Hewlett Packard Enterprise

HPE Network Node Manager i Software

Using the NNMi Web Services SDK

Release 10.20

White paper

Table of Contents

Introduction ... 3
Assumed Knowledge ... 3
WS-Interoperability (WS-I) Compared with WS-Eventing (WS-E) ... 3
Licensing and the SDK ... 3
NNMi WS-Interoperability (WS-I) Web Services ... 3
The InventoryClient Example ... 3
Generating a WS-I Web Service Client ... 8
Axis2 and wsdl2java 8
Basic HTTP Authentication 10
WS-I Filters 11
Run the WS-I Client 12
Adding additional WS-I clients 12
NNMi WS-Eventing (WS-E) Web Services ... 13
The NotifyClient Example ... 13
The Northbound EventInjectorTool ... 15
Generating a WS-E Notification Client ... 16
The dom4j Servlet Sample Client 16
Setting up a Maven Build Environment 18
Receiving WS-E *Node* Notifications 19
Receiving WS-E *Incident* Notifications 20
WS-E Filters 22
Specifying an Expiration Time 25
Tcpmon 25

Introduction

Network Node Manager i Software (NNMi) 8.0 introduced the concept of web service based integration.
Since NNMi 8.0, the NNMi web services APIs have grown to include even more functionality. This white paper explains the NNMi web services, how to exercise the most popular web service examples in the jmx-console, and how to develop a custom web service client to programmatically invoke the NNMi web services. The author of this white paper derived the content from many customer engagements. Using information from these engagements, this white paper answers the most common questions surrounding NNMi web service based integration and guides developers through the process of writing a custom web service client to invoke NNMi web services.

Assumed Knowledge

This white paper does not explain the larger concept of web services or the key technical components, such as WSDL, SOAP, XML, HTTP, WS-<*service_name*> specifications, and others. If you lack any of this knowledge, you will need to do some background reading. A basic knowledge of these concepts and technologies is necessary to understand this document.

You should have a working knowledge of the Java programming language, C++ programming language, or both if you intend to write a web service client. *Integrating with NNMi using web services requires programming client-side software.*

For the most part, this white paper uses examples from an NNMi 9.0x installation on a Linux NNMi management server. However, all of the functionality described in this white paper exists on all other NNMi supported operating systems running NNMi 9.0 or greater, including NNMi 9.20.

WS-Interoperability (WS-I) Compared with WS-Eventing (WS-E)

NNMi provides two basic types of web services. The first are WS-Interoperability (WS-I) services for creating, reading, updating, and deleting (CRUD model) NNMi inventory or configuration data. WS-I web services operate on a request-response model, meaning that a *3rd party client* initiates a call to an NNMi web services method to read or change NNMi data. An example of an NNMi WS-I web service is the `NodeBean` service which reads, updates, or deletes node inventory objects in NNMi.

In contrast to WS-I services are WS-Eventing (WS-E) services. WS-E services operate according to a publish-subscribe model and are used to ***push*** notifications of NNMi events to 3rd party listeners. This means that *NNMi* initiates the action to send event data to subscribed listeners using web services. Whereas in WS-I services the WS client initiated the web service operation, in WS-E services NNMi initiates the publishing of data. An example of an NNMi WS-E web service is the `Incident Notification` service that sends out real time notifications of new or changed NNMi incidents to subscribed 3rd party listeners.

Licensing and the SDK

In NNMi 8.x and 9.0x the SDK web services (both WS-I and WS-E web services) needed to be enabled with an Integration Enablement (IE) or an SDK Developer license. The SDK web services will not respond to SOAP requests until one or more of these licenses are installed. *This will affect both the SDK jmx-console examples and any custom written web service clients.*
In NNMi 9.20 and newer, you no longer need to enable SDK web services with a license. NNMi web services are enabled out-of-the-box. However, if you need access to the WS samples from sdk-dev-kit.jar (described below), you need to install an SDK Developer license. All NNMi web services in NNMi 9.20 (and newer) respond to SOAP requests without additional product licensing. In 9.20 and newer, the Integration Enablement license is obsolete, but the Developer license is still used to unlock the SDK documentation and sample Java code.

