Prosys OPC UA Java SDK

Server SDK Tutorial

Hello world!

Welcome to the Prosys OPC UA Java SDK Tutorial for Server software development. With this quick introduction you should be able to grab the basic ideas behind the Java UA SDK. 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 the OPC UA communications, although you can get to the start without much prior knowledge.

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

Contents

1. Installation

See the installation instructions in README.txt (and at the download site). There are also notes about the usage and deployment of external libraries used by the SDK.

2. Sample Applications

The SDK contains a sample server application, `SampleConsoleServer`. This tutorial will refer to that code while explaining the different steps to take in order to accomplish the main tasks of an OPC UA server.

3. 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 application. Alternatively, you can inherit your own version of the server; if you need to modify the default behavior 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:[1]

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

3.1 Application Identity

In every case, all applications must define an application instance certificate, which is used to validate that the other application we are communicating with, is the one that we trust. The servers will only accept connections from clients, to which they have granted access.

All OPC UA applications must also define some characteristics of themselves. This information is communicated to other applications via the OPC UA protocol, when the applications are connected. By this means, you can validate and/or audit the applications with which your application is communicating with.

3.1.1 Application Description

The characteristics of the OPCUA applications are defined in the following structure:

```
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(applicationName +
    "@localhost"));
appDescription.setApplicationUri("urn:localhost:OPCUA:" + applicationName);
appDescription.setProductUri("urn:prosysopc.com:OPCUA:" + applicationName);
appDescription.setApplicationType(ApplicationType.Server);
```

ApplicationUri is a unique identifier for each running instance. Therefore, it is usually defined with ‘localhost’, which will be replaced with the actual host name.

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.

[1] You will find the code in SampleConsoleServer.java: locate the `main`-method from it – and see the methods called from there.

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 warning with your application.

3.1.2 Application Instance Certificate

You can define the Application Instance Certificate[2] for the server with the `ApplicationIdentity` property of `UaServer` The simplest way to do this is:

```
final ApplicationIdentity identity = ApplicationIdentity
     .loadOrCreateCertificate(appDescription, "Sample Organisation",
     /* Private Key Password */"opcua",
     /* Key File Path */new File(validator.getBaseDir(), "private"),
     /* Enable renewing the certificate */true);
```

Here you can see a sample of creating a self-signed certificate using the service of `ApplicationIdentity.loadOrCreateCertificate`. On the first run, it creates a certificate and a private key and stores them in files `SampleConsoleServer.der` and `SampleConsoleServer.key`, respectively (`appDescription.getApplicationName()=`*`APP_NAME`* `{="SampleConsoleServer"}` defines the file names). The private key is used by the server, to create a secret token sent to the client. The certificate is used by the client to decrypt the token and validate that the server created it.[3]

The fourth parameter in `loadOrCreateCertificate` simply defines the path where the certificate files are stored. Do not mind about it at the moment: it is clarified later…

The fifth parameter enables automatic certification renewal, when it gets out-dated.

3.1.3 Issuer Certificate

In a perfect world, the certificates are signed by a recognized Certificate Authority, instead of using the self-signed keys as above. For the UA applications, it is usually best to use the UA Configuration Tool available from the OPC Foundation.

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

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

which creates a certificate key pair with a private key password ("opcua") for "ProsysSampleCA" for 10 years (3650 days). The key pair is stored in the privatePath (which referes to the PKI directory of the validator, as above).

---

[2] 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 aparameter for loadOrCreateCertificate, which extracts the ApplicationUri from it.

[3] 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, 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. **Note**: As from version 1.3 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. Before 1.3, the SDK (and Java stack) stored the private keys in *.key* files, which are plain binary files.

3.1.4 Multiple Application Instance Certificates

The new stack (v1.02) adds support for a new security profile, Basic256Sha256. It can only be used with big certificates (2048 to 4096 bits). This may require that the applications define two certificates, if they wish to continue using 1024 bit certificates (which are not compatible with this new profile) or wish to use 4096 bit certificates (which are only usable with the new profile).

