIntegers to which you can compare them. For example StatusCodes.Bad_UserAccessDenied.equals(exception.getStatusCode().getValue()).

12. Subscribe to Data Changes

In order to monitor changes in the server, you need to define subscriptions. These include a number of monitored items, which you listen to. To monitor data changes, you use the MonitoredDataItem class. For example:

```
subscription = new Subscription();
MonitoredDataItem item = new MonitoredDataItem(nodeId, attributeId, MonitoringMode.Reporting);
subscription.addItem(item);
client.addSubscription(subscription);
```

This defines a subscription with a single item. Of course you can define any number of subscriptions and each with any number of items. The subscription defines the default monitoring properties for all its items, but the items may also define individual sampling intervals (see the properties of MonitoredDataItem) that can be set with the setSamplingInterval() method.

In the client, you can then just listen to the change notifications which the server is instructed to send when the item values change according to the subscription properties:

```
item.setDataChangeListener(dataChangeListener);
```

The listener is defined as follows:

```
  protected MonitoredDataItemListener dataChangeListener = new MyMonitoredDataItemListener(this);
```

where

```
/**
 * A sample listener for monitored data changes.
 */
public class MyMonitoredDataItemListener implements MonitoredDataItemListener {
  private final SampleConsoleClient client;

   public MyMonitoredDataItemListener(SampleConsoleClient client) {
    super();
    this.client = client;
  }

  @Override
  public void onDataChange(MonitoredDataItem sender, DataValue prevValue, DataValue value) {
    SampleConsoleClient.println(client.dataValueToString(sender.getNodeId(), sender.getAttributeId(), value));
  }
};
```

You can add a notification listener either to the subscription or to each item. The latter will provide you more detailed information on the data changes, but creating the listener to the subscription can be easier to use and more light-weight. The SampleConsoleClient demonstrates both, but uses mainly the item-based listener.

You may also wish to listen to the alive and timeout events in the subscription. These will help you verify that the server is actively monitoring the values, even in the case that the values are not actually changing and therefore new data change notifications are not being sent. The example below demonstrates how to add such a listener to a subscription:

```
protected SubscriptionAliveListener subscriptionAliveListener = new MySubscriptionAliveListener();
...
subscription.addAliveListener(subscriptionAliveListener);
```

The functionality related to different events needs be implemented in the respective methods of the listener class:

```
/**
 * A sampler listener for subscription alive events.
 */
public class MySubscriptionAliveListener implements SubscriptionAliveListener {

  @Override
  public void onAfterCreate(Subscription s) {
    // the subscription was (re)created to the server
    // this happens if the subscription was timed out during
    // a communication break and had to be recreated after reconnection
    SampleConsoleClient.println(String.format("%tc Subscription created: ID=%d lastAlive=%tc", Calendar
.getInstance(),
        s.getSubscriptionId().getValue(), s.getLastAlive()));
  }

  @Override
  public void onAlive(Subscription s) {
    // the server acknowledged that the connection is alive,
    // although there were no changes to send
    SampleConsoleClient.println(String.format("%tc Subscription alive: ID=%d lastAlive=%tc", Calendar
.getInstance(),
        s.getSubscriptionId().getValue(), s.getLastAlive()));
  }

  @Override
  public void onLifetimeTimeout(Subscription s) {
    SampleConsoleClient.println(String.format("%tc Subscription lifetime ended: ID=%d lastAlive=%tc",
        Calendar.getInstance(), s.getSubscriptionId().getValue(), s.getLastAlive()));

  }

  @Override
  public void onTimeout(Subscription s) {
    // the server did not acknowledge that the connection is alive, and the
    // maxKeepAliveCount has been exceeded
    SampleConsoleClient.println(String.format("%tc Subscription timeout: ID=%d lastAlive=%tc", Calendar
.getInstance(),
        s.getSubscriptionId().getValue(), s.getLastAlive()));

  }

}
```

13. 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 instead of MonitoredDataItem you use MonitoredEventItem. You define an event listener, which gets notified when new events are received from the server. Additionally, you must also define the event filter (using the EventFilter class, which defines the event fields you wish to monitor (with the SelectClauses property of the EventFilter class) and optionally also which events you wish to receive (with the WhereClause property of the EventFilter class).