NNMi WS-Interoperability (WS-I) Web Services

Begin by looking at the NNMi WS-I web services. First, deploy and use the InventoryClient example which is available after installing the SDK Developer license on the NNMi management server. Next, step through the creation of a WS-I Java client using the Apache Axis2 toolkit.

The InventoryClient Example

As explained earlier, you must install an SDK Developer license on the NNMi management server to access any of the SDK samples or sample client code.

After you install the SDK Developer license, a new JAR file, `sdk-dev-kit.jar`, is placed in the following directory:

- *Windows*: `<install_dir>\doc`
- *UNIX*: `/opt/OV/doc`

1. Before extracting the contents of the JAR, make sure your Java environment is set up correctly. For windows, set the `JAVA_HOME` and `Path` variables to match your Java environment.
   On UNIX NNMI management servers, do this using the following commands:

```
#>  export JAVA_HOME=/opt/OV/nonOV/jdk/nnm
#>  export PATH=$JAVA_HOME/bin:$PATH
```

2. To confirm which JAR binary you are using, run the following command:

   ```
   #>which jar
   ```
   Example Display: `/opt/OV/nonOV/jdk/nnm/bin/jar`

3. Run the **jar –xvf** command to extract the contents of sdk-dev-kit.jar:
   a. #> **cd /opt/OV/doc**
   b. #>**jar –xvf sdk-dev-kit.jar**

After completing Run the **jar –xvf** command to extract the contents of sdk-dev-kit.jar: step, view the following files in the `/opt/OV/doc` (UNIX) or `<install_dir>\doc` (Windows) directory:

- META-INF
- nms-sdk-samples
- nms-sdk-sources.jar
- NNM-SDK.pdf

`NNM-SDK.pdf` is the official document describing the NNMi web services. Use this document as a reference for determining the location of the web services (hosted WSDL locations) as well as the methods, parameters, enum values, and data objects used by the web services.

The nms-sdk-samples directory is the primary location for the SDK sample code. Both the SDK sample client source code and compiled archives are included in each of the subdirectories of nms-sdk-samples as shown in

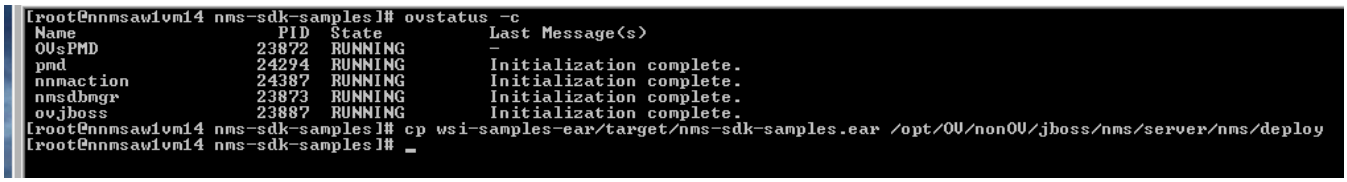
Figure 1.

Figure 1: SDK Sample Clients

4. All of the sample clients in this directory are already built into two consolidated enterprise archive at `nms-sdk-samples/wsi-samples-ear/target/nms-sdk-samples.ear and nms-sdk-samples/wsi-samples2-ear/target/nms-sdk-samples2.ear`
   To run all of the samples clients, copy the consolidated nms-sdk-samples.ear and nms-sdk-samples2.ear files into the NNMi ovjboss deploy directory as show in Figure 2.
   - *Windows*: `<install_dir>\NNM\server\deploy`
   - *UNIX*: `/opt/OV/NNM/server/deploy`

After the copy completes, the `ovjboss` process starts up all of the SDK client code in the `nms-sdk-samples.ear and nms-sdk-samples2.ear` files (assuming that the ovjboss process is currently running) as shown in Figure 2:

