OPC UA
SDK for Java

Server 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 server application development. With this quick introduction you should be able to grab the basic ideas behind the Prosys OPC UA SDK for Java. You might like to take a look at the Client Tutorial as well, but it is not a requirement.

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 server application in the SampleConsoleServer 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 server.

4. UaServer Object

The UaServer class is the main class you will be working with. It defines a full OPC UA server implementation which you can use in your own applications. Alternatively, you can inherit your own version of the server in case you need to modify the default behaviour or you otherwise prefer to configure your server that way. We will describe how you can simply instantiate the UaServer and define your server functionality by customizing the service managers that perform specific functionality in the server.

We can simply start by creating the server:

```
server = new UaServer();
```

You will find the code in the 'SampleConsoleServer.java' file. Start by locating the main()-method from it and examine the methods called from there.

4.1. 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.

4.1.1. Application Description

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

```
static String APP_NAME = 'SampleConsoleServer';
[...]
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.Server);
```

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.

4.1.2. Application Instance Certificate

You can define the Application Instance Certificate for the client by setting an ApplicationIdentity for the UaServer 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 SampleConsoleServer@hostname_keysize.der and SampleConsoleServer@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 client. The certificate is used by the client to decrypt the token and validate that the server 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.

Note that if some other application gets the same key pair, it can pretend to be the same 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.

4.1.3. 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.

HTTPS protocol may require a CA signed certificate (especially with .NET applications), and therefore it may be necessary to create your own CA key. You will need to provide the CA key to the other applications so that they can verify the Application Instance Certificates signed by this key.

4.1.4. 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);
```

4.1.5. HTTPS Certificate

If you wish to use HTTPS for connecting to the servers, you must also define a separate HTTPS certificate. This is done with:

```
String hostName = InetAddress.getLocalHost().getHostName();
identity.setHttpsCertificate(ApplicationIdentity
    .loadOrCreateHttpsCertificate(appDescription, hostName,
        "opcua", issuerCertificate, privatePath, true));
```

The HTTPS certificate is a little bit different from the Application Instance Certificates, which are used for UA TCP binary transport and application authentication. In HTTPS, the slightly different certificate is needed for the underlying TLS encryption.

Most server applications do not support HTTPS at all and in normal use cases UA TCP is the best alternative regardless.

4.1.6. Assigning the Application Identity

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

```
server.setApplicationIdentity(identity);
```

4.2. Server Endpoints

The server endpoints are the connection points to which the client applications can connect. Each endpoint consists of an URL address and a security settings.

The server defines which endpoints are available and the client decides which of these it will use. The UaClient client implementation in the SDK, for example, will pick the matching endpoint automatically according to desired security settings.

4.2.1. Endpoint URLs

First, we define the endpoint URL(s) using setPort() and setServerName() for the transport protocols we wish to support (OpcTcp and OpcHttps are the currently supported options):

```
// TCP Port number for the UA TCP protocol
server.setPort(Protocol.OpcTcp, 52520);
// TCP Port for the HTTPS protocol
server.setPort(Protocol.OpcHttps, 52443);
server.setServerName("OPCUA/SampleConsoleServer");
```

The properties will define the endpoint URLs of the server as follows

```
<Protocol>://<Hostname>:<Port>/<ServerName>
```

An endpoint URL is always defined using the actual hostname. The ServerName is optional and can be

defined separately for each protocol as well.

4.2.2. BindAddresses

If you need to limit the accessibility of the server to some network interfaces only, you can use the setBindAddresses(). This is the default:

```
// Optionally restrict the InetAddresses to which the server is bound.
// You may also specify the addresses for each Protocol.
// This is the default (isEnableIPv6 defines whether IPv6 address should
// be included in the bound addresses. Note that it requires Java 7 or
// later to work in practice in Windows):
server.setBindAddresses(EndpointUtil.getInetAddresses(server.isEnableIPv6()));
```

The addresses can be defined separately for each protocol.

4.2.3. Security Modes

The server can support different security modes. They can be configured simply like this:

```
/*
 * Define the security modes to support for the Binary protocol.
 *
 * Note that different versions of the specification might add/deprecate some modes, in this
 * example all the modes are added, but you should add some way in your application to configure
 * these. The set is empty by default, you must add at least one SecurityMode for the server to
 * start.
 */
Set<SecurityPolicy> supportedSecurityPolicies = new HashSet<SecurityPolicy>();

/*
 * This policy does not support any security. Should only be used in isolated networks.
 */
supportedSecurityPolicies.add(SecurityPolicy.NONE);

// Modes defined in previous versions of the specification
supportedSecurityPolicies.addAll(SecurityPolicy.ALL_SECURE_101);
supportedSecurityPolicies.addAll(SecurityPolicy.ALL_SECURE_102);
supportedSecurityPolicies.addAll(SecurityPolicy.ALL_SECURE_103);

/*
 * Per the 1.04 specification, only these should be used. However in practice this list only
 * contains very new security policies, which most of the client applications as of today that
 * are used might not be unable to (yet) use.
 */
supportedSecurityPolicies.addAll(SecurityPolicy.ALL_SECURE_104);
```

```
Set<MessageSecurityMode> supportedMessageSecurityModes = new HashSet<MessageSecurityMode>();

/*
 * This mode does not support any security. Should only be used in isolated networks. This is
 * also the only mode, which does not require certificate exchange between the client and server
 * application (when used in conjunction of only ANONYMOUS UserTokenPolicy).
 */
supportedMessageSecurityModes.add(MessageSecurityMode.None);

/*
 * This mode support signing, so it is possible to detect if messages are tampered. Note that
 * they are not encrypted.
 */
supportedMessageSecurityModes.add(MessageSecurityMode.Sign);

/*
 * This mode signs and encrypts the messages. Only this mode is recommended outside of isolated
 * networks.
 */
supportedMessageSecurityModes.add(MessageSecurityMode.SignAndEncrypt);

/*
 * This creates all possible combinations (NONE pairs only with None) of the configured
 * MessageSecurityModes and SecurityPolicies) for opc.tcp communication.
 */
server.getSecurityModes()
    .addAll(SecurityMode.combinations(supportedMessageSecurityModes, supportedSecurityPolicies));
```

4.2.4. HTTPS Security Policies

If you define the port number for HTTPS, you may also define the TLS security policies that are supported. Note that starting from 4.0.0 you must also define which SecurityModes are supported, since application level authentication is now based on the Application Instance Certificates.

```
/*
 *
 * NOTE! The MessageSecurityMode.None for HTTPS means Application level authentication is not
 * used. If used in combination with the UserTokenPolicy ANONYMOUS anyone can access the server
 * (but the traffic is encrypted). HTTPS mode is always encrypted, therefore the given
 * MessageSecurityMode only affects if the UA certificates are exchanged when forming the
 * Session.
 */
server.getHttpsSecurityModes().addAll(SecurityMode
    .combinations(EnumSet.of(MessageSecurityMode.None, MessageSecurityMode.Sign),
supportedSecurityPolicies));

// The TLS security policies to use for HTTPS
Set<HttpsSecurityPolicy> supportedHttpsSecurityPolicies = new HashSet<HttpsSecurityPolicy>();
// (HTTPS was defined starting from OPC UA Specification 1.02)
supportedHttpsSecurityPolicies.addAll(HttpsSecurityPolicy.ALL_102);
supportedHttpsSecurityPolicies.addAll(HttpsSecurityPolicy.ALL_103);
// Only these are recommended by the 1.04 Specification
supportedHttpsSecurityPolicies.addAll(HttpsSecurityPolicy.ALL_104);
server.getHttpsSettings().setHttpsSecurityPolicies(supportedHttpsSecurityPolicies);
```

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 clients support it and use it if you need it. But usually you should do better with the UA TCP protocol, instead.

If you wish to disable HTTPS from your server, you can just use `server.setPort(Protocol.Https, 0)` to undefine the HTTPS port number.

OPC UA security is not used for HTTPS encryption, but it can be for application level authentication. If it is used for that, then the MessageSecurityMode.Sign should be used. Also note that it is dangerous to have a combination of NONE and ANONYMOUS UserTokenPolicy, as it allows anyone to connect to the server (but the traffic is encrypted).

4.2.5. User Security Tokens

You must define one or more user token policies according to the type of tokens you wish to support. For example, to define all three alternatives: anonymous, username and certificate-based user authentication, you would add them all as:

```
server.addUserTokenPolicy(UserTokenPolicy.ANONYMOUS);
server.addUserTokenPolicy(UserTokenPolicy.SECURE_USERNAME_PASSWORD);
server.addUserTokenPolicy(UserTokenPolicy.SECURE_CERTIFICATE);
```

If you support user tokens, you should also implement a UserValidator, for example:

```
server.setUserValidator(userValidator);
```

where

```
private static UserValidator userValidator = new UserValidator() {

    @Override
    public boolean onValidate(Session session, UserIdentity userIdentity) {
        // Return true, if the user is allowed access to the server
        // Note that the UserIdentity can be of different actual types,
        // depending on the selected authentication mode (by the client).
        println("onValidate: userIdentity=" + userIdentity);
        if (userIdentity.getType().equals(UserTokenType.UserName))
            if (userIdentity.getName().equals("opcua")
                    && userIdentity.getPassword().equals("opcua"))
                return true;
            else
                return false;
        if (userIdentity.getType().equals(UserTokenType.Certificate))
            // Implement your strategy here, for example using the
            // PkiFileBasedCertificateValidator
            return true;
        return true;
    }
};
```

4.2.6. Custom Endpoint Configuration

If you wish to omit the automatic generation of endpoints and need more customization options, you can use UaServer.addEndpoint() and removeEndpoint() to modify the list of endpoints more flexibly. This makes it possible to, for example, define different security settings for a certain network interface.

4.2.7. Endpoint Initialization

The UaServer will initialize automatically one endpoint for each endpoint URL and SecurityMode combination during server initialization (see Server Endpoints).

4.3. Validating Client Applications via 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 certificate of OPC UA clients, you need to define a CertificateValidator in the UaServer. This validator is used to validate the certificates received from the clients 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.

If you have used Java SDK 1.x or 2.x, you are familiar with PkiFileBasedCertificateValidator included in the SDK. Since Java SDK 3.0 the classes provided by the Java Stack, namely DefaultCertificateValidator and PkiDirectoryCertificateStore, replace the same functionality with a more flexible design. In 3.x the old validator was deprecated and is now removed in 4.x.

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