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);
```

3.1.5 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 to the Application Instance certificates, which are used for secure OPC UA communication. In HTTPS, those are not used at all, and both authentication of applications and encryption of communications is based on the HTTPS certificates only.

3.1.6 Application Identity

Now, we can just assign the identity to the Server:

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

In addition, you can add *Software Certificates* that your application has received from the OPC UA certification process[4] to the `ApplicationIdentity`. These are used to validate your application's conformance to the OPC UA protocol, to the client applications it is communicating with.

3.2 Server Endpoints

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

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

[4] OPC Certification Process Web site, http://www.opcfoundation.org/Certification.aspx

3.2.1 Endpoint URLs

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

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

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

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

An endpoint URL is always defined using the actual host name. The ServerName can be defined separately for each protocol as well.

3.2.2 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.

3.2.3 Security Modes

The server can support different security modes. Simply like this:

```
// Define the security modes to support - ALL is the default
server.setSecurityModes(SecurityMode.ALL);
```

And `SecurityMode.ALL` is defined as follows:

```
// Security Mode Sets
// The 101-modes are the default for the time being, until all stacks add support
for BASIC256SHA256
public final static SecurityMode[] ALL_102 = new SecurityMode[] {NONE,
BASIC128RSA15_SIGN_ENCRYPT, BASIC128RSA15_SIGN, BASIC256_SIGN_ENCRYPT,
BASIC256_SIGN, BASIC256SHA256_SIGN_ENCRYPT, BASIC256SHA256_SIGN};

public final static SecurityMode[] ALL_101 = new SecurityMode[] {NONE,
BASIC128RSA15_SIGN_ENCRYPT, BASIC128RSA15_SIGN, BASIC256_SIGN_ENCRYPT,
BASIC256_SIGN};

public final static SecurityMode[] ALL = ALL_101;
```

3.2.4 HTTPS Security Policies

If you define the port number for HTTPS, you may also define the TLS security policies that are supported. By default the server is initialized to support ALL available policies:

```
// The TLS security policies to use for HTTPS
server.getHttpsSettings().setHttpsSecurityPolicies(HttpsSecurityPolicy.ALL)
```

The default policies (included in ALL) are TLS 1.0 and TLS 1.1 (there is a problem with the current JREs, which makes TLS 1.2 not to work with the Java stack).

The *CertificateValidator* and *HttpsHostnameVerifier* provide an option to verify the server certificate. The latter is just defining if a standard validation is done against the host name of the server. By default, no checking is made, but you can define stricter rules with, for example these

```
server.getHttpsSettings().setHttpsCertificateValidator(yourValidator);
server.getHttpsSettings().setHttpsHostnameVerifier(org.apache.http.conn.ssl
.SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

3.2.5 User Security Tokens

You must define at least one user token policy, according to the type of tokens you wish to support. For example, to define all three alternatives, anonymous and user name and user 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;
    }
};
```

3.2.6 Custom endpoint configuration

If you wish to omit the automatic generation of endpoints and need more customization options, i.e. you wish to define insecure connections only for a certain IP address, you can use `UaServer.addEndpoint()` and `removeEndpoint()` to define the list of endpoints and the security settings for each, individually.

3.2.7 Endpoint initialization

The UaServer will initialize automatically one endpoint for each endpointUrl and each SecurityMode combination, at server initialization (see 3.5).

3.3 Validating Client Applications via Certificates

To validate the client applications that are connecting to your server, you should use a CertificateValidator. The SDK comes with a default PKI file based validator, which you can add to your server:

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

The `validationListener` enables you to decide what to do with untrusted certificates. By default these are rejected and moved to the `PKI/CA/rejected` directory. You can then enable the client applications by moving the certificates to the `PKI/CA/certs` directory.

For example, the following is used to omit an invalid applicationUri parameter in the certificates.