We have defined the fields as follows:

```
protected final QualifiedName[] eventFieldNames = {
    new QualifiedName("EventType"), new QualifiedName("Message"),
    new QualifiedName("SourceName"), new QualifiedName("Time"),
    new QualifiedName("Severity"), new QualifiedName("ActiveState/Id")
};
```

The current version of the client is actually adding two custom fields in eventFieldNames as well. These are initialized to null and created at run-time, because the QualifiedName-identifiers need a namespaceIndex which depends on the dynamic index in the server. See the code sample for details in initEventFieldNames().

The createBrowsePath() method used in the SampleConsoleClient is a special feature of the sample. It breaks the QualifiedNames, which have '/' delimiters to a browse path, i.e. an array of QualifiedNames.

We can define SelectClauses for the filter respectively:

```
NodeId eventTypeId = Identifiers.BaseEventType;
UnsignedInteger eventAttributeId = Attributes.Value;
String indexRange = null;
SimpleAttributeOperand[] selectClauses =
    new SimpleAttributeOperand[eventFields.length + 1];
for (int i = 0; i < eventFields.length; i++) {
    QualifiedName[] browsePath = createBrowsePath(eventFields[i]);
    selectClauses[i] = new SimpleAttributeOperand(eventTypeId, browsePath, eventAttributeId, indexRange);
}
// Add a field to get the NodeId of the event source
selectClauses[eventFields.length] = new SimpleAttributeOperand(
    eventTypeId, null, Attributes.NodeId, null);
EventFilter filter = new EventFilter();
// Event field selection
filter.setSelectClauses(selectClauses);
```

And next we filter the events we want to receive using the WhereClause:

```
// Event filtering: the following sample creates a
// "Not OfType GeneralModelChangeEventType" filter
ContentFilterBuilder fb = new ContentFilterBuilder();

// The element operand refers to another operand -
// operand #1 in this case which is the next,
// LiteralOperand
fb.add(FilterOperator.Not, new ElementOperand(
   UnsignedInteger.valueOf(1)));
final LiteralOperand filteredType = new LiteralOperand(
   new Variant(Identifiers.GeneralModelChangeEventType));
fb.add(FilterOperator.OfType, filteredType);
filter.setWhereClause(fb.getContentFilter());
```

This one just filters out possible ModelChangeEvents. There are various operators that you can use. Most of them require two arguments, e.g. fb.add(FilterOperator.Equals, operand1, operand2). Note however, that the WhereClause of the filter is optional; if you leave it out, you will get all events under the object Node that you subscribe to.

So, finally we are ready to create the event item using the filter that was just created and the NodeId of the node that should be listened to in the nodeId variable:

```
MonitoredEventItem eventItem = new MonitoredEventItem(nodeId, filter);
eventItem.addEventListener(eventListener);
subscription.addItem(eventItem);
```

The event listener is defined as follows, and used to react to the event notification:

```
 protected final MonitoredEventItemListener eventListener = new MyMonitoredEventItemListener(this,
eventFieldNames);
```

where

```
/**
 * A sampler listener for monitored event notifications.
 */
public class MyMonitoredEventItemListener implements MonitoredEventItemListener {
  private final SampleConsoleClient client;
  private final QualifiedName[] requestedEventFields;

  /**
   * @param client
   * @param eventFieldNames
   */
  public MyMonitoredEventItemListener(SampleConsoleClient client, QualifiedName[] requestedEventFields)
{
    super();
    this.requestedEventFields = requestedEventFields;
    this.client = client;
  }

  @Override
  public void onEvent(MonitoredEventItem sender, Variant[] eventFields) {
    SampleConsoleClient.println(client.eventToString(sender.getNodeId(), requestedEventFields, eventFields)
);
  }
};
```

14. History Access

OPC UA servers may also provide history information for the Nodes, including historical time series data and events. You can read the AccessLevel and UserAccessLevel attributes of a Variable Node to see whether history is available (if yes, then AccessLevel.HistoryRead should be included).

14.1. Reading History

To actually read history data, you have several options. The basic way is to use UaClient.historyRead(), which is recommended if you need to do several readings at once. This example reads a complete history for a single Node (specified by nodeId):

```
HistoryReadDetails details = new ReadRawModifiedDetails(false,
    DateTime.MIN_VALUE, DateTime.currentTime(),
    UnsignedInteger.MAX_VALUE, true);
HistoryReadValueId nodesToRead = new HistoryReadValueId(
    nodeId, null,
    QualifiedName.DEFAULT_BINARY_ENCODING, null);
HistoryReadResult[] result = client.historyRead(details,
    TimestampsToReturn.Both, true, nodesToRead);

HistoryData d = result[0].getHistoryData().decode();
DataValue[] values = d.getDataValues();
```

What you need to be aware of is that there are several “methods” that the historyRead() actually 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).

