

Proslys OPC UA Java SDK

Migration from SDK 1.x to 2.0

This document describes the changes in the SDK design between version 1.x and 2.0. You can use this document to help you to migrate your applications built with SDK 1.x to use the new SDK version 2.0.

Contents

1. Installation	3
2. Deployment	3
3. Stack improvements	3
3.1 HTTPS Protocol	3
3.2 Bind Addresses	3
3.3 Basic256Sha256 and Big Certificates	3
3.4 Security Libraries	4
3.4.1 Bouncy Castle	4
3.4.2 Spongy Castle	4
3.4.3 Sun JCE	4
3.4.4 Unrestricted Policy Files required for 256 bit encryption!	4
5. SDK changes	4
5.1 ApplicationIdentity and multiple certificates	4
5.2 UaClient object	5
5.3 UaServer	5
5.3.1 HTTPS	5
5.3.2 Basic256Sha256 SecurityMode	6
5.4 NodeManager and server-side UaNodes	6
5.4.1 Node instantiation	6
5.4.2 Java classes for the nodes	6
5.5 IoManager	7
5.6 EventType and EventManager	7
5.7 HistoryManager	7
6. Code generation and nodes	7

1. Installation

The SDK comes with a new version of the OPC Foundation Java Stack 1.02. It uses some new libraries, which you will also need to use in your application, to access the respective new features. These libraries are the Apache HTTP Components (`http*.jar`), Apache Commons Logging (`commons-logging-*.jar`, used by `http-client*.jar`) and Spongy Castle (`sc*.jar`, the Android version of Bouncy Castle).

2. Deployment

The libraries are now optional, except for the log4j library (`log4j-1.2.17.jar`). The HTTP Components (`http*.jar`) are only necessary, if your application supports HTTPS. If Bouncy Castle security libraries (`bc*.jar`) are not used, the SunJCE libraries, which are included in Java SE are used instead. In Android, Spongy Castle (`sc*.jar`) must be used to support any security features. See more below (3.4).

So, in principle, if you wish to continue with your old application, you can just do it – and only need to update Bouncy Castle libraries to the new version, 1.47, in addition to the `Opc.Ua.Stack-1.02.*.jar`.

3. Stack improvements

SDK 2.0 will now use the OPC Foundation stack 1.02, instead of 1.01. This enables several new features.

3.1 HTTPS Protocol

The new stack enables the HTTPS protocol. It exposes endpoints that use the *'https-uabinary'* transport. In practice, UA binary encoded messages are delivered via HTTP requests and responses. Security is provided by standard SSL/TLS security protocols. The level of security can be selected between TLS 1.0, TLS 1.1 (recommended) and TLS 1.2. Each application defines which TLS versions it supports and the applications negotiate a common encryption cipher accordingly on connection.

3.2 Bind Addresses

The server SDK 1.x used to define properties for initialization of different endpoints using various hostnames and IP address names. The new stack enables binding each endpoint to certain network interfaces via specific `InetAddresses`. This enables a better way to configure which endpoint is available to which network segments and it is no longer necessary to define various `endpointUrls` for that task. Therefore, the respective properties (`UseLocalHost`, `UseAllIpAddresses`) of `UaServer` have been deprecated.

3.3 Basic256Sha256 and Big Certificates

The new stack 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).

Note: Since the other stacks of OPC Foundation (.NET & AnsiC) do not support this feature, yet, the `Basic256Sha256` security profile is not enabled by default. But the `SampleConsoleServer` has been updated to initialize certificates of size 1024 and 4096 bits.

3.4 Security Libraries

The security library usage has been changed to enable different libraries to be used. This is not fully flexible yet, but you have a couple of alternatives available. The stack will pick the used library automatically depending on which one it finds from the class path.

3.4.1 Bouncy Castle

Bouncy Castle is still the primary option in standard Java environments. The stack is now using version 1.47. This version includes two jar-files (bcprov & bcpkix) instead of one as in 1.46 (which was used by SDK 1.x). To keep on using Bouncy Castle, you just need to keep that in the class path of your application.

3.4.2 Spongy Castle

Spongy Castle is the full version of Bouncy Castle for Android. Android typically has a limited version of Bouncy Castle available by default, but it does not provide all the functionality that is necessary for the OPC UA stack. Spongy Castle consists of three necessary jar files. You should link your Android project against these, instead of the Bouncy castle libraries.

3.4.3 Sun JCE

The stack can also use now the Sun JCE classes for all security related features. This has not been tested as much as the Bouncy Castle support, so keep that in mind if you take it in use. If you don't include Bouncy Castle (or Spongy Castle) in your application's class path, the Sun JCE classes will be used automatically.