```
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;
     }
     };
```

3.4 Registration to a DiscoveryServer

3.4.1 Internal DiscoveryServer

The UA Server 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.

3.4.2 External DiscoveryServer

But the real DiscoveryServer (often called the Local Discovery Server, when running in localhost) is an application that specifically keeps a list of servers that are available locally or via network connection. The clients can then query all available servers from the Discovery Server. There is no generic Java implementation of such a server available yet, but you can use the DiscoveryServer provided by the OPC Foundation (implemented in .NET).

To get your server listed by the external DiscoveryServer, you can define the DiscoveryServerUrl in your server[5]:

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

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.

3.5 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!

4. Address Space

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

4.1 Standard Node Managers

By default, UaServer always contains a base Node Manager, `NodeManagerRoot`. It handles the standard OPC UA Server address space, i.e. the OPCUA Namespace (namespaceIndex=0). This defines the root structure of the address space, which constitutes 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 UA Clients.

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

4.2 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 `NodeManager`, with your own namespace. You have a couple of alternatives for choosing, which node manager to create.

Note that you can always decide to split your node hierarchy and manage different parts of it with different node managers.

[5] 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.

4.2.1 NodeManagerUaNode

Typically, the easiest option is to create a `NodeManagerUaNode`, in which you can create all the nodes as implementations of `UaNode` objects (or actually as `UaObject`, `UaVariable`, etc. according to the actual `NodeClass` of each).

To create your node manager, you must specify your own namespace URL, which identifies the namespace of your product. For example:[6]

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

4.2.2 Adding Nodes

Next you need to create the node objects and add them to your node manager. We must use the same `NamespaceIndex` as 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 need, 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 (note that we use 'ns' here, to define the node IDs using our own namespaceIndex):

```
// 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
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);
```

[6] The current `SampleConsoleServer.java` is actually defining a new subclass for NodeManagerUaNode, which it instantiates. And the rest of the code is in this class. See `MyNodeManager.createAddressSpace()`.

```
// 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);
```

Standard node objects (such as the FolderTypeNode) must be created with NodeManagerUaNode.createInstance() or a NodeBuilder, instead of using the 'new' keyword as in SDK 1.x. MyDeviceType and the others are defined using the UaObjectTypeNode, UaObjectNode and CacheVariable. These can still be used with the 'new' - and are used to construct the standard node structures in behind as well. See 13. 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 references: The original way is to use `NodeManager.addNodeAndReference()` for that. As the name describes, it will add the node to the node manager, and also it will create a reference from a parent node to this 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, you can do that directly with `UaNode.addComponent()` or `UaNode.addProperty()`, respectively.

4.2.3 Custom Node Manager

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

See section 12. *MyBigNodeManager* for a complete example of such a node manager.

4.3 NodeManagerListener

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.

Simply create your listener and add it to your node manager:

```
myNodeManager.addListener(myNodeManageListener);
```

This is a simple example of a listener, which just denies node management actions from anonymous users[7]:

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

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

	@Override
	public void onAddNode(ServiceContext serviceContext, NodeId parentNodeId,
			UaNode parent, NodeId nodeId, UaNode node, 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 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 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, UaNode node,
			UnsignedInteger attributeId, NumericRange indexRange,
			MonitoringParameters params, MonitoringFilter filter,
```

```
                AggregateFilterResult filterResult) 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);
        }
};
```

4.4 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 attribute values into the objects and use them in your application to represent the data.

There are various implementations of UaNode and the other interface types that define the attributes and reference requirements according to the respective OPC UA NodeClasses.

4.4.1 Generic nodes

The generic node types are defined in the Java namespace `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, which it keeps in memory. These are abstract base classes, and you cannot instantiate neither of them.

The types that you will typically use are `UaObjectNode` for creating objects and one of the following variable and property nodes[8]:

- `CacheVariable` & `CacheProperty`

These offer the same interface for all data types and you can easily keep any values in the variables. You can use `CacheVariable.updateValue()` to provide new samples to the variables.

- `PlainVariable` & `PlainProperty`

These are generic types, i.e. you can define the data type used for the value of the variable, 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 CurrentValue:

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

4.4.2 Instances

Most OPC UA types are defined with a structure. The objects and variables are both *instances* and may contain a specific structure as defined by the respective TypeDefinition.