To make your life a bit easier, UaClient also defines several convenience methods to make specific history requests. For example, the above can also be performed with

```
DataValue[] result = client.historyReadRaw(nodeId,
    DateTime.MIN_VALUE, DateTime.currentTime(),
    UnsignedInteger.MAX_VALUE, true, null, TimestampsToReturn.Source);
```

14.2. Updating or Deleting History

To modify existing history data in the server, you can use the historyUpdate() method or, again, one of the convenience methods that provide you with more semantics. See the documentation for the various historyUpdateXxx() and historyDeleteXxx() methods in UaClient for more about those.

15. 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

```
List<UaMethod> methods = client.getAddressSpace().getMethods(nodeId);
```

UaMethod is a Node object, which gets stored into the NodeCache (see Using Node Objects). If you wish to perform a light browse, you can just call:

```
List<ReferenceDescription> methodRefs = client.getAddressSpace().browseMethods(nodeId);
```

to get a list of the Method References from the Node.

The UaMethod is initialized with the InputArguments and OutputArguments properties, which you can examine for the argument name, type, etc.

```
Argument[] inputArguments = method.getInputArguments();
Argument[] outputArguments = method.getOutputArguments();
```

To actually call the Method, you need to provide a valid value (as Variant) for each of the InputArguments, and just call it:

```
Variant[] outputs = client.call(nodeId, methodId, inputs);
```

As a result you get values for the OutputArguments.

Note also that you will usually need to use the DataTypeConverter to convert the inputArguments to the correct data type, before calling the Method. The OPC UA specification defines that the server may not convert the arguments, if they are provided with incorrect data types. So you will get Bad_InvalidArgument errors for each argument that is not provided in the correct data type. See the sample code (inside SampleConsoleClient.readInputArguments()) for more details.

16. Registering and Unregistering Nodes

These services are meant for improved performance. You can request the server to prepare some Nodes to which you will refer often in your client application by registering them with the RegisterNodes service call. The server may also define new and more efficient NodeIds for the Nodes and the client can then use the new NodeIds instead of the NodeIds it received by browsing the address space.

You can access these from the AddressSpace. To register a Node for quick access, call:

```
NodeId[] registeredNodeId = client.getAddressSpace().registerNodes(nodeId);
```

When you are done, you can unregister the Nodes using, for example:

```
NodeId[] nodes = client.getAddressSpace().unregisterAllNodes();
```

These methods are not usually necessary and not always supported by the servers anyway, so you can usually ignore them. If the server manufacturer suggests, you could consider using them.

17. Using Node Objects

The AddressSpace object in the UaClient can also cache Nodes on the client side. These UaNode objects will help you to browse the address space and to use the information in your application more conveniently.

You can simply request the Node objects from the address space using the methods getNode(), getType(), getMethods(), etc.

To see it in action, just go and explore the sample code in more detail – especially the methods printCurrentNode() and referenceToString().

getNode() is even more useful when used together with code generation. If you register generated classes, you can use complete UA types in your application through the respective Java classes. Read on to learn more about that.

18. Information Modeling and Code Generation

The Prosys OPC UA SDK for Java supports loading existing OPC UA information models in the NodeSet2 XML format (a standard format defined in the OPC UA specification). By default, type definitions from the most important companion specifications are already generated for the client SDK. You can also define your own information models and utilise the type definitions in client applications by importing the XML files. For modeling, you can use the OPC UA Modeler application.

18.1. Prosys OPC UA SDK for Java Code Generator

You can generate Java classes based on information models stored in the NodeSet2 XML format with the help of the Code Generator provided with the Prosys OPC UA SDK for Java. The Code Generator is located in the 'codegen' folder of the distribution package.

For instructions on using the Code Generator provided with the Prosys OPC UA SDK for Java, please refer to the Code Generator Manual in the 'codegen' folder of the distribution package.

Follow the instructions in the included manual and experiment with the samples to learn how to configure and execute the code generation procedure. Then you may return to this tutorial and read the following sections on how to utilize the generated classes in your own applications.

18.2. Using the Generated Classes from Applications

The classes generated with the Code Generator enable simple usage of the type definitions defined in the source information models. With the generated classes, the client application is able to handle the instances of the custom types in OPC UA servers as Java Node objects.