Figure 2: Copy the nms-sdk-samples.ear File into the deploy Directory

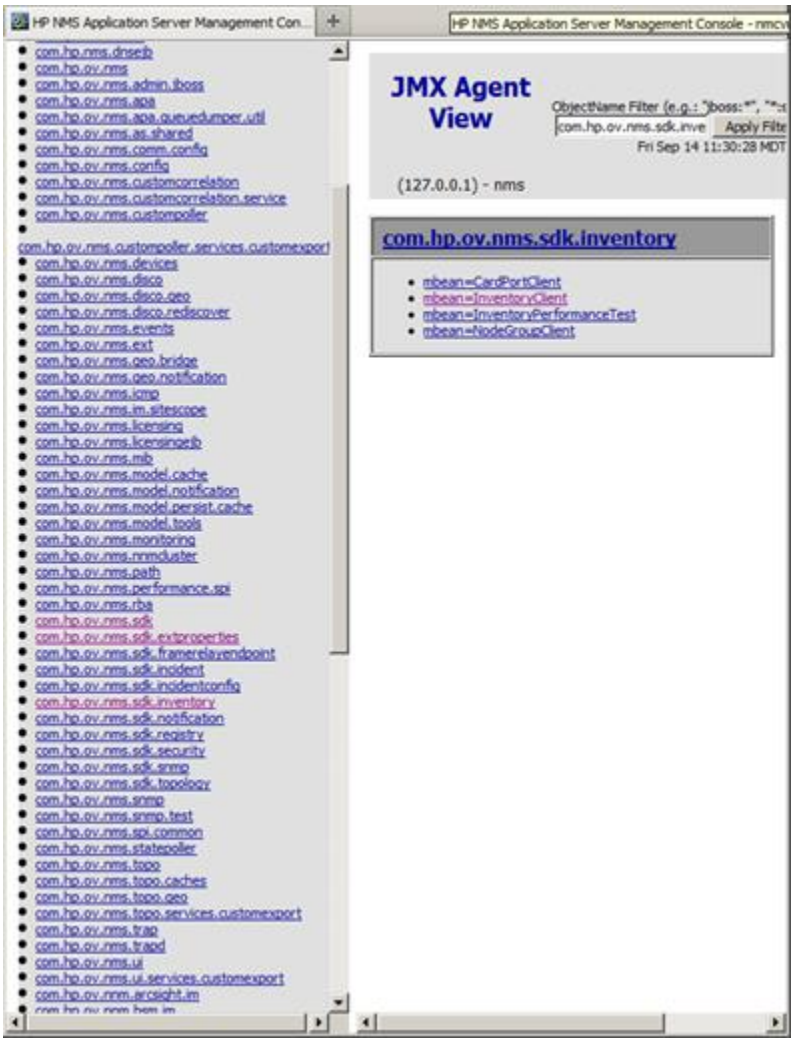


5. To confirm that the `ovjboss` process has started all of the SDK client code in the nms-sdk-samples.ear file, view the end of following file:
   - `Windows: <dir>:\ProgramData\HP\HP BTO Software\log\nnm\ovjboss.log`
   - `UNIX: /var/opt/OV/log/nnm/ovjboss.log`
6. After the ovjboss process has started up all the code in the `nms-sdk-samples.ear` file, look for the following log message in the `ovjboss.log` file:

```
INFO  [org.jboss.deployment.EARDeployer] Started J2EE application:
file:/opt/OV/NNM/server/deploy/nms-sdk-samples.ear
```

7. After the `ovjboss` process starts all of the SDK client code in the `nms-sdk-samples.ear` file, you can access the SDK samples from the NNMi jmx-console. To open the jmx-console, open your browser to http://<nnmServer>:<nnmPort>/jmx-console and log on as the **NNMi system user**. The **NNMi system user** is the only user able to access the jmx-console. If you are currently logged into the NNMi console as a different user, log out, then log back in as the **NNMi system user**.
8. After the browser displays the jmx-console, scroll down to the `com.hp.ov.nms.sdk` sample client mbeans as shown in Figure 3.