The Java SDK 2.0 enables creation of complete instances, including the structure as the respective 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");
```

[8] Note that properties are actually defined as special kind of variables in the OPCUA 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. So all property nodes of the SDK just inherit from the respective variable type and additionally implement the UaProperty interface.

By default this will create a complete object or variable instance including a structure, as defined in the type address space. Only the Mandatory nodes are created for the nodes by default, but you can configure which optional nodes should be created using the NodeBuilder directly. For example:

```
// Configure the optional nodes using a NodeBuilderConfiguration
NodeBuilderConfiguration conf = new NodeBuilderConfiguration();

// You can use NodeIds to define Optional nodes (good for standard UA
// nodes as they always have namespace index of 0)
conf.addOptional(Identifiers.AnalogItemType_EngineeringUnits);

// You can also use ExpandedNodeIds with NamespaceUris if you don't know
// the namespace index.
conf.addOptional(new ExpandedNodeId(NamespaceTable.OPCUA_NAMESPACE,
      Identifiers.AnalogItemType_InstrumentRange.getValue()));

// You can also use the BrowsePath from the type if you like (the type's
// BrowseName is not included in the path, so this config will apply to
// any type which has the same path)
// You can use Strings for 0 namespace index, QualifiedNames for 1-step
// paths and BrowsePaths for full paths
// Each type interface has constants for it's structure (1-step deep)
conf.addOptional(AnalogItemType.DEFINITION);

// Use the NodeBuilder to create the node
final AnalogItemType node = nodeManager
      .createNodeBuilder(AnalogItemType.class, conf)
      .setName(dataTypeName + "AnalogItem").build();
```

The objects can now be used simply as

```
node.setDefinition("Sample AnalogItem");
node.setEngineeringUnits(new EUInformation("http://www.example.com", 3,
      new LocalizedText("kg", LocalizedText.NO_LOCALE),
      new LocalizedText("kilogram", Locale.ENGLISH)));
node.setEuRange(new Range(0.0, 1000.0));
node.setValue(new DataValue(new Variant(0.0), StatusCode.GOOD,
      DateTime.currentTime(), DateTime.currentTime()));
```

4.4.3 OPCUA 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 UA node types, included in the SDK are defined in the Java namespace `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' package.

These interfaces and implementations are based on the Code Generation, which you can also use for your own types as described in chapter 13.

The SDK contains also generated classes for the companion standards, DI, ADI and PLCOpen. Whereas the standard types have also some internal logic written in them, these companion implementations do not yet contain any specific code. If you need that, it may be best to generate your own versions and modify those.

5. I/O Manager

I/O managers are used to handle read and write calls from the client applications. The default implementation of `NodeManagerUaNode` is `IoManagerUaNode`, respectively. It will read attribute values directly from the `UaNode` objects of the node manager.

5.1 Nodes As Data Cache

If you use UaNodes that cache all values in memory, you do not need to do anything else than provide new values to the variables, to make your server work as expected. Examples of such nodes are the above mentioned `CacheVariable`, `CacheProperty`, `PlainVariable` & `PlainProperty`, as well as all non-variable nodes, which keep all attribute values in memory.

5.2 Custom I/O Manager

If you go for the custom node manager, as described in 4.2.3, and you have the data already in a background system, which you do not wish to replicate to UaNodes, you must also use a custom IoManager implementation. Refer to 12.4 for a sample of such a customized I/O Manager.

5.3 IoManagerListener

If you wish to perform more customized readings than what the default I/O manager does, the first option is that you implement your own `IoManager` and assign it to your node manager:

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

Alternatively, you can use an `IoManagerListener`, to keep the standard IoManager, but provide your own definitions for some of the methods:

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

The listener is defined as follows:

```
public class MyIoManagerListener implements IoManagerListener {
    @Override
    public EnumSet<AccessLevel> onGetUserAccessLevel(
        ServiceContext serviceContext, NodeId nodeId, UaVariable node) {
        // The AccessLevel defines the accessibility of the Variable.Value
        // attribute
        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)
        return EnumSet.allOf(WriteAccess.class);
    }

    @Override
    public void onReadNonValue(ServiceContext serviceContext,
        NodeId nodeId, UaNode node, UnsignedInteger attributeId,
        DataValue dataValue) throws StatusException {
```

```
            // OK
        }

        @Override
        public void onReadValue(ServiceContext serviceContext, NodeId nodeId,
            UaValueNode node, NumericRange indexRange,
            TimestampsToReturn timestampsToReturn, DateTime minTimestamp,
            DataValue dataValue) throws StatusException {
            // OK
        }