Note that the Sun classes are only available with the Standard Java JVM. Only Oracle JVMs have been tested in general.

3.4.4 Unrestricted Policy Files required for 256 bit encryption!

4. The stack is now using the standard Java Cryptography Extension (JCE) framework for enabling the selection of security library that is used (via SecurityProvider). The main consequence is that since the standard security policy in the JCE allows a maximum of 128-bit encryption algorithms, you must install the "JCE Unlimited Strength Jurisdiction Policy Files" into the JRE to enable 256-bit security. See the release notes for 2.0.0 for the links – or look them up from the Oracle web site for the version of your JRE. Unfortunately, these may not be delivered with the stack or SDK.

5. SDK changes

This section describes the major changes in the SDK. In addition, several details have changed, also affecting some interfaces and method signatures (throws clauses mostly). Check the Release Notes for a more complete list of changes.

5.1 ApplicationIdentity and multiple certificates

The `ApplicationIdentity.loadOrCreateCertificate()` has now an overload version, which can take an array of `keySizes` as well as an `issuerKey` for signing the certificates.

Since the endpoint related properties are deprecated in the server, the application is typically bound to just one host name. Therefore, the `HostNames` argument to `loadOrCreateCertificate` is not normally necessary any more and this works fine (without `issuerKeys`):

```
final ApplicationIdentity identity = ApplicationIdentity
    .loadOrCreateCertificate(appDescription, "Sample Organisation",
        /* Private Key Password *//"opcua",
        /* Key File Path */privatePath,
        /* Issuer Certificate & Private Key */null,
```

```

/* Key Sizes for instance certificates to create */keySizes,
/* Enable renewing the certificate */true);

```

If HTTPS is to be used a `HttpsCertificate` must be assigned to `ApplicationIdentity` as well, e.g.:

```

identity.setHttpsCertificate(
    ApplicationIdentity.loadOrCreateHttpsCertificate(
        appDescription, hostName, "opcua",
        issuerKeys, privatePath, true));

```

Both the client and server applications can use the same identity definitions. Unless you need to use big certificates, you won't need to change to that, though.

5.2 UaClient object

The `UaApplication.Protocol` enumeration has a new option, `Https`, which can be used with `UaClient.setProtocol()`. You can also define an URI of mode "https://...". Also the old `Opc` value has been renamed to `OpcTcp`.

The new client side property for configuring the HTTPS connection is:

```
UaClient.getHttpsSettings()
```

The following properties of the settings are used to define the security policies and certificate validation

```

.get/setHttpsSecurityPolicies()
.get/setCertificateValidator()
.get/setHostnameVerifier()

```

The default policies are TLS 1.0 and TLS 1.1 (there is a problem with the current JRE 7, which makes TLS 1.2 not to work with the Java stack). By default, all certificates and all host names are accepted.

In addition, `UaClient` enables browsing endpoints by Protocol with

```
UaClient.discoverEndpoints(Protocol...)
```

And you can now use a discovered endpoint to define the connection with

```
UaClient.setEndpoint(EndpointDescription)
```

5.3 UaServer

5.3.1 HTTPS

To enable HTTPS on the server side, you only need to define

```
UaServer.setPort(Protocol.Https, <portnr>)
```

Using a non-zero port number will add an HTTPS endpoint to the server. You can use the new `BindAddresses` to limit the availability of the each protocol.

```
UaServer.setBindAddresses(<protocol>, <inetAddresses>)
```

By default, the HTTPS endpoint will use the same `ServerName` as the binary endpoints. (UA) `SecurityMode` is always `None` for HTTPS, but the enabled TLS security policies are defined with

```

UaServer.getHttpsSettings()
UaServer.setHttpsSecurityPolicies(<securityPolicies>)

```

and properties

```
.get/setHttpsSecurityPolicies()
.get/setCertificateValidator()
.get/setHostnameVerifier()
```

similar to the client side.

`HttpsSecurityPolicies.ALL` includes all default TLS security policies (which is {`TLS1_0`, `TLS_1_1`} at the moment).

5.3.2 Basic256Sha256 SecurityMode

`SecurityMode.ALL_102` includes now also the new profile, `Basic256Sha256`, which on the other hand requires at least 2048 bit certificate. If you only use a smaller one, the profile will not be enabled.

`SecurityMode.ALL`, which is the default, still equals to `ALL_101`, which does not include the new profile, since support for this mode is not available in other stacks, yet (as mentioned above).

5.4 NodeManager and server-side UaNodes

The big change in 2.0 is that the node objects for standard UA types are now based on generated classes (see 6.), whereas in 1.x they were hand-written. This affects a few things.