```
validator.setValidationListener(validationListener);

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

where the validationListener can be defined according to the example below. This example implementation will accept certificates even though the ApplicationUri does not match the one in the ApplicationDescription, as is done in the SampleConsoleServer:

```
private static CertificateValidationListener validationListener = new CertificateValidationListener() {

    @Override
    public ValidationResult onValidate(Cert certificate,
        ApplicationDescription applicationDescription,
        EnumSet<CertificateCheck> passedChecks) {
        // Do not mind about URI...
        if (passedChecks.containsAll(EnumSet.of(CertificateCheck.Trusted,
            CertificateCheck.Validity, CertificateCheck.Signature))) {
        if (!passedChecks.contains(CertificateCheck.Uri))
            try {
            println("Client's ApplicationURI ("
                + applicationDescription.getApplicationUri()
                + ") does not match the one in certificate: "
                + PkiFileBasedCertificateValidator
                    .getApplicationUriOfCertificate(certificate));
            } catch (CertificateParsingException e) {
            throw new RuntimeException(e);
            }
        return ValidationResult.AcceptPermanently;
        }
        return ValidationResult.Reject;
    }
};
```

4.3.1. Validating HTTPS Certificates

The HTTPS certificates are not used for application authentication. Instead, the SDK enables usage of Application Instance Certificates for that, similar to UA TCP. This requires usage of MessageSecurityMode=Sign to be used.

4.4. Registration to a Discovery Server

4.4.1. Internal Discovery Server

The UaServer implements the DiscoveryService by itself. So you can use the FindServers service from any client application to get a list of servers (or actually just your server) and endpoints that are available from it.

4.4.2. Local Discovery Server

The Local Discovery Server (LDS) is an application provided by the OPC Foundation that specifically keeps a list of servers that are available locally. The clients can query all available servers from the Local Discovery Server.

To get your server listed in the Local Discovery Server, you can define the Discovery Server URL for your server with:

```
// Register to the local discovery server (if present)
server.setDiscoveryServerUrl("opc.tcp://localhost:4840");
```

The standard port number used by the DiscoveryServer is 4840, and you should be able to access it without any server name, i.e. even when the actual URL for the .NET DiscoveryServer is opc.tcp://localhost:4840/UADiscovery.

Alternatively, you can call UaServer.registerServer() yourself.

The discovery server registration is done via a secure channel. The Discovery Server must trust your server before it allows registration. This requires that you copy the application certificate of your application to the certificate store used by the Discovery Server.

If you wish to handle errors in registration, you can do that in a UaServerListener (onRegisterServerError), which you can add to UaServer.

4.5. Server Initialization

Once you have setup the server parameters, call

```
server.init();
```

After this you can proceed with your own customizations, such as defining your data in the address space.

5. Address Space

Although security is crucial, the most important aspect of the server is the address space that defines the data in the server and how it is managed.

The Address Space is managed by NodeManager objects which are used to define the OPC UA Nodes. Nodes are used for all elements of the address space, including Objects, Variables, different types, etc.

5.1. Standard Node Managers

By default, UaServer always contains a base node manager, NodeManagerRoot. It handles the standard OPC UA server address space, i.e. nodes located in the OPC UA standard namespace (namespaceIndex = 0). It includes the root structure of the address space consisting of the main folders (Views, Objects and Types) and the default UA types. It also manages the Server Object which is used to publish server status and diagnostic information to OPC UA clients.

In addition, the UaServer also contains an internal NodeManagerUaServer which manages the server specific diagnostics, as specified by the OPC UA specification (namespaceIndex = 1).

5.2. Your Own Node Managers

In order to be able to add your own nodes into the address space of your server, you must define your own node manager(s) with your own namespace(s). You have a couple of alternatives for choosing which node manager to create.

Instead of defining a single node manager for your data, you can always decide to split your node hierarchy and manage different parts of it with different node managers.

It is a good convention to define types and instances in separate namespaces. The OPC Foundation defines several companion specifications (i.e. different domain-specific information models) and their respective types can be loaded into the server in their respective namespaces. See Information Modeling for more about the information models.

5.2.1. NodeManagerUaNode

Typically, the easiest option to create your own node manager is to use NodeManagerUaNode in which you can create all the nodes as implementations of UaNode objects (or actually subtypes of UaNode, such as UaObject, UaVariable, etc. according to the actual NodeClass of each node).

To create your node manager, you must specify your own NamespaceURI which identifies the namespace of your product. For example:

```
myNodeManager = new NodeManagerUaNode(server,
"http://www.prosysopc.com/OPCUA/SampleAddressSpace");
```

The server will also assign a NamespaceIndex for each namespace. In practice, the index will be used to refer to the namespace more often than the NamespaceURI.

The current SampleConsoleServer application is actually defining a new subclass of NodeManagerUaNode, which it then instantiates. And the rest of the implementation code is in the subclass. See MyNodeManager.createAddressSpace().

5.2.2. Adding Nodes

Next you need to create the node objects and add them to your node manager. Each node is identified by a unique NodeId. It comprises of a NamespaceIndex and the actual identifier. We must use the same NamespaceIndex that our node manager uses also in our NodeIds. So we begin by recording that:

```
int ns = myNodeManager.getNamespaceIndex();
```

Next, we will find the base types and folders that we will use when we add our data nodes to the address space:

```
final UaObject objectsFolder = server.getNodeManagerRoot().getObjectsFolder();
final UaType baseObjectType = server.getNodeManagerRoot().getType(Identifiers.BaseObjectType);
final UaType baseDataVariableType = server.getNodeManagerRoot().getType(Identifiers
.BaseDataVariableType);
```

Now, we are ready to define our nodes. The following example demonstrates how you can create different node types manually in the code. Later on, we will learn how to use information models to initialize types and also how to instantiate objects with complete structures in an easier way.

```
// Folder for my objects
final NodeId myObjectsFolderId = new NodeId(ns, "MyObjectsFolder");
myObjectsFolder = createInstance(FolderTypeNode.class, "MyObjects",     myObjectsFolderId);

this.addNodeAndReference(objectsFolder, myObjectsFolder, Identifiers.Organizes);

// My Device Type

// The preferred way to create types is to use Information Models, but this example shows how
// you can do that also with your own code

final NodeId myDeviceTypeId = new NodeId(ns, "MyDeviceType");
UaObjectType myDeviceType = new UaObjectTypeNode(this, myDeviceTypeId,"MyDeviceType", Locale
.ENGLISH);
this.addNodeAndReference(baseObjectType, myDeviceType, Identifiers.HasSubtype);

// My Device
final NodeId myDeviceId = new NodeId(ns, "MyDevice");
myDevice = new UaObjectNode(this, myDeviceId, "MyDevice", Locale.ENGLISH);
myDevice.setTypeDefinition(myDeviceType);
myObjectsFolder.addReference(myDevice, Identifiers.HasComponent, false);

// My Level Type
final NodeId myLevelTypeId = new NodeId(ns, "MyLevelType");
UaType myLevelType = this.addType(myLevelTypeId, "MyLevelType", baseDataVariableType);

// My Level Measurement
final NodeId myLevelId = new NodeId(ns, "MyLevel");
UaType doubleType = getServer().getNodeManagerRoot().getType(Identifiers.Double);
myLevel = new CacheVariable(this, myLevelId, "MyLevel", LocalizedText.NO_LOCALE);
myLevel.setDataType(doubleType);
myLevel.setTypeDefinition(myLevelType);
myDevice.addComponent(myLevel);
```

We use ns here, to define the NodeIds using our own NamespaceIndex.

Standard node instances (such as myObjectsFolder, which is of type FolderTypeNode) must be created with NodeManagerUaNode.createInstance(), instead of using the constructor against the type (with new).

MyDeviceType and the rest are defined using UaObjectTypeNode, UaObjectNode and CacheVariable. These are the basic building blocks, corresponding to the different OPC UA NodeClasses. Custom nodes such as these can still be instantiated using the new command. In practice, it is advisable to use them only to add single nodes. We are doing it here mostly to show you how to use these basic building blocks.

See Information Modeling for more details on the standard types and about defining your own types.

Note that we are using alternative strategies for defining the OPC UA References for nodes. The basic way is to use NodeManagerUaNode.addNodeAndReference() for that. As the name describes, it will add the node to the node manager and it will also create a Reference from the parent node to the added node.

The more convenient way is to use UaNode.addReference(), which will in fact do the same if you are defining a Hierarchical Reference. If you wish to add a HasComponent or HasProperty Reference, then you can do that directly with UaNode.addComponent() or UaNode.addProperty() respectively.

5.2.3. Custom Node Manager

Instead of using the NodeManagerUaNode, you can declare a custom node manager that is derived from NodeManager. You will need to implement all node handling yourself, but you don't need to instantiate a UaNode object in the memory for every node. This is especially useful if your OPC UA server is just wrapping an existing data store and you do not want to replicate all the data in the memory of the server. Also, if you need to provide access to a large amount of nodes (actual number depending on the amount of memory available), this may be the only option for you.

See MyBigNodeManager for a complete example of such a node manager.