This section explains how the Java classes generated using the Code Generator can be imported to and utilized in applications developed using the Prosys OPC UA Java Client SDK.

18.3. Registering the Model

The generated Java classes for ObjectTypes and VariableTypes are more extended versions of the standard UaNode implementations. In order for the SDK to use the generated classes instead of the basic implementations, the SDK must be made aware of the generated classes. This is called registering the model.

18.4. Using Instances of Generated Types

The client implementation classes are generated into 'client' sub-packages under the defined generation folders.

To use the generated Java classes in your applications:

1. Add the generated files to your project source path.
2. Register generated classes with UaClient.registerModel(CodegenModel model). For example:

   ```
   client.registerModel(example.packagename.client.ClientInformationModel.MODEL);
   ```

   Starting from 4.0.0, you can ignore this, if you have generated support for Automatic Discovery of Generated Models in Code Generator. Please see the Codegen Manual for more information.

3. Read instances with client.getAddressSpace().getNode(NodeId id). You have to cast the result to the correct generated type. Alternatively you can pass a Class parameter of the correct type as additional parameter, e.g.

```
AnalogItemType node = client.getAddressSpace().getNode(<NodeId>, AnalogItemType.class);
```

The SDK distribution package provides a sample information model in the 'SampleTypes.xml' file inside the 'models' folders of both of the Code Generator versions (i.e. command line and Maven). The classes generated based on this model are used in the following example. A complete example of the procedure for using an instance of the ValveObjectType from the 'SampleTypes.xml' information model is also demonstrated in the example below:

```
// 1. Register the generated classes in your UaClient object by
// using the ClientInformationModel class that is generated in the client package.
client.registerModel(example.packagename.client.ClientInformationModel.MODEL);

// 2. Get a node from the server using an AddressSpace object.
// Give the nodeId of the instance as a parameter.
// Cast the return value to the corresponding generated class
ValveObjectType sampleValve =
  (ValveObjectType) client.getAddressSpace().getNode(nodeId);

// 3. Use the instance.
// e.g. get the cached value of the PowerInput Property
sampleValve.getPowerInput();
```

❗ Calling the getter for component values will internally get the node from the AddressSpace and get the value from it. It does not directly make a read call to the server, unless the node is not present in the AddressSpace's NodeCache. You need to use read() methods in the UaClient if you need the latest values.

❗ Calling the setter for component values sets the value to the local node, i.e. it is not written to the server. You need to use write() methods in the UaClient to write the value to the server.

19. Reverse Connections

OPC UA Specification 1.04 defines a new way to open UA TCP connections, called Reverse Connection. In this mode, the server application will open the connection, contrary to the normal connection opened by the client. This can be useful in situations where the server is behind a firewall that cannot let client connections go through to the server.

In order to enable the reverse connection, the client application will first open a TCP/IP socket in a custom port. The Server may then open the connection to the client socket. After that the client will create the UA TCP secure channel and session to the server as usual, including all security details.

In UaClient this can be achieved by defining the socket to be listened with either UaClient.setReversePort() or UaClient.setReverseAddress(). Now, when you call connect(), the client will go waiting for connections from servers. Note that you should not define the connection address to UaClient in this case.

You should also define a listener to validate incoming connections from the server using UaClient.setReverseConnectionListener(). For example:

```
public class MyReverseConnectionListener implements ReverseConnectionListener {

 @Override
 public boolean onConnect(String serverApplicationUri, String endpointUrl, SocketAddress remoteAddress) {
   /*
    * You can perform here validation for reverse connections. Return false, if you wish to stop
    * the connection. Note that the connection is already initiated by the server at the
    * SocketAddress.
    *
    * NOTE! The endpointUrl parameter is sent by the Server and used in the Client in calls forming
    * the higher level communication channel. It is not the actual address the client is connecting
    * (as the socket is already open to the remoteAddress) and most of the time is one of the
    * endpointUrls used in normal non-reverse connections for the Server.
    */
   System.out.println("Accepting reverse connection to server: " + serverApplicationUri + " at: " +
remoteAddress
       + " , using endpointUrl: " + endpointUrl);
   return true;
 }

}
```

At this point in the communication the channel is not encrypted. Therefore when possible you should validate that the server is within the list of known servers by checking the remote SocketAddress of the server. The server ApplicationURI and EndpointUrl are given by the server when they connect. You can use all the security options as in usual connections to validate that the client is communicating with a trusted server.