        @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 {
            return false;
        }
    }; [9]
```

this one is quite dummy, but you can define your custom I/O operations in the respective methods, and return the results by setting the `DataValue` parameters.

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

The method result in write operations defines, whether the value was written already 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.

6. Events, Alarms and Conditions

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

The event manager actually handles commands related to standard event and condition management[10].

To trigger events you must use the respective event nodes.

6.1 Event Manager

The default event manager used by `NodeManagerUaNode` is an `EventManagerUaNode`. It handles client commands, related to the condition methods, such as enable, disable, acknowledge, etc.

[9] UaValueNode is a common interface shared between UaVariable and UaVariableType; Value can be read both node types.

[10] Conditions are special event types, defined as sub types of ConditionType. Condition objects typically exist in the address space, where as all other event types are merely just triggered from the server without any object nodes that would represent them.

See the OPC UA Specification Part 9 for a full description of the condition types and condition methods.

6.1.1 Custom Event Manager

`NodeManagerUaNode` uses a default implementation of the `EventManagerUaNode`, but you can also replace it with your custom version, if you wish full control over event management. By defining your own event manager, you can react to monitored items (for events) being created, modified or removed from client subscriptions.

The EventManager is automatically attached to your NodeManager, if you create it like this:

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

The implementation of a custom event manager is very similar to the custom `EventManagerListener`, explained below.

6.1.2 EventManagerListener

Instead of creating your own event manager, where you react to client actions, you can define an `EventManagerListener`, which you plug into the event manager.

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

where `myEventManagerListener` is of the following type:

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

    private int eventId = 0;

    @Override
    public boolean onAcknowledge(ServiceContext serviceContext,
            AcknowledgeableConditionTypeNode condition, byte[] eventId,
            LocalizedText comment) throws StatusException {
        // Handle acknowledge request to a condition event
        println("Acknowledge: Condition=" + condition + "; EventId="
                + eventIdToString(eventId) + "; Comment=" + comment);
        // If the acknowledged event is no longer active, return an error
        if (!Arrays.equals(eventId, 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 byte[] 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, byte[] eventId,
            LocalizedText comment)
```

```
            throws StatusException {
        // Handle add command request to a condition event
        println("AddComment: Condition=" + condition + "; Event="
                + eventIdToString(eventId) + "; Comment=" + comment);
        // Only the current eventId can get comments
        if (!Arrays.equals(eventId, condition.getEventId()))
            throw new StatusException(StatusCodes.Bad_EventIdUnknown);
        // triggers a new event
        final byte[] 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, byte[] eventId,
            LocalizedText comment) throws StatusException {
        // Handle confirm request to a condition event
        println("Confirm: Condition=" + condition + "; EventId="
                + eventIdToString(eventId) + "; Comment=" + comment);
        // If the confirmed event is no longer active, return an error
        if (!Arrays.equals(eventId, 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 byte[] 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 String eventIdToString(byte[] eventId) {
		return eventId == null ? "(null)" : Arrays.toString(eventId);
	}

	/**
	 * @param string
	 */
	private void println(String string) {
		MyNodeManager.println(string);
	}

	/**
	 * @return
	 * @throws RuntimeException
	 */
	byte[] getNextUserEventId() throws RuntimeException {
		return EventManager.createEventId(eventId++);
	}
}
```

As you can see there is quite much in complete event management. If you do not define your own implementation, simply return `false` in the methods to make the event manager use its default implementation.

6.2 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.

6.2.1 Normal events

To define a normal event, you can just create it on the fly. For example:

```
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 0.

6.2.2 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.

In SDK 2.x you have two alternative ways to define new event types: 1. to define the event types in XML (with UaModeler) and use the code generator; 2. to write the type yourself. Check MyEventType and MyNodeManager.createMyEventType() for the latter. However, the code generator is typically the easier one. See chapter 13. for more about the code generation.

6.2.3 Conditions

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

```
// 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";
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("Level exceeded"); // Default locale
myAlarm.setMessage("Füllständalarm!", Locale.GERMAN);
myAlarm.setSeverity(500); // Medium level warning
myAlarm.setHighHighLimit(90.0);
myAlarm.setHighLimit(70.0);
myAlarm.setLowLowLimit(10.0);
myAlarm.setLowLimit(30.0);
myAlarm.setEnabled(true);
myDevice.addComponent(myAlarm);