Figure 3: Scroll to the com.hp.ov.nms.sdk Sample Client mbeans



9. The `InventoryClient` is the primary sample client for exercising the NNMi WS-Interoperability (WS-I) web services. These are the web services that get and set inventory object data. Do the following, using Figure 4 as a guide.
    a. To access the `InventoryClient` mbean click the **mbean=InventoryClient** link in the jmx-console.
    b. After you see the `InventoryClient` mbean view, fill in the `username`, `password`, and `WSURL` attributes at the top of the page with values that are valid for your NNMi management server. You must specify an administrator user or a web services role user.
    c. Set the `WSURL` field to the fully-qualified domain name (FQDN) of the NNM management server. However, most of the time you can leave the `WSURL` field set to `localhost` since the `ovjboss` process invokes locally running web services.
    d. Add the port you normally use to access the NNMi console.
    e. After making these changes, click **Apply Changes** located at the bottom of the attribute list.

Figure 4

List of MBean attributes:

Name	Type	Access	Value	Description
Count	int	R	0	Attribute exposed for management
Password	java.lang.String	RW	mypassword	Attribute exposed for management
User	java.lang.String	RW	system	Attribute exposed for management
WSURL	java.lang.String	RW	http://localhost:80/	Attribute exposed for management

Apply Changes

10. The `InventoryClient` is a single access point for exercising many of the WS-I services, such as NodeBean, `InterfaceBean`, `IncidentBean` (this refers to the WS-I `IncidentBean`, not the WS-E `incident notification service`), and others.

    Scroll down to the `list*` methods to retrieve inventory items from NNMi. These methods call the basic `get` methods on the WS-I SDK beans. For example, `listNodes()` calls `NodeBean.getNodes()` and `listInterfaces()` calls `InterfaceBean.getInterfaces()`. The `list*` methods use predefined filters that can be selected using the `selectFilter()` method (directly above the `list*` methods). To see the list of predefined filters that might be applied to the `list*` methods to restrict the returned inventory objects, open the source Java code for `InventoryClient`. This code can be found at the following location:

    - *Windows*: *<install_dir>*\doc\`nms-sdk-samples/wsi-inventory-client/src/main/java/com/hp/ov/nms/sdk/inventory/InventoryClient.java`
    - *UNIX*: `/opt/OV/doc/nms-sdk-samples/wsi-inventory-client/src/main/java/com/hp/ov/nms/sdk/inventory/InventoryClient.java`

11. Near the top of the **java** class there are private methods titled `getCanned*Filters()` which store different predefined filters. The `selectFilter()` method, shown in Figure 5, sets the filter array index which, in turn, determines which of these predefined filters are applied to the query in the `list*` methods.

    Back in the jmx-console, supply an integer for the filter index as the single parameter to the `selectFilter()` method as shown in Figure 5Figure 6; then click **Invoke** to invoke that method. Use `index 0` to start.

Figure 5: The selectFilter() Method

void selectFilter()

Operation exposed for management

Param	ParamType	ParamValue	ParamDescription
p1	java.lang.Integer	0	(no description)

Invoke

12. Next, invoke the `listNodes()` method as shown in Figure 6.

Figure 6: The listNodes() Method

java.lang.String listNodes()

Operation exposed for management

Invoke

Following along in the code, one can see that invoking `InventoryClient.listNodes()` calls `NodeBean.getNodes()` with a node filter defined by the first canned node filter. This produces a rudimentary printout of the specified filter and all nodes returned to `InventoryClient` by the `NodeBean` web service using the first canned filter. As you can see in Figure 7, filter `index=0` is a simple filter that only includes the `includeCustomAttributes=true` constraint. The other predefined node filters are more restrictive and return smaller subsets of nodes.