5.3. Node Manager Listener

Instead of creating your own version of the NodeManager (or NodeManagerUaNode), you can simply define your own listener in which you may react to the browse and node management requests from the clients. The listener must implement the NodeManagerListener interface. After creating your listener you need to add it to your node manager:

```
myNodeManager.addListener(myNodeManagerListener);
```

This is a simple example of a listener which just denies node management actions from anonymous users:

```
/**
 * A sample implementation of a NodeManagerListener.
 */
public class MyNodeManagerListener implements NodeManagerListener {
```

```
  @Override
  public void onAfterAddNode(ServiceContext serviceContext, NodeId parentNodeId, UaNode parent, NodeId
nodeId,
      UaNode node, NodeClass nodeClass, QualifiedName browseName, NodeAttributes attributes,
      UaReferenceType referenceType, ExpandedNodeId typeDefinitionId, UaNode typeDefinition) throws
StatusException {
  }

  @Override
  public void onAfterAddReference(ServiceContext serviceContext, NodeId sourceNodeId, UaNode
sourceNode,
      ExpandedNodeId targetNodeId, UaNode targetNode, NodeId referenceTypeId, UaReferenceType
referenceType,
      boolean isForward) throws StatusException {
  }

  @Override
  public void onAfterCreateMonitoredDataItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredDataItem item) {
  }

  @Override
  public void onAfterDeleteMonitoredDataItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredDataItem item) {
  }

  @Override
  public void onAfterModifyMonitoredDataItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredDataItem item) {
  }

  @Override
  public void onAddNode(ServiceContext serviceContext, NodeId parentNodeId, UaNode parent, NodeId
nodeId,
      NodeClass nodeClass, QualifiedName browseName, NodeAttributes attributes, UaReferenceType
referenceType,
      ExpandedNodeId typeDefinitionId, UaNode typeDefinition) throws StatusException {
    // Notification of a node addition request.
    // Note that NodeManagerTable#setNodeManagementEnabled(true) must be
    // called to enable these methods.
    // Anyway, we just check the user access.
    checkUserAccess(serviceContext);
  }

  @Override
  public void onAddReference(ServiceContext serviceContext, NodeId sourceNodeId, UaNode sourceNode,
      ExpandedNodeId targetNodeId, UaNode targetNode, NodeId referenceTypeId, UaReferenceType
referenceType,
```

```
      boolean isForward) throws StatusException {
    // Notification of a reference addition request.
    // Note that NodeManagerTable#setNodeManagementEnabled(true) must be
    // called to enable these methods.
    // Anyway, we just check the user access.
    checkUserAccess(serviceContext);
  }

  @Override
  public boolean onBrowseNode(ServiceContext serviceContext, ViewDescription view, NodeId nodeId,
UaNode node,
      UaReference reference) {
    // Perform custom filtering, for example based on the user
    // doing the browse. The method is called separately for each reference.
    // Default is to return all references for everyone
    return true;
  }

  @Override
  public void onCreateMonitoredDataItem(ServiceContext serviceContext, Subscription subscription, NodeId
nodeId,
      UaNode node, UnsignedInteger attributeId, NumericRange indexRange, MonitoringParameters params,
      MonitoringFilter filter, AggregateFilterResult filterResult, MonitoringMode monitoringMode)
      throws StatusException {
    // Notification of a monitored item creation request

    // You may, for example start to monitor the node from a physical
    // device, only once you get a request for it from a client
  }

  @Override
  public void onDeleteMonitoredDataItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredDataItem monitoredItem) {
    // Notification of a monitored item delete request
  }

  @Override
  public void onDeleteNode(ServiceContext serviceContext, NodeId nodeId, UaNode node, boolean
deleteTargetReferences)
      throws StatusException {
    // Notification of a node deletion request.
    // Note that NodeManagerTable#setNodeManagementEnabled(true) must be
    // called to enable these methods.
    // Anyway, we just check the user access.
    checkUserAccess(serviceContext);
  }

  @Override
```

```
  public void onDeleteReference(ServiceContext serviceContext, NodeId sourceNodeId, UaNode sourceNode,
      ExpandedNodeId targetNodeId, UaNode targetNode, NodeId referenceTypeId, UaReferenceType
referenceType,
      boolean isForward, boolean deleteBidirectional) throws StatusException {
    // Notification of a reference deletion request.
    // Note that NodeManagerTable#setNodeManagementEnabled(true) must be
    // called to enable these methods.
    // Anyway, we just check the user access.
    checkUserAccess(serviceContext);
  }

  @Override
  public void onGetReferences(ServiceContext serviceContext, ViewDescription viewDescription, NodeId
nodeId,
      UaNode node, List<UaReference> references) {
    // Add custom references that are not defined in the nodes here.
    // Useful for non-UaNode-based node managers - or references.
  }

  @Override
  public void onModifyMonitoredDataItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredDataItem item, UaNode node, MonitoringParameters params, MonitoringFilter filter,
      AggregateFilterResult filterResult) {
    // Notification of a monitored item modification request
  }

  private void checkUserAccess(ServiceContext serviceContext) throws StatusException {
    // Do not allow for anonymous users
    if (serviceContext.getSession().getUserIdentity().getType().equals(UserTokenType.Anonymous)) {
      throw new StatusException(StatusCodes.Bad_UserAccessDenied);
    }
  }
}
```

Note that node management is not enabled by default at all. In order to enable it, call NodeManagerTable.setNodeManagementEnabled(true).

5.4. Node Types

The NodeManagerUaNode uses UaNode objects to manage the nodes in the address space. These are simple to use, as you can define the OPC UA Attribute values into the objects and use them in your application to represent the data.

There are various implementations of UaNode as well as other interface types that define the Attributes and Reference requirements according to the respective OPC UA NodeClasses.

5.4.1. Generic Nodes

The generic node types are defined in the Java package com.prosysopc.ua.server.nodes.

All the server-side nodes descend from ServerNode which implements the basis of a node owned by a NodeManager. BaseNode is the actual base implementation which adds a field for each Attribute and a list of References that it keeps in the memory. These are abstract base classes and you cannot instantiate either of them.

The classes that you will typically use are UaObjectNode for representing OPC UA Objects and one of the following for OPC UA Variable and Property nodes respectively:

- CacheVariable & CacheProperty

They offer the same interface for all OPC UA DataTypes and you can easily keep any values in the Variables. You can use CacheVariable.updateValue() to provide new samples to the Variables.

In addition, you can use the following generic types:

- PlainVariable & PlainProperty

With them, you can define the data type used for the value of the Variable or Property, and you can also access the value in a bit simpler way.

For example, if you define

```
mySwitch = new PlainVariable<Boolean>(myNodeManager, mySwitchId, "MySwitch", Locale.ENGLISH);
```

you can set it's value using setCurrentValue():

```
mySwitch.setCurrentValue(false);
```

The Properties are actually defined as special kind of Variables in the OPC UA Specification. Their NodeClass is Variable but their TypeDefinition is always PropertyType. In the SDK, we use the interface UaProperty to enable an alternate way to separate Properties from other Variables. All Java classes of the SDK that represent Properties just inherit from the respective classes representing Variables and additionally implement the UaProperty interface.

5.4.2. Instances

Most OPC UA types are defined with a structure. Objects and Variables are both instances and may contain a specific structure as defined by their respective TypeDefinitions. The Java SDK enables creation of complete instances, including the structure, as Java objects. The creation of instances is based on the NodeBuilder, which is available from NodeManagerUaNode. There is also a convenience method, createInstance(), which can be used to create new instances of known types. For example, you can create a new DataItem with

```
DataItemType node = nodeManager.createInstance(DataItemTypeNode.class, "DataItem");
```

By default, this will create a complete Object or Variable instance including its structure, as defined in the type address space. Only nodes inside the structure with the Mandatory ModellingRule are created by default, but you can configure which Optional nodes should be created using the NodeBuilder directly.

Please see Conditions for one example of configuring Optional nodes.

5.4.3. OPC UA Standard Types

The OPC UA standard defines several types which typically define a specific structure. All of these types are modeled in the SDK and they are available as Java objects as shown in the above examples.

The standard OPC UA types included in the SDK are defined in the Java package com.prosysopc.ua.types.opcua.

This package contains the interface definitions. The server-specific implementations are in the 'server' sub-package and the respective client implementations in the client sub-package.

These interfaces and implementations are based on code generation, which you can also use for your own types as described in the chapter Code Generation.

The SDK does not contain any pre-generated classes for the OPC UA companion specifications by default. If you wish to use them, you can use the Code Generation to create the classes for your own needs.

6. I/O Manager

I/O managers are used to handle read and write calls from the client applications. The abstract base class that defines the interface is IoManager. The default implementation used in the NodeManagerUaNode is IoManagerUaNode. It reads attribute values directly from the UaNode objects of the node manager.

6.1. Nodes as Data Cache

If you use UaNode-based node objects that cache all values in the memory, you do not need to do anything else than provide new values to the objects to make your server work as expected. Examples of such nodes are the above mentioned CacheVariable, CacheProperty, PlainVariable and PlainProperty as well as all non-variable nodes.

6.2. Custom I/O Manager

If you go for the custom node manager, as described in Custom Node Manager, you can also use a custom IoManager implementation. In case you have your data already in a background system and do not wish to replicate the data in the node objects, you can refer read and write calls to the custom I/O manager that can communicate with the background system. Refer to MyBigIoManager for a sample of

such a customized I/O Manager.