5.4.1 Node instantiation

In general, the nodes are now created with a `NodeBuilder` instead of the constructors. And there is a nice convenience method in `NodeManagerUaNode` for that: `createInstance()`, which you can use, for example as:

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

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
NodeIdBasedNodeBuilderConfiguration conf = new NodeBuilderConfiguration()
    .addOptional(Identifiers.AnalogItemType_EngineeringUnits.getValue())
    .addOptional(Identifiers.AnalogItemType_InstrumentRange.getValue())
    .addOptional(Identifiers.DataItemType_Definition.getValue());

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

Instead of using the `NodeId` of the instance declarations (elements of the type), you can use their `BrowseNames` or `BrowsePaths`.

5.4.2 Java classes for the nodes

SDK 1.0 contained implementations of the standard nodes in `com.prosysopc.ua.server.nodes.opcua`. In SDK 2.0, the standard nodes are in `com.prosysopc.ua.types.server.opcua`. These are generated classes and therefore they behave and are used a bit differently. As mentioned above, the nodes should be created with the `NodeBuilder` and they don't have any public constructor. They have similar setters and getters for the property and variable values as in the old SDK. In addition, each object and variable node is available from the classes.

If you have your own node implementations, you will need to modify them accordingly. The server side node implementations are now named `XxxTypeNode`, whereas in the SDK 1.0 they were `XxxType`. Also, you should not need to implement the sub nodes yourself in the Java types, but you can rely on the `NodeBuilder` to create the structure for you. You just need to ensure that the type definition defines the structure – and specifies the `ModellingRules` as well for the instance declarations.¹ See the `MyEventType` and `MyNodeManager` of the new `SampleConsoleServer`.

5.5 IoManager

`IoManager` includes new overridable methods, `beginRead`, `beginWrite`, `endRead` & `endWrite`, which can be used to handle complete read/write calls or just to prepare communications for the individual value settings.

5.6 EventType and EventManager

The `EventManager` itself is not changed very much. Triggering events is done similar as with the old event types: you create an instance of an event or condition node and call `triggerEvent()` for it.

The old `EventType` has been removed though, since now all the standard event types are generated for you. And you can instantiate new event objects using `NodeManagerUaNode.createEvent()`. See `MyNodeManager.sendEvent()` of the new `SampleConsoleServer`.

If you need to override your own event or condition types, note that the new standard types are now `XxxTypeNode`, instead of `XxxType`.

This affects also the `EventManagerListener`, which must refer to the new node types instead of the old ones.

5.7 HistoryManager

`HistoryManager` includes new overridable methods, `beginHistoryRead`, `beginHistoryUpdate`, `endHistoryRead` & `endHistoryUpdate`, which can be used to handle complete read/update calls or just to prepare a history dataset, which is then used when retrieving individual histories for specific readings. The begin methods can return a custom dataset (anything you need), which is then provided for the following methods as a parameter.

The same change applies to `HistoryManagerListener`.

You will need to modify your `HistoryManager` or `HistoryManagerListener` implementations according to the new method signatures.

6. Code generation and nodes

One of the biggest changes with SDK 2.0 is the code generator (codegen), which can be used to generate Java classes out of UA Nodeset definitions (in the Nodeset2.xml format).

The nodes used for the OPC UA standard types have now also been generated. This will lead to some changes, most notable in how the nodes must be constructed in the server side.

¹ Instance declaration is the object or variable of the type. It defines how the respective instance in the actual object or variable should appear.

In practice, this will affect the Server object (available from `NodeManagerRoot.getServerData()`) and the condition types. Since the nodes have also had a lot of functionality written into them, it is possible that the changes may affect your application.

The standard code-generated classes can be found in packages

- `com.prosysopc.ua.types.opcua` contains interfaces for each UA type
- `com.prosysopc.ua.types.opcua.client` contains client side implementations
- `com.prosysopc.ua.types.opcua.server` contains server side implementations

The Java interface for each UA types is generated as `xxxType`, the corresponding client side class as `xxxTypeImpl` and the server side class as `xxxTypeNode`. For example, `AnalogItemType`, `AnalogItemTypeImpl` and `AnalogItemTypeNode`. In addition, there are Base-classes for each class.

On the client side, the `getNode` methods from `AddressSpace` are overloaded with ones that take a `Class` parameter, therefore you can now call

```
AnalogItemType node = uaClient.getAddressSpace().getNode(NodeId,
    AnalogItemType.class)
```

See the documentation for the codegen for more details on the usage of it and the new nodes. Note that they are now available on the client side as well!

For the respective changes applied to the server, see 5.4 and 5.6.