Figure 7: The InventoryClient.listNodes Method

JMX MBean Operation Result `listNodes()`

Back to Agent View Back to MBean View Reinvoke MBean Operation

161 nodes found using filter:

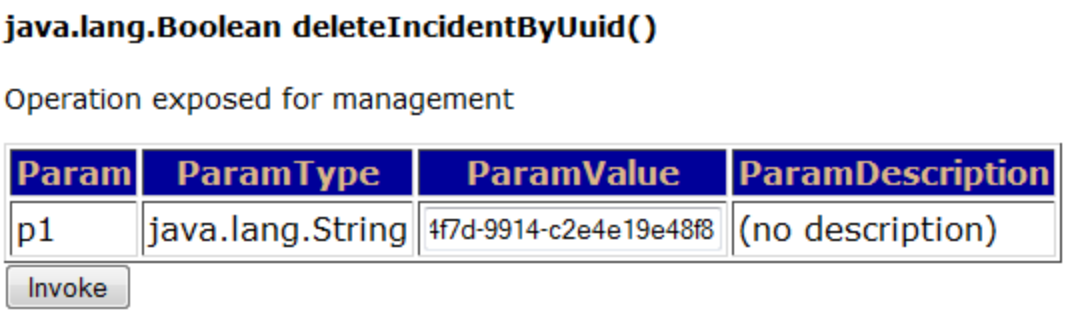
(includeCustomAttributes=true)

```
id, uuid, name, status, isSnmpSupported, systemName, systemContact, systemDescription, systemLocation, systemObjectId, longNam
2147527825, 57d4d451-581b-4931-941f-559a9480d871, gorams, CRITICAL, false, , , , , , gorams.fc.usa.hp.com, MANAGED, DISCOVERY_
2147525733, 7c61d57a-3ae2-46ec-85e4-2140176c33a1, nsntc-n3140-10, CRITICAL, false, , , , , , nsntc-n3140-10.fc.usa.hp.com, MAN
2147521334, 0adefade-5342-442d-acf1-f4a176866f01, procurvemesh3, CRITICAL, false, , , , , , procurvemesh3.fc.usa.hp.com, MANAG
2147527793, cc745233-eadc-4ebf-8288-70c1d74e063b, lab-a, CRITICAL, false, , , , , , lab-a.fc.usa.hp.com, MANAGED, DISCOVERY_CO
2147527889, 45bc352f-c200-4272-bded-2fc426d600c3, nsntc-n2840-1, CRITICAL, false, , , , , , nsntc-n2840-1.fc.usa.hp.com, MANAG
2147520709, e5e18f26-a7e5-4d9f-88e4-78cf653b1c17, redcloud, CRITICAL, false, , , , , , redcloud.fc.usa.hp.com, MANAGED, DISCOV
2147521489, bbe8c05c-fd32-44d3-8871-e26dfa20d272, mplspe05, NORMAL, false, , , , , , mplspe05.fc.usa.hp.com, MANAGED, DISCOVER
2147527929, 7debaa88-2f60-4441-ba3e-525275dfdc09, wan-vrrp-sw, CRITICAL, false, , , , , , wan-vrrp-sw.fc.usa.hp.com, MANAGED,
2147527853, 755624fb-6fab-4085-ad48-11a037f52ddc, ntc-clan1, NORMAL, false, , , , , , ntc-clan1.fc.usa.hp.com, MANAGED, DISCOV
2147527817, b70f1fa2-6bdc-4c8e-bcf5-f2c4295cacaf, gr2000a, NORMAL, false, , , , , , gr2000a.fc.usa.hp.com, MANAGED, DISCOVERY_
2147527901, 4df416e1-49f0-4f58-b922-fbed3bdcb6f6, dnali, CRITICAL, false, , , , , , dnali.fc.usa.hp.com, MANAGED, DISCOVERY_CO
2147527865, 58e08b3c-f4fa-4acf-a123-8e567fea8271, southwest-sw1, CRITICAL, false, , , , , , southwest-sw1.fc.usa.hp.com, MANAG
2147527779, 468a1b53-15d3-428d-9fff-28ce2dc950ed, ntc-medpro2-1a14, NORMAL, false, , , , , , ntc-medpro2-1a14.fc.usa.hp.com, M
2147527849, 4180023b-8ca5-4468-b36e-f8ee18d8b421, ntc-clan2, NORMAL, false, , , , , , ntc-clan2.fc.usa.hp.com, MANAGED, DISCOV