If you define your own IoManager implementation, you can assign it to your node manager with:

```
myNodeManager.setIoManager(myIoManager);
```

6.3. I/O Manager Listener

An alternative way to customize the functionalities of an I/O manager is to create your own listener implementing the IoManagerListener interface. It will let you keep the standard IoManager (or IoManagerUaNode), but still provide your own definitions for some of the methods. The listener is assigned to the I/O manager with

```
myNodeManager.getIoManager().setListener(myIoManagerListener);
```

and a sample listener looks like this:

```
/**
 * A sample implementation of an IoManagerListener.
 */
public class MyIoManagerListener implements IoManagerListener {
  private static Logger logger = LoggerFactory.getLogger(MyIoManagerListener.class);

  @Override
  public EnumSet<AccessLevel> onGetUserAccessLevel(ServiceContext serviceContext, NodeId nodeId,
UaVariable node) {
    // The AccessLevel defines the accessibility of the Variable.Value
    // attribute

    // Define anonymous access
    // if (serviceContext.getSession().getUserIdentity().getType()
    // .equals(UserTokenType.Anonymous))
    // return EnumSet.noneOf(AccessLevel.class);
    if (node.getHistorizing()) {
      return EnumSet.of(AccessLevel.CurrentRead, AccessLevel.CurrentWrite, AccessLevel.HistoryRead);
    } else {
      return EnumSet.of(AccessLevel.CurrentRead, AccessLevel.CurrentWrite);
    }
  }

  @Override
  public Boolean onGetUserExecutable(ServiceContext serviceContext, NodeId nodeId, UaMethod node) {
    // Enable execution of all methods that are allowed by default
    return true;
  }
```

```
  @Override
  public EnumSet<WriteAccess> onGetUserWriteMask(ServiceContext serviceContext, NodeId nodeId,
UaNode node) {
    // Enable writing to everything that is allowed by default
    // The WriteMask defines the writable attributes, except for Value,
    // which is controlled by UserAccessLevel (above)

    // The following would deny write access for anonymous users:
    // if
    // (serviceContext.getSession().getUserIdentity().getType().equals(
    // UserTokenType.Anonymous))
    // return EnumSet.noneOf(WriteAccess.class);
    return EnumSet.allOf(WriteAccess.class);
  }

  @Override
  public boolean onReadNonValue(ServiceContext serviceContext, NodeId nodeId, UaNode node,
UnsignedInteger attributeId,
      DataValue dataValue) throws StatusException {
    return false;
  }

  @Override
  public boolean onReadValue(ServiceContext serviceContext, NodeId nodeId, UaValueNode node,
NumericRange indexRange,
      TimestampsToReturn timestampsToReturn, DateTime minTimestamp, DataValue dataValue) throws
StatusException {
    if (logger.isDebugEnabled()) {
      logger.debug("onReadValue: nodeId=" + nodeId + (node != null ? " node=" + node.getBrowseName() : ""));
    }
    return false;
  }

  @Override
  public boolean onWriteNonValue(ServiceContext serviceContext, NodeId nodeId, UaNode node,
UnsignedInteger attributeId,
      DataValue dataValue) throws StatusException {
    return false;
  }

  @Override
  public boolean onWriteValue(ServiceContext serviceContext, NodeId nodeId, UaValueNode node,
NumericRange indexRange,
      DataValue dataValue) throws StatusException {
    logger.info("onWriteValue: nodeId=" + nodeId + (node != null ? " node=" + node.getBrowseName() : "")
        + (indexRange != null ? " indexRange=" + indexRange : "") + " value=" + dataValue);
    return false;
  }
```

```
}
```

UaValueNode is a common interface shared between UaVariable and UaVariableType. The value Attribute can be read from both nodes.

The example above is a bare-bones dummy implementation, but you can define your custom I/O operations in the respective methods, and for read calls, return the results by setting the value of the dataValue argument inside the methods.

Also, you can perform user-specific operations and return user-specific results (e.g. for onGetUserAccessLevel()) by using the ServiceContext parameter, which contains Session information including the UserIdentity of the session.

The method result in write operations indicates if the value was already written to the actual data source. If the operation completes asynchronously, and you do not know that it succeeded yet, you should return false. If the operations fail, you should throw a StatusException, as usual when you need to return an error to the client.

7. Events, Alarms and Conditions

To add support for OPC UA events, you must use an event manager and use event objects to trigger the events.

The event manager actually handles commands related to standard event and condition management.

Conditions are special event types defined as subtypes of ConditionType. Condition Objects typically exist in the server address space whereas all other event types are merely just triggered from the server without any Object instances that would represent them.

To trigger events you must use the respective event nodes.

7.1. Event Manager

The default event manager used by the NodeManagerUaNode is EventManagerUaNode. It handles client commands related to the condition methods, such as enable, disable, acknowledge, etc. See the OPC UA Specification Part 9 for a full description of the condition types and condition methods.

7.1.1. Custom Event Manager

Alternatively, you can replace the event manager of any node manager with your custom version. This enables you to react to the creation, modification and removal of monitored items in client subscriptions.

The event manager is automatically attached to your node manager, if you create it like this:

```
EventManagerUaNode myEventManager = new MyEventManager(myNodeManager);
```

The implementation of a custom event manager is very similar to the implementation of an event manger listener, explained below.

7.1.2. Event Manager Listener

Instead of creating your own event manager, where you react to client actions, you can define an event manager listener that implements the EventManagerListener interface. The event manager listener can be plugged into the event manager as follows:

```
myNodeManager.getEventManager().setListener(myEventManagerListener);
```

where myEventManagerListener is defined as follows:

```
/**
 * A sample implementation of an EventManagerListener.
 */
public class MyEventManagerListener implements EventManagerListener {

  /**
   * Internal counter for UserEventId:s. Used from {@link #getNextUserEventId()}.
   */
  private int eventId = 0;

  @Override
  public boolean onAcknowledge(ServiceContext serviceContext, AcknowledgeableConditionTypeNode
condition,
      ByteString eventId, LocalizedText comment) throws StatusException {
    // Handle acknowledge request to a condition event
    println("Acknowledge: Condition=" + condition + "; EventId=" + eventId + "; Comment=" + comment);
    // If the acknowledged event is no longer active, return an error
    if (!eventId.equals(condition.getEventId())) {
      throw new StatusException(StatusCodes.Bad_EventIdUnknown);
    }
    if (condition.isAcked()) {
      throw new StatusException(StatusCodes.Bad_ConditionBranchAlreadyAcked);
    }

    final DateTime now = DateTime.currentTime();
    condition.setAcked(true, now);
    final ByteString userEventId = getNextUserEventId();
    // addComment triggers a new event
    condition.addComment(eventId, comment, now, userEventId);
    return true; // Handled here
    // NOTE: If you do not handle acknowledge here, and return false,
    // the EventManager (or MethodManager) will call
    // condition.acknowledge, which performs the same actions as this
```

```
    // handler, except for setting Retain
  }

  @Override
  public boolean onAddComment(ServiceContext serviceContext, ConditionTypeNode condition, ByteString
eventId,
      LocalizedText comment) throws StatusException {
    // Handle add command request to a condition event
    println("AddComment: Condition=" + condition + "; Event=" + eventId + "; Comment=" + comment);
    // Only the current eventId can get comments
    if (!eventId.equals(condition.getEventId())) {
      throw new StatusException(StatusCodes.Bad_EventIdUnknown);
    }
    // triggers a new event
    final ByteString userEventId = getNextUserEventId();
    condition.addComment(eventId, comment, DateTime.currentTime(), userEventId);
    return true; // Handled here
    // NOTE: If you do not handle addComment here, and return false,
    // the EventManager (or MethodManager) will call
    // condition.addComment automatically
  }

  @Override
  public void onAfterCreateMonitoredEventItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredEventItem item) {
    //
  }

  @Override
  public void onAfterDeleteMonitoredEventItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredEventItem item) {
    //
  }

  @Override
  public void onAfterModifyMonitoredEventItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredEventItem item) {
    //
  }

  @Override
  public void onConditionRefresh(ServiceContext serviceContext, Subscription subscription) throws
StatusException {
    //
  }

  @Override
  public boolean onConfirm(ServiceContext serviceContext, AcknowledgeableConditionTypeNode condition,
```

```
    ByteString eventId, LocalizedText comment) throws StatusException {
  // Handle confirm request to a condition event
  println("Confirm: Condition=" + condition + "; EventId=" + eventId + "; Comment=" + comment);
  // If the confirmed event is no longer active, return an error
  if (!eventId.equals(condition.getEventId())) {
    throw new StatusException(StatusCodes.Bad_EventIdUnknown);
  }
  if (condition.isConfirmed()) {
    throw new StatusException(StatusCodes.Bad_ConditionBranchAlreadyConfirmed);
  }
  if (!condition.isAcked()) {
    throw new StatusException("Condition can only be confirmed when it is acknowledged.",
        StatusCodes.Bad_InvalidState);
  }
  // If the condition is no longer active, set retain to false, i.e.
  // remove it from the visible alarms
  if (!(condition instanceof AlarmConditionTypeNode) || !((AlarmConditionTypeNode) condition).isActive()) {
    condition.setRetain(false);
  }

  final DateTime now = DateTime.currentTime();
  condition.setConfirmed(true, now);
  final ByteString userEventId = getNextUserEventId();
  // addComment triggers a new event
  condition.addComment(eventId, comment, now, userEventId);
  return true; // Handled here
  // NOTE: If you do not handle Confirm here, and return false,
  // the EventManager (or MethodManager) will call
  // condition.confirm, which performs the same actions as this
  // handler
}

@Override
public void onCreateMonitoredEventItem(ServiceContext serviceContext, NodeId nodeId, EventFilter
eventFilter,
    EventFilterResult filterResult) throws StatusException {
  // Item created
}

@Override
public void onDeleteMonitoredEventItem(ServiceContext serviceContext, Subscription subscription,
    MonitoredEventItem monitoredItem) {
  // Stop monitoring the item?
}

@Override
public boolean onDisable(ServiceContext serviceContext, ConditionTypeNode condition) throws
```

```
StatusException {
    // Handle disable request to a condition
    println("Disable: Condition=" + condition);
    if (condition.isEnabled()) {
      DateTime now = DateTime.currentTime();
      // Setting enabled to false, also sets retain to false
      condition.setEnabled(false, now);
      // notify the clients of the change
      condition.triggerEvent(now, null, getNextUserEventId());
    }
    return true; // Handled here
    // NOTE: If you do not handle disable here, and return false,
    // the EventManager (or MethodManager) will request the
    // condition to handle the call, and it will unset the enabled
    // state, and triggers a new notification event, as here
  }

  @Override
  public boolean onEnable(ServiceContext serviceContext, ConditionTypeNode condition) throws
StatusException {
    // Handle enable request to a condition
    println("Enable: Condition=" + condition);
    if (!condition.isEnabled()) {
      DateTime now = DateTime.currentTime();
      condition.setEnabled(true, now);
      // You should evaluate the condition now, set Retain to true,
      // if necessary and in that case also call triggerEvent
      // condition.setRetain(true);
      // condition.triggerEvent(now, null, getNextUserEventId());
    }
    return true; // Handled here
    // NOTE: If you do not handle enable here, and return false,
    // the EventManager (or MethodManager) will request the
    // condition to handle the call, and it will set the enabled
    // state.

    // You should however set the status of the condition yourself
    // and trigger a new event if necessary
  }

  @Override
  public void onModifyMonitoredEventItem(ServiceContext serviceContext, Subscription subscription,
      MonitoredEventItem monitoredItem, EventFilter eventFilter, EventFilterResult filterResult)
      throws StatusException {
    // Modify event monitoring, when the client modifies a monitored
    // item
  }
```

```
    @Override
    public boolean onOneshotShelve(ServiceContext serviceContext, AlarmConditionTypeNode condition,
        ShelvedStateMachineTypeNode stateMachine) throws StatusException {
      return false;
    }

    @Override
    public boolean onTimedShelve(ServiceContext serviceContext, AlarmConditionTypeNode condition,
        ShelvedStateMachineTypeNode stateMachine, double shelvingTime) throws StatusException {
      return false;
    }

    @Override
    public boolean onUnshelve(ServiceContext serviceContext, AlarmConditionTypeNode condition,
        ShelvedStateMachineTypeNode stateMachine) throws StatusException {
      return false;
    }

    private void println(String string) {
      MyNodeManager.println(string);
    }

    ByteString getNextUserEventId() throws RuntimeException {
      return EventManager.createEventId(eventId++);
    }
  }
```