// + 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);
```

6.3 Triggering Events

6.3.1 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:

```
// If the type has TypeDefinitionId, you can use the class
MyEventType ev = createEvent(MyEventType.class);
ev.setMessage("MyEvent");
ev.setMyVariable(new Random().nextInt());
ev.setMyProperty("Property Value " + ev.getMyVariable());
ev.triggerEvent(null);
```

6.3.2 Triggering conditions

Triggering a condition (or alarm) is basically the same, but the condition objects are not created on the fly, since they usually exist in the address space and keep the state in them. Nevertheless, you should first update the state of it:[11]

```
myAlarm.setActive(true);
myAlarm.setRetain(true);
myAlarm.setAcked(false); // Also sets confirmed to false
myAlarm.setSeverity(severity);
```

and then you can trigger it

```
// Trigger event
final DateTime now = DateTime.currentTime();
byte[] myEventId = getNextUserEventId();
byte[] fullEventId = event.triggerEvent(now, now, myEventId);
```

`myEventId` is your own identifier for the event. `fullEventId` is generated by the SDK and is provided back to you in the Command interface (above). To extract your custom identifier from it, you can use:

```
final byte[] userEventId = EventManager.extractUserEventId(fullEventId);
```

7. Method Manager

The method manager handles incoming method calls from clients. It dispatches the calls to various locations and returns the result to the client.

7.1 Declaring Methods

The methods are also easiest to define for the objects in the UA Nodeset files (and UaModeler) and then generating them with the code generator. The SampleTypes.xml includes a sample method as well.

Another option is to define the methods manually. See `MyNodeManager.createMethodNode()` for an example of this.

[11] See `SampleConsoleServer.activateAlarm()`.

7.2 Handling Methods

For generated methods, the code generator will also generate a method handler for each method. For example (in the generated ValveTypeNode.java):

```
@Override
protected void onChangeState(ServiceContext serviceContext,
     ValveState newState) throws StatusException {
     //TODO: Implement the generated method
     throw new StatusException(StatusCodes.Bad_NotImplemented);
}
```

So you need to find the method handler and write your own implementation in there.

The alternative way, if you define the method nodes in code is to handle the method with the MethodManager or MethodManagerListener:

```
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, you can also implement the `UaCallable` interface in your node objects: the `MethodManagerUaNode` will call their `callMethod()`, if the listener does not handle the call.

8. History Manager

The 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 options for defining the implementation. 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 manager to your NodeManager. Or 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, byte[][] 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 `EventHistory` and `ValueHistory` 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 `HistoryManager` of your `NodeManager`:

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

9. FileNodeManager

OPC UA 1.02 defines a new type, FileType, which used to enable File Transfer over OPC UA.

The SDK contains two ready to use classes for that: FileNodeManager and FileSyncClient.

FileNodeManager is capable of monitoring disk folders and providing the respective FileType nodes in the server address space, so that client applications can read the files using the methods defined in the FileType.

FileSyncClient, on the other hand, defines a client object, which can be used to synchronize these files between the client and the server. It works with the FileNodeManager, but may also be able to work with other servers that provide FileType objects, although it hasn't been tried with any other one.

Initializing the FileNodeManager is simple:

```
fileNodeManager = new FileNodeManager(getServer(),
     "http://prosysopc.com/OPCUA/FileTransfer", "Files");
getServer()
     .getNodeManagerRoot()
     .getObjectsFolder()
     .addReference(fileNodeManager.getRootFolder(),
     Identifiers.Organizes, false);
fileNodeManager.addFolder("Folder");
```

Here we define that the root folder of the file node manager is "Files" and it is added to the Objects folder. Then we add a new folder to monitor in "Folder", which is interpreted as a relative path in respect to the server application. Use an absolute path, fi that is better for your application.

The monitored folders are defined as instances of `FileFolderType`, which is a custom implementation of FolderType and allows you to add files to it manually. But the FileNodeManager will actually monitor the files in the defined folders automatically, as defined by the `monitoringInterval` and keep the folders up to date.