As you can see, complete event management requires quite a complex implementation. However, you do not need to define your own implementation for all the functionality. Simply return `false` in the methods where you wish the event manager to use its default implementation.

7.2. Defining Events and Conditions

To actually model events in the address space and trigger them, you can use the event types defined in the SDK.

There are two main types of events: normal events and conditions. Events are just notifications to the client applications, whereas conditions can also contain a state. Therefore, condition nodes are typically also available in the address space as nodes.

7.2.1. Normal Events

To define a normal event, you can just create it on the fly. For example (see MyNodeManager.sendEvent()):

```
MyEventType ev = createEvent(MyEventType.class);
```

and define the field values:

```
ev.setMessage("MyEvent");
ev.setMyVariable(new Random().nextInt());
ev.setMyProperty("Property Value " + ev.getMyVariable());
```

Then you can just trigger the event as described below in Triggering Events.

7.2.2. Custom Event Types

If you wish to use custom fields in your events, you will need to define custom event types. The fields are defined as Properties or Variable components of the event type.

You have two alternative ways to define new event types:

- Define the event types in an information model XML (with OPC UA Modeler) and use the Code Generator provided with the SDK (see chapter Code Generation).
- Write the type yourself. Check MyEventType and MyNodeManager.createMyEventType() for the latter. However, using the Code Generator is typically the easier way. See chapter Code Generation for more about the code generation.

7.2.3. Conditions

For example, ExclusiveLevelAlarmType, which is a specific Condition type is used to initialize an alarm node as follows:

```
/**
 * Create a sample alarm node structure.
 *
 * @param source the variable that is the source of the alarm
 *
 * @throws StatusException if something goes wrong in the initialization
 * @throws UaInstantiationException if something goes wrong regarding object instantiation
 */
private void createAlarmNode(UaVariable source) throws StatusException, UaInstantiationException {

  // Level Alarm from the LevelMeasurement

  // See the Spec. Part 9. Appendix B.2 for a similar example

  int ns = this.getNamespaceIndex();
  final NodeId myAlarmId = new NodeId(ns, source.getNodeId().getValue() + ".Alarm");
  String name = source.getBrowseName().getName() + "Alarm";

  // Since the HighHighLimit and others are Optional nodes,
  // we need to define them to be instantiated.
  TypeDefinitionBasedNodeBuilderConfiguration.Builder conf =
TypeDefinitionBasedNodeBuilderConfiguration.builder();
```

```
    conf.addOptional(UaBrowsePath.from(Ids.LimitAlarmType, UaQualifiedName.standard("HighHighLimit")));
    conf.addOptional(UaBrowsePath.from(Ids.LimitAlarmType, UaQualifiedName.standard("HighLimit")));
    conf.addOptional(UaBrowsePath.from(Ids.LimitAlarmType, UaQualifiedName.standard("LowLimit")));
    conf.addOptional(UaBrowsePath.from(Ids.LimitAlarmType, UaQualifiedName.standard("LowLowLimit")));

    // The configuration must be set to be used
    // this.getNodeManagerTable().setNodeBuilderConfiguration(conf.build()); //global
    // this.setNodeBuilderConfiguration(conf.build()); //local to this NodeManager
    // createNodeBuilder(ExclusiveLevelAlarmTypeNode.class, conf.build()); //NodeBuilder specific
    // (createInstance uses this internally)

    // for purpose of this sample program, it is set to this manager, normally this would be set
    // once after creating this NodeManager
    this.setNodeBuilderConfiguration(conf.build());

    myAlarm = createInstance(ExclusiveLevelAlarmTypeNode.class, name, myAlarmId);

    // ConditionSource is the node which has this condition
    myAlarm.setSource(source);
    // Input is the node which has the measurement that generates the alarm
    myAlarm.setInput(source);

    myAlarm.setMessage(new LocalizedText("Level exceeded"));
    myAlarm.setSeverity(500); // Medium level warning
    myAlarm.setHighHighLimit(90.0);
    myAlarm.setHighLimit(70.0);
    myAlarm.setLowLimit(30.0);
    myAlarm.setLowLowLimit(10.0);
    myAlarm.setEnabled(true);
    myDevice.addComponent(myAlarm); // addReference(...Identifiers.HasComponent...)

    // + HasCondition, the SourceNode of the reference should normally
    // correspond to the Source set above
    source.addReference(myAlarm, Identifiers.HasCondition, false);

    // + EventSource, the target of the EventSource is normally the
    // source of the HasCondition reference
    myDevice.addReference(source, Identifiers.HasEventSource, false);

    // + HasNotifier, these are used to link the source of the EventSource
    // up in the address space hierarchy
    myObjectsFolder.addReference(myDevice, Identifiers.HasNotifier, false);
}
```

7.3. Triggering Events

7.3.1. Triggering Normal Events

You must monitor the events in your client application to get notified. When you want to send an event from the server, you can create a new instance and just trigger it:

```
ev.triggerEvent(null);
```

7.3.2. Triggering Conditions

Triggering a condition (or alarm) is basically the same, once you have the event or condition object around and you have modified it's state according to the current situation (see above). Then you can just trigger the event:

```
/**
 * Send an event notification.
 *
 * @param event The event to trigger.
 */
private void triggerEvent(BaseEventTypeNode event) {
  // Trigger event
  final DateTime now = DateTime.currentTime();
  // Use your own EventId to keep track of your events, if you need to (for example when alarms
  // are acknowledged)
  ByteString myEventId = getNextUserEventId();
  // If you wish, you can record the full event ID that is provided by triggerEvent, although your
  // own 'myEventId' is usually enough to keep track of the event.
  /* ByteString fullEventId = */event.triggerEvent(now, now, myEventId);
}
```

myEventId is your own identifier for the event. On the other hand, fullEventId is generated by the SDK and is provided back to you. To extract your custom identifier from it, you can use:

```
ByteString userEventId = EventManager.extractUserEventId(fullEventId);
```

8. Methods

It is simplest to define the Methods for OPC UA Objects in an information model (using the OPC UA Modeler software) and then using it as input for the Code Generator provided with the SDK (see chapter Code Generation). For implementing Methods when using the code generated types, see the section Implementing Methods in Generated Types.

Another option is to define the Methods manually in your application code. See

MyNodeManager.createMethodNode() for an example of this.

8.1. Handling Methods

If you define Method nodes manually in the code, you need to also handle the related Method calls by implementing a method manager or a method manager listener.

A method manager handles incoming Method calls from clients. It dispatches the calls to various locations and returns the result to the client. The implementation of a method manager should be based on the MethodManager class.

Alternatively, you can also utilize a method manager listener that implements the CallableListener interface. The principle is similar to the use of a method manager and is demonstrated in the example below:

```
MethodManagerUaNode m = (MethodManagerUaNode) myNodeManager
    .getMethodManager();
m.addCallListener(myMethodManagerListener);
```