10. Start up

Once you have initialized the 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)
```

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 with UaNodes. In this case, you can implement a custom NodeManager (and other managers), which manages the service requests from the OPC UA clients and provides the requested data. Prosys OPC UA Java SDK enables this by overriding the necessary methods in the managers with your own code.

The `SampleConsoleServer` includes a sample of such a custom node manager, demonstrating the basic capabilities. We will go through the necessary aspects in this document, trying to explain the steps you need to take with your own implementation.

12.1 Your NodeManager

You start of course by creating a new class which inherits from `NodeManager`. You will need a constructor, which can prepare your node manager and also a connection to your actual 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 a parameter as all NodeManagers 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: we just define 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 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);
```

`getReferences` is the key method: for every node in your address space, you must define the references it has. And remember, the references are typically bidirectional: in every “subnode” 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 with implementation of the `UaReference` interface. We use a custom implementation of that, `MyReference`. In principle, it must define the `SourceId` and `TargetId` (as `ExpandedNodeId`). 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 straight forward, 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 a `nodeId` and `node` as parameters. The latter is always null in our case, since we are not supporting UaNodes. So everything we do must be based on nodeIds – either of type `NodeId` or `ExpandedNodeId`. These are typically compatible, 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 namespaceUri.

So checking for equality between these, is not always that simple: if you use `NodeId.equals()` (with an ExpandedNodeId) or `ExpandedNodeId.equals()` (with anything) you easily get unwanted results. The best option is to compare them with `getNamespaceTable().nodeIdEquals()`, which can check against the namespaceIndex or namespaceUri.

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

Of course, in practice, it is best to define ExpandedNodeId objects also with namespaceIndexes, instead of namespaceUris, to keep them better compatible with NodeIds inside your NodeManager.

12.4 MyBigIoManager

Next you need to define the IoManager, which handles the attribute services, i.e. Read and Write calls. You can either override the `readAttribute()` and `writeAttribute()` methods or both `readValue()` and `readNonValue()` as well as `writeValue()` and `writeNonValue()`. Our sample defines `MyBigIoManager`, which overrides readValue and readNonValue only – it does not support writing.

The difference in the Value and other attributes is mainly that the Value typically also has a `StatusCode` and `SourceTimestamp` related to it. The other attributes just have the value. Nevertheless, all read and write methods use `DataValue` structures to carry the complete values.

In our case, readValue is simple, as 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);
}
```

readNonValue 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 are also provided in the browse results, we can simply use the same methods from our NodeManager 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 NodeManager overrides `afterCreateMonitoredDataItem()` and `deleteMonitoredItem()` to keep track 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 `NodeManager` or `EventManager`). 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 use information modeling capabilities of OPC UA, you have to first create an information model. The SDK supports loading information models in NodeSet2.xml-format.

The best tool for creating the models is **UaModeler**. You will need to purchase a license for it, to be able to generate the XML files for the Java code generator.

13.1 Loading Information Models

You can load an information model (for example “source.NodeSet2.xml”) to the server with

```
server.getNodeManagerTable.loadModel(new
    File("source.NodeSet2.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`, and with several companion models. See `SampleConsoleServer.loadInformationModels()` for examples of those. They are not loaded by default.

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 application. For this purpose, you can generate Java classes according to the type definitions.

The SDK includes a code generator in the codegen-folder. Follow the instructions in Readme.md and try the sample as described in ReadmeSample.md to learn how to use it.

13.3 Creating instances on UA server

Once you have generated the classes, you can register the InformationModel to your application and also load the respective type definitions and you are ready to go:

```
// 1. Register classes on your UaServer object.
// The Information model class is generated in the server package.
server.registerModel(InformationModel.MODEL);

// 2. Load type nodes from your imaginary source.NodeSet2.xml file.
server.getNodeManagerTable.loadModel(
    new File("source.NodeSet2.xml").toURI());
```

```
// 3. Now you can create an instance of an imaginary GeneratedType
// with a NodeManagerUaNode:
GeneratedType generatedInstance =
       manager.createInstance(GeneratedType.class, "GeneratedInstance");

// 4. Use the instance.
// e.g., set the value of an imaginary GeneratedProperty.
generatedInstance.setGeneratedPropertyValue(1.23);
```

See also 4.4.2 for examples on how to create the instances with optional nodes.