The listener is then implemented as follows:

```
public class MyMethodManagerListener implements CallableListener {

    private static Logger logger = Logger
            .getLogger(MyMethodManagerListener.class);
    final private UaNode myMethod;
    /**
     * @param myMethod
     *            the method node to handle.
     */
    public MyMethodManagerListener(UaNode myMethod) {
        super();
        this.myMethod = myMethod;
    }
        @Override
        public boolean onCall(ServiceContext serviceContext, NodeId objectId,
                UaNode object, NodeId methodId, UaMethod method,
                final Variant[] inputArguments,
                final StatusCode[] inputArgumentResults,
                final DiagnosticInfo[] inputArgumentDiagnosticInfos,
                final Variant[] outputs) throws StatusException {
            // Handle method calls
            // Note that the outputs is already allocated
            if (methodId.equals(myMethod.getNodeId())) {
                logger.info("myMethod: " + Arrays.toString(inputArguments));
                MethodManager.checkInputArguments(new Class[] { String.class,
                    Double.class }, inputArguments, inputArgumentResults,
```

```
            inputArgumentDiagnosticInfos, false);
        String operation;
        try {
            operation = (String) inputArguments[0].getValue();
        } catch (ClassCastException e) {
            throw inputError(0, e.getMessage(),
                inputArgumentResults,
                inputArgumentDiagnosticInfos);
        }
        double input;
        try {
            input = inputArguments[1].intValue();
        } catch (ClassCastException e) {
            throw inputError(1, e.getMessage(),
                inputArgumentResults,
                inputArgumentDiagnosticInfos);
        }
        operation = operation.toLowerCase();
        double result;
        if (operation.equals("sin"))
            result = Math.sin(Math.toRadians(input));
        else if (operation.equals("cos"))
            result = Math.cos(Math.toRadians(input));
        else if (operation.equals("tan"))
            result = Math.tan(Math.toRadians(input));
        else if (operation.equals("pow"))
            result = input * input;
        else
            throw inputError(1, "Unknown function '" + operation
                + "': valid functions are sin, cos, tan, pow",
                inputArgumentResults,
                inputArgumentDiagnosticInfos);
        outputs[0] = new Variant(result);
        return true; // Handled here
    } else
        return false;
}
/**
 * Handle an error in method inputs.
 *
 * @param index
 *          index of the failing input
 * @param message
 *          error message
 * @param inputArgumentResults
 *          the results array to fill in
 * @param inputArgumentDiagnosticInfos
```

```
     *          the diagnostics array to fill in
     * @return StatusException that can be thrown to break further method
     *         handling
     */
    private StatusException inputError(final int index,
            final String message, StatusCode[] inputArgumentResults,
            DiagnosticInfo[] inputArgumentDiagnosticInfos) {
        logger.info("inputError: #" + index + " message=" + message);
        inputArgumentResults[index] = new StatusCode(
                StatusCodes.Bad_InvalidArgument);
        final DiagnosticInfo di = new DiagnosticInfo();
        di.setAdditionalInfo(message);
        inputArgumentDiagnosticInfos[index] = di;
        return new StatusException(StatusCodes.Bad_InvalidArgument);
    }
};
```

This one will only handle one method. In practice you should be prepared to handle all your Methods here.

Instead of the centralised method listener implementation of the previous example, you can also implement the UaCallable interface in your node objects. The MethodManagerUaNode will then call their callMethod() if the listener does not handle the Method call.

9. History Manager

A history manager enables you to handle all historical data and event functionality. There is no default functionality for these in the SDK, so you must keep track of the historical data yourself and implement the services.

Again, you have two different options for implementing history management:

- You can define your own subclass of the HistoryManager class and override the methods that deal with the various history operations. And then just use myNodeManager.getHistoryManager() to set the history manager to your node manager.
- You can simply define a new listener in which you define the functionality.

The following is a sample historian implementation, which relies on memory-based ValueHistory and EventHistory objects that handle the history of each node that is added to the historian:

```
/**
 * A sample implementation of a data historian.
 * <p>
 * It is implemented as a HistoryManagerListener. It could as well be a
 * HistoryManager, instead.
 */
public class MyHistorian implements HistoryManagerListener {
```

```
    private static Logger logger = Logger.getLogger(MyHistorian.class);
    private final Map<UaObjectNode, EventHistory> eventHistories =
new HashMap<UaObjectNode, EventHistory>();
    private final Map<UaVariableNode, ValueHistory> variableHistories =
new HashMap<UaVariableNode, ValueHistory>();

    public MyHistorian() {
        super();
    }

    /**
     * Add the object to the historian for event history.
     * <p>
     * The historian will mark it to contain history (in EventNotifier
     * attribute) and it will start monitoring events for it.
     *
     * @param node
     *            the object to initialize
     */
    public void addEventHistory(UaObjectNode node) {
    EventHistory history = new EventHistory(node);
    // History can be read
    EnumSet<EventNotifierClass> eventNotifier = node.getEventNotifier();
    eventNotifier.add(EventNotifierClass.HistoryRead);
    node.setEventNotifier(eventNotifier);

    eventHistories.put(node, history);
    }

    /**
     * Add the variable to the historian.
     * <p>
     * The historian will mark it to be historized and it will start monitoring
     * value changes for it.
     *
     * @param variable
     *            the variable to initialize
     */
    public void addVariableHistory(UaVariableNode variable) {
    ValueHistory history = new ValueHistory(variable);
    // History is being collected
    variable.setHistorizing(true);
    // History can be read
    final EnumSet<AccessLevel> READ_WRITE_HISTORYREAD = EnumSet.of(
        AccessLevel.CurrentRead, AccessLevel.CurrentWrite,
        AccessLevel.HistoryRead);
    variable.setAccessLevel(READ_WRITE_HISTORYREAD);
```

```
    variableHistories.put(variable, history);
    }

    @Override
    public Object onBeginHistoryRead(ServiceContext serviceContext,
        HistoryReadDetails details, TimestampsToReturn timestampsToReturn,
        HistoryReadValueId[] nodesToRead,
        HistoryContinuationPoint[] continuationPoints,
        HistoryResult[] results) throws ServiceException {
        return null;
    }

    @Override
    public Object onBeginHistoryUpdate(ServiceContext serviceContext,
        HistoryUpdateDetails[] details, HistoryUpdateResult[] results,
        DiagnosticInfo[] diagnosticInfos) throws ServiceException {
        return null;
    }

    @Override
    public void onDeleteAtTimes(ServiceContext serviceContext, Object operationContext,
        NodeId nodeId, UaNode node, DateTime[] reqTimes,
        StatusCode[] operationResults, DiagnosticInfo[] operationDiagnostics)
        throws StatusException {
        ValueHistory history = variableHistories.get(node);
    if (history != null)
        history.deleteAtTimes(reqTimes, operationResults,
            operationDiagnostics);
    else
        throw new StatusException(StatusCodes.Bad_NoData);
    }

    @Override
    public void onDeleteEvents(ServiceContext serviceContext,
Object operationContext,
        NodeId nodeId, UaNode node, ByteString[] eventIds,
        StatusCode[] operationResults, DiagnosticInfo[] operationDiagnostics)
        throws StatusException {
    EventHistory history = eventHistories.get(node);
    if (history != null)
        history.deleteEvents(eventIds, operationResults,
            operationDiagnostics);
    else
        throw new StatusException(StatusCodes.Bad_NoData);
    }

    @Override
    public void onDeleteModified(ServiceContext serviceContext,
```

```
Object operationContext,
        NodeId nodeId, UaNode node, DateTime startTime, DateTime endTime)
        throws StatusException {
        throw new StatusException(
StatusCodes.Bad_HistoryOperationUnsupported);
    }

    @Override
    public void onDeleteRaw(ServiceContext serviceContext,
Object operationContext,
        NodeId nodeId, UaNode node, DateTime startTime, DateTime endTime)
        throws StatusException {
    ValueHistory history = variableHistories.get(node);
    if (history != null)
        history.deleteRaw(startTime, endTime);
    else
        throw new StatusException(StatusCodes.Bad_NoData);
    }

    @Override
    public void onEndHistoryRead(ServiceContext serviceContext,
Object operationContext,
        HistoryReadDetails details, TimestampsToReturn timestampsToReturn,
        HistoryReadValueId[] nodesToRead,
        HistoryContinuationPoint[] continuationPoints,
        HistoryResult[] results) throws ServiceException {
    }

    @Override
    public void onEndHistoryUpdate(ServiceContext serviceContext,
        Object operationContext, HistoryUpdateDetails[] details,
        HistoryUpdateResult[] results, DiagnosticInfo[] diagnosticInfos)
        throws ServiceException {
    }

    @Override
    public Object onReadAtTimes(ServiceContext serviceContext,
Object operationContext,
        TimestampsToReturn timestampsToReturn, NodeId nodeId, UaNode node,
        Object continuationPoint, DateTime[] reqTimes,
        NumericRange indexRange, HistoryData historyData)
        throws StatusException {
    ValueHistory history = variableHistories.get(node);
    if (history != null)
        historyData.setDataValues(history.readAtTimes(reqTimes));
    else
        throw new StatusException(StatusCodes.Bad_NoData);
```

```
    return null;
    }

    @Override
    public Object onReadEvents(ServiceContext serviceContext,
Object operationContext,
        NodeId nodeId, UaNode node, Object continuationPoint,
        DateTime startTime, DateTime endTime,
        UnsignedInteger numValuesPerNode, EventFilter filter,
        HistoryEvent historyEvent) throws StatusException {
    EventHistory history = eventHistories.get(node);
    if (history != null) {
        List<HistoryEventFieldList> events =
new ArrayList<HistoryEventFieldList>();
        int firstIndex = continuationPoint == null ? 0
            : (Integer) continuationPoint;
        Integer newContinuationPoint = history.readEvents(startTime,
            endTime, numValuesPerNode.intValue(), filter, events,
            firstIndex);
        historyEvent.setEvents(events
            .toArray(new HistoryEventFieldList[events.size()]));
        return newContinuationPoint;
    } else
        throw new StatusException(StatusCodes.Bad_NoData);
    }

    @Override
    public Object onReadModified(ServiceContext serviceContext,
Object operationContext,
        TimestampsToReturn timestampsToReturn, NodeId nodeId, UaNode node,
        Object continuationPoint, DateTime startTime, DateTime endTime,
        UnsignedInteger numValuesPerNode, NumericRange indexRange,
        HistoryModifiedData historyData) throws StatusException {
        throw new StatusException(
StatusCodes.Bad_HistoryOperationUnsupported);
    }

    @Override
    public Object onReadProcessed(ServiceContext serviceContext,
        Object operationContext, TimestampsToReturn timestampsToReturn,
        NodeId nodeId, UaNode node, Object continuationPoint,
        DateTime startTime, DateTime endTime, Double resampleInterval,
        NodeId aggregateType,
        AggregateConfiguration aggregateConfiguration,
        NumericRange indexRange, HistoryData historyData)
        throws StatusException {
        throw new StatusException(
```

```
StatusCodes.Bad_HistoryOperationUnsupported);
    }

    @Override
    public Object onReadRaw(ServiceContext serviceContext,
Object operationContext,
        TimestampsToReturn timestampsToReturn, NodeId nodeId, UaNode node,
        Object continuationPoint, DateTime startTime, DateTime endTime,
        UnsignedInteger numValuesPerNode, Boolean returnBounds,
        NumericRange indexRange, HistoryData historyData)
        throws StatusException {
    ValueHistory history = variableHistories.get(node);
    if (history != null) {
        List<DataValue> values = new ArrayList<DataValue>();
        int firstIndex = continuationPoint == null ? 0
            : (Integer) continuationPoint;
Integer newContinuationPoint = history.readRaw(startTime, endTime, numValuesPerNode.intValue(),
returnBounds, firstIndex,   values);
        historyData.setDataValues(values.toArray(new DataValue[values
            .size()]));
        return newContinuationPoint;
    }
    return null;
    }

    @Override
    public void onUpdateData(ServiceContext serviceContext,
Object operationContext,
        NodeId nodeId, UaNode node, DataValue[] updateValues,
        PerformUpdateType performInsertReplace,
        StatusCode[] operationResults, DiagnosticInfo[] operationDiagnostics)
        throws StatusException {
        throw new StatusException(
StatusCodes.Bad_HistoryOperationUnsupported);

}

    @Override
    public void onUpdateEvent(ServiceContext serviceContext,
 Object operationContext,
        NodeId nodeId, UaNode node, Variant[] eventFields,
        EventFilter filter, PerformUpdateType performInsertReplace,
        StatusCode[] operationResults, DiagnosticInfo[] operationDiagnostics)
        throws StatusException {
        throw new StatusException(
StatusCodes.Bad_HistoryOperationUnsupported);
    }
```

```
    @Override
    public void onUpdateStructureData(ServiceContext serviceContext,
        Object operationContext, NodeId nodeId, UaNode node,
        DataValue[] updateValues, PerformUpdateType performUpdateType,
        StatusCode[] operationResults, DiagnosticInfo[] operationDiagnostics)
        throws StatusException {
        throw new StatusException(
StatusCodes.Bad_HistoryOperationUnsupported);
}
};
```

Go and look up the implementations for ValueHistory and EventHistory from the sample code to find out the actual algorithms. As you see, the Modified data and updates are not implemented yet, either. This implementation also requires the use of UaNode objects.

You must then add the listener to the history manager of your node manager:

```
myNodeManager.getHistoryManager().setListener(myHistoryManagerListener);
```

10. Start Up

Once you have initialized your server, you simply need to start it:

```
server.start();
```

11. Shutdown

Once you are ready to close the server, call shutdown() to notify the clients before the server actually closes down:

```
server.shutdown(5, new LocalizedText("Closed by user", Locale.ENGLISH));
```

The first argument for the method defines the delay (in seconds) until the server shuts down after notifying the clients. The second argument defines the reason for the shutdown.

12. MyBigNodeManager

The UaNode-based approach to the implementation of an OPC UA server is very good as long as you do not need to manage a huge number of data nodes. Also, if your data is already in an existing subsystem, it may not feel very reasonable to replicate it all using the UaNodes. In this case, you can

implement a custom node manager (and other managers) which handles the service requests from the OPC UA clients and provides the requested data. Prosys OPC UA SDK for Java enables this by allowing you to override the necessary methods in the managers with your own code.

The SampleConsoleServer includes a sample of such a custom node manager that demonstrates the basic capabilities. We will go through the necessary aspects of this quite comprehensive example in this chapter, trying to explain the steps you need to take with your own implementations.

12.1. Your Node Manager

You start by creating a new class which inherits from NodeManager. You will need a constructor that prepares your node manager and also a connection to your actual underlying data. Our sample is constructed like this:

```
public MyBigNodeManager(UaServer server, String namespaceUri, int nofItems) {
    super(server, namespaceUri);
    DataItemType = new ExpandedNodeId(null, getNamespaceIndex(),
        "DataItemType");
    DataItemFolder = new ExpandedNodeId(null, getNamespaceIndex(),
        "MyBigNodeManager");
    try {
        getNodeManagerTable()
            .getNodeManagerRoot()
            .getObjectsFolder()
            .addReference(getNamespaceTable().toNodeId(DataItemFolder),
                Identifiers.Organizes, false);
    } catch (ServiceResultException e) {
        throw new RuntimeException(e);
    }
    dataItems = new HashMap<String, MyBigNodeManager.DataItem>(nofItems);
    for (int i = 0; i < nofItems; i++)
        addDataItem(String.format("DataItem_%04d", i));

    myBigIoManager = new MyBigIoManager(this);
}
```

It takes the server and namespaceUri as parameters like all node managers do. In addition, we have a parameter for defining how many data items we initialize in our sample class.

After that, we prepare the nodes that we use by just defining the NodeIds for these. Except for the data, we also define lightweight DataItem objects. You must figure out the best way to map your data to the server with some custom DataItem objects.

12.2. Browse Support

To support the OPC UA Browse service, you must implement a few abstract methods. NodeManager includes a default implementation for the service, which just requires you to provide the necessary

information from your system. The necessary methods are

```
protected abstract QualifiedName getBrowseName(ExpandedNodeId nodeId, final UaNode node);
protected abstract LocalizedText getDisplayName(ExpandedNodeId nodeId, UaNode targetNode, Locale
locale);
protected abstract NodeClass getNodeClass(ExpandedNodeId nodeId, UaNode node);
protected abstract UaReference[] getReferences(NodeId nodeId, UaNode node);
protected abstract ExpandedNodeId getTypeDefinition(ExpandedNodeId nodeId, UaNode node);
```

getReference() is the key method: for every node in your address space, you must define the OPC UA References it has. And remember, the References are typically bidirectional: in every child node there is also an inverse Reference to the parent, for example.

```
@Override
protected UaReference[] getReferences(NodeId nodeId, UaNode node) {
   try {
      // Define reference to our type
      if (nodeId.equals(getNamespaceTable().toNodeId(DataItemType)))
         return new UaReference[] { new MyReference(new ExpandedNodeId(
            Identifiers.BaseDataVariableType), DataItemType,
            Identifiers.HasSubtype) };
      // Define reference from and to our Folder for the DataItems
      if (nodeId.equals(getNamespaceTable().toNodeId(DataItemFolder))) {
         UaReference[] folderItems = new UaReference[dataItems.size() + 2];
         // Inverse reference to the ObjectsFolder
         folderItems[0] = new MyReference(new ExpandedNodeId(
            Identifiers.ObjectsFolder), DataItemFolder,
            Identifiers.Organizes);
         // Type definition reference
         folderItems[1] = new MyReference(DataItemFolder,
            getTypeDefinition(
               getNamespaceTable().toExpandedNodeId(nodeId),
               node), Identifiers.HasTypeDefinition);
      int i = 2;
   // Reference to all items in the folder
      for (DataItem d : dataItems.values()) {
         folderItems[i] = new MyReference(DataItemFolder,
            new ExpandedNodeId(null, getNamespaceIndex(),
               d.getName()), Identifiers.HasComponent);
         i++;
      }
      return folderItems;
      }
   } catch (ServiceResultException e) {
      throw new RuntimeException(e);
   }

   // Define references from our DataItems
   DataItem dataItem = getDataItem(nodeId);
   if (dataItem == null)
      return null;
   final ExpandedNodeId dataItemId = new ExpandedNodeId(null,
      getNamespaceIndex(), dataItem.getName());
   return new UaReference[] {
      new MyReference(DataItemFolder, dataItemId,
         Identifiers.HasComponent),
      new MyReference(dataItemId, DataItemType,
         Identifiers.HasTypeDefinition) };
}
```

The References are defined using an implementation of the UaReference interface. The custom implementation is called MyReference. In principle, for a given Reference it must define the ExpandedNodeId for the source node (sourceId) and the ExpandedNodeId for the target node (targetId). The direction of the reference is always from the source to the target. If you include References for which "this node" is the Target, you are in practice defining an inverse reference. The rest of the methods are rather straightforward, for example:

```
@Override
protected NodeClass getNodeClass(ExpandedNodeId nodeId, UaNode node) {
    if (getNamespaceTable().nodeIdEquals(nodeId, DataItemType))
        return NodeClass.VariableType;
    if (getNamespaceTable().nodeIdEquals(nodeId, DataItemFolder))
        return NodeClass.Object;
    // All data items are variables
    return NodeClass.Variable;
}
```

12.3. NodeId & ExpandedNodeId

As you can see, all the methods take nodeId and node as arguments. The latter is always null in our case, since we are not supporting UaNodes in this implementation. So everything we do must be based on nodeIds – either of type NodeId or ExpandedNodeId. These are typically intercompatible, but you must be sure which one you are using and also note that the ExpandedNodeId has two flavors in practice. It can be defined using a NamespaceIndex or a NamespaceUri.

Due to these reasons checking for equality between a NodeId and an ExpandedNodeId is not always that simple: if you use NodeId.equals() with an ExpandedNodeId or ExpandedNodeId.equals() with either, you easily get unwanted results. The best option is to compare them with getNamespaceTable().nodeIdEquals(), which can check against `both the NamespaceIndex and the NamespaceUri.

You can convert between NodeId and ExpandedNodeId best with getNamespaceTable().toNodeId() and getNamespaceTable().toExpandedNodeId().

In practice, it is best to define ExpandedNodeId objects with NamespaceIndices instead of NamespaceUris to keep them better compatible with the NodeIds inside your node manager.

12.4. MyBigIoManager

Next you need to define the I/O manager which handles the Attribute services, i.e. Read and Write calls to nodes. You can start by defining a class that extends the IoManager class. You can then either override the readAttribute() and writeAttribute() methods or the readValue() and readNonValue() as well as the writeValue() and writeNonValue() methods. Our sample defines the MyBigIoManager class, which overrides readValue() and readNonValue() only – it does not support writing.

The difference between the Value versus the other OPC UA Attributes is mainly that the Value typically also has a StatusCode and a SourceTimestamp related to it. The other Attributes just have the actual value of the Attribute. Nevertheless, all read and write methods use DataValue structures to carry the complete values.

In our case, readValue() is simple, because we know that it's only available for our DataItems (which are Variables):

```
@Override
protected void readValue(ServiceContext serviceContext, NodeId nodeId,
        UaVariable node, NumericRange indexRange,
        TimestampsToReturn timestampsToReturn, DateTime minTimestamp,
        DataValue dataValue) throws StatusException {
    DataItem dataItem = getDataItem(nodeId);
    if (dataItem == null)
        throw new StatusException(StatusCodes.Bad_NodeIdInvalid);
    dataItem.getDataValue(dataValue);
}
```

Implementation of the readNonValue() method is a bit more complicated:

```
@Override
protected void readNonValue(ServiceContext serviceContext,
        NodeId nodeId, UaNode node, UnsignedInteger attributeId,
        DataValue dataValue) throws StatusException {
    Object value = null;
    UnsignedInteger status = StatusCodes.Bad_AttributeIdInvalid;
    DataItem dataItem = getDataItem(nodeId);
    final ExpandedNodeId expandedNodeId = getNamespaceTable()
            .toExpandedNodeId(nodeId);
    if (attributeId.equals(Attributes.NodeId))
        value = nodeId;
    else if (attributeId.equals(Attributes.BrowseName))
        value = getBrowseName(expandedNodeId, node);
    else if (attributeId.equals(Attributes.DisplayName))
        value = getDisplayName(expandedNodeId, node, null);
    else if (attributeId.equals(Attributes.Description))
        status = StatusCodes.Bad_OutOfService;
    else if (attributeId.equals(Attributes.NodeClass))
        value = getNodeClass(expandedNodeId, node);
    else if (attributeId.equals(Attributes.WriteMask))
        value = UnsignedInteger.ZERO;
    // the following are only requested for the DataItems
    else if (dataItem != null) {
        if (attributeId.equals(Attributes.DataType))
            value = Identifiers.Double;
        else if (attributeId.equals(Attributes.ValueRank))
            value = ValueRanks.Scalar;
        else if (attributeId.equals(Attributes.ArrayDimensions))
            status = StatusCodes.Bad_OutOfService;
        else if (attributeId.equals(Attributes.AccessLevel))
            value = AccessLevel.getMask(AccessLevel.READONLY);
        else if (attributeId.equals(Attributes.Historizing))
            value = false;
    }
    // and this is only requested for the folder
    else if (attributeId.equals(Attributes.EventNotifier))
        value = EventNotifierClass.getMask(EventNotifierClass.NONE);
    if (value == null)
        dataValue.setStatusCode(status);
    else
        dataValue.setValue(new Variant(value));
    dataValue.setServerTimestamp(DateTime.currentTime());
}
```

Since many of the Attributes can also be accessed using the Browse service, we can simply utilize the existing methods from our node manager to provide the responses. And since we do not support

writing, all nodes can be treated the same: WriteMask = 0 for all, for example.

12.5. Subscriptions and MonitoredDataItems

The last part in defining a complete data access server is providing data change notifications to the clients. This requires that you manage the MonitoredItems yourself and call notifyDataChange() for them. It will check the value against the deadband, filter and DataChangeTrigger of the item to see if the client really wants to see that change. The sample node manager MyBigNodeManager overrides afterCreateMonitoredDataItem() and deleteMonitoredItem() to keep track of which DataItems are being monitored. And whenever the values are changed (by simulate()), the clients are also notified:

```
private void notifyMonitoredDataItems(DataItem dataItem) {
   // Get the list of items watching dataItem
   Collection<MonitoredDataItem> c = monitoredItems
      .get(dataItem.getName());
   if (c != null)
      for (MonitoredDataItem item : c) {
         DataValue dataValue = new DataValue();
         dataItem.getDataValue(dataValue);
         item.notifyDataChange(dataValue);
      }
}
```

12.6. MonitoredEventItems

Events are monitored via MonitoredEventItems. In principle, the system is equal to monitoring the DataItems, but you must track the item creations with afterCreateMonitoredEventItem() (in a node manager or an event manager). And when you are ready to trigger an event, you must call MonitoredEventItem.notifyEvent() to send it to the client. For notifyEvent() you will need an EventData structure, which defines the values of all condition fields. You should refer to the OPC Foundation specification for that or take a look at the respective node implementations in the SDK.

13. Information Modeling

To utilize the information modeling capabilities of OPC UA in your applications, you have to first create an information model. The SDK supports loading information models in the NodeSet2 XML format. The best tool for creating the models is the OPC UA Modeler. You will need to purchase a license for it in order to generate the XML files needed for the Java Code Generator.

13.1. Loading Information Models

You can load an information model (for example “SampleTypes.xml”) to the server with

```
server.getAddressSpace().loadModel(new File("SampleTypes.xml").toURI());
```

This will add all types and instances defined in the XML file to the address space.

The SDK ships with the standard OPC UA information model, which is always initialized into the NodeManagerRoot. See SampleConsoleServer.loadInformationModels() for examples on loading the companion specifications.

13.2. Code Generation

In addition to just loading the types and instances from an XML, you may wish to use the types in your Java applications. For this purpose, you can generate Java classes according to the type definitions.

13.2.1. 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 dedicated code generation 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.

13.2.2. 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.

13.2.3. Loading the Model

In addition to registering the Java classes to be used on the server side, the type Nodes in the information model need to be loaded to the address space of the server. The SDK reads these nodes from the address space when instantiating a type. It should be noted that the registration of the model must be done before loading the model. Otherwise, the SDK will use the default implementation for the nodes instead of the generated ones.

If you load the model first and register only after that, you will get ClassCastExceptions. The SDK does not check for registration, since it also works without the generated classes by using the default implementations of the nodes.

13.2.4. Using Instances of Generated Types

The server implementation classes are generated in 'server' sub-packages under the defined generation folders.

If your information models contain any methods, you will need to implement these as well in the implementation nodes. Check your generated source (refresh the project in Eclipse first, for example) and see if it gives any errors for these.

To use the nodes then in your applications:

1. Add the generation target directories to your project source path.
2. Register generated classes with UaServer.registerModel(CodegenModel model). You can use the generated server-side InformationModel class for the registration, for example server.registerModel(example.packagename.server.ServerInformationModel.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. Load the information model NodeSet2 XML with server.getAddressSpace().loadModel(URI path).
4. Create instances with NodeManagerUaNode.createInstance(Class class, String name).
5. Write method implementations. If your types define methods, the generated implementations will throw a Bad_NotImplemented StatusException by default. You must write the actual implementation to the generated implementation ('impl') classes or write an implementation for the method interface and register it to the generated base class.

The SDK distribution package provides a sample information model in the 'SampleTypes.xml' file inside the 'models' folders of both of the the Code Generator versions (i.e. command line and Maven). The classes generated based on this model are used in the following examples. A complete example of the procedure for creating 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 UaServer object by
// using the ServerInformationModel class that is generated in the server package.
server.registerModel(example.packagename.server.ServerInformationModel.MODEL);

// 2. Load the type nodes from the SampleTypes.xml file.
server.getAddressSpace().loadModel(
   new File("SampleTypes.xml").toURI());

// 3. Now you can create an instance of ValveObjectType by using the NodeManagerUaNode:
ValveObjectType sampleValve =
  manager.createInstance(ValveObjectTypeNode.class, "SampleValve");

// 4. Use the instance.
// e.g., set the value of the PowerInput Property.
sampleValve.setPowerInput(160.5);
```

See also section Instances for examples on how to create instances with optional nodes.

13.2.5. Implementing Methods in Generated Types

There are two different approaches to implementing the OPC UA Method functionality into the classes generated by the Code Generator:

- Writing method handlers inside the generated classes
- Providing outside implementations through static methods

For the first option, the Code Generator creates a method handler for each OPC UA Method in an Object in the generated implementation class. You need to find the method handler and write your own implementation in there.

An example of a method handler is presented from the ValveObjectTypeNode class:

```
@Override
protected void onChangeState(ServiceContext serviceContext, ValveStateDataType newState) throws
    StatusException {
  //Implement the generated method here (and remove the code below) OR set implementation via static
method setChangeStateMethodImplementation
  throw new StatusException(StatusCodes.Bad_NotImplemented);
}
```

For the second option, the Code Generator also provides a static method in the generated base classes that allows for setting the implementation for an OPC UA Method. For example, setChangeStateMethodImplementation() in the base class ValveObjectTypeNodeBase can be used to provide an implementation for the same Method as in the previous example. The provided implementation must implement the generated interface for the Method, such as ValveObjectTypeChangeStateMethod for the example Method.

14. 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 open a reverse connection with UaServer, you can use UaServer.addReverseConnection(String clientServerEndpointUrl, String endpointUrlForClientConnection). To remove previously added reverse connections you can use UaServer.removeReverseConnection(String clientServerEndpointUrl, String endpointUrlForClientConnection).

Reverse connections can be added while the server is running, however in order to remove them the server must be shut down (see removeReverseConnection javadocs for more information).

The clientServerEndpointUrl must be in the EndpointUrl format, for example opc.tcp://client_hostname_or_ip:port. The endpointUrlForClientConnection must be a suitable EndpointUrl, which the client will need to finalize the connection. After UaServer.init(), assuming you have only bound single network interface, the following should find a suitable one:

```
String endpointUrlForClientConnection = null;
for (EndpointDescription ed : uaServer.getEndpoints()) {
  if (ed.getEndpointUrl().startsWith("opc.tcp")) {
    if (endpointUrlForClientConnection == null) {
      endpointUrlForClientConnection = ed.getEndpointUrl();
    } else {
      if (!endpointUrlForClientConnection.equals(ed.getEndpointUrl())) {
        logger.warn("Found more than one EndpointUrl for opc.tcp, using one of them");
      }
    }
  }
}
```