2147527925, 60425060-9d17-42f0-93d7-534a2b065a4a, edhostptllstflr, NOSTATUS, false, , , , , , edhostptllstflr.fc.usa.hp.com, N
2147527921, 25f7dea6-a64e-452c-abcb-1fb5dcd1edea, nortelnetsw1, NOSTATUS, false, , , , , , nortelnetsw1.fc.usa.hp.com, NOTMANA
2147527917, 3a887c58-6d37-4626-ae15-4551aef75bb1, nortela, NOSTATUS, false, , , , , , nortela.fc.usa.hp.com, NOTMANAGED, DISCO
2147527905, 4a5fb370-63d3-49b2-9f14-670f938ee11e, nortelb, NOSTATUS, false, , , , , , nortelb.fc.usa.hp.com, NOTMANAGED, DISCO
2147527897, 5956d7e4-47e8-4330-a130-d1f1bf827988, drams, NOSTATUS, false, , , , , , drams.fc.usa.hp.com, NOTMANAGED, DISCOVERY_
2147527893, 02be67ca-2adb-4566-83f2-585b495fa268, deco, NOSTATUS, false, , , , , , deco.fc.usa.hp.com, NOTMANAGED, DISCOVERY_C
2147527885, 1984c751-0415-4198-b9d9-795f6c9601de, southwest-gw1, NOSTATUS, false, , , , , , southwest-gw1.fc.usa.hp.com, NOTMA
2147527877, 3eb27568-8ca0-41ff-97e7-402ffbfff68c, drail, NOSTATUS, false, , , , , , drail.fc.usa.hp.com, NOTMANAGED, DISCOVERY_
2147527873, 9b860038-dcbd-483a-9952-fed308c667c9, ntc-val, NOSTATUS, false, , , , , , ntc-val.fc.usa.hp.com, NOTMANAGED, DISCO
2147527869, f20d39e7-1bd2-4f06-a45c-0b4fe77f1d45, vwan-switch-3, NOSTATUS, false, , , , , , vwan-switch-3.fc.usa.hp.com, NOTMA
2147527857, a30a5da3-e538-4f07-bc99-43690b7ee54a, ntc-medpro-1a10, NOSTATUS, false, , , , , , ntc-medpro-1a10.fc.usa.hp.com, N
2147527841, 9cb59c0e-8cec-4ac3-ad8c-80856497b864, ntc2rams, NOSTATUS, false, , , , , , ntc2rams.fc.usa.hp.com, NOTMANAGED, DIS
2147527837, 8851d771-ca3a-44a6-bfd8-2797ef5189c7, hendrix, NOSTATUS, false, , , , , , hendrix.fc.usa.hp.com, NOTMANAGED, DISCO
2147527833, ad5eb2ad-feab-48b0-8706-6806ac131b59, ipsi, NOSTATUS, false, , , , , , ipsi.fc.usa.hp.com, NOTMANAGED, DISCOVERY_C
2147527829, e759832f-525e-4e53-87c7-24fc420eb2c9, adamsst-main, NOSTATUS, false, , , , , , adamsst-main.fc.usa.hp.com, NOTMANA
2147527821, eac42903-8d78-433b-8666-317474a5585f, cisco0, NOSTATUS, false, , , , , , cisco0.fc.usa.hp.com, NOTMANAGED, DISCOVE
2147527811, 61601d33-7622-4e0f-ad8b-1ca8aeb86bdc, c3524xl, NOSTATUS, false, , , , , , c3524xl.fc.usa.hp.com, NOTMANAGED, DISCO
2147527783, 55305116-07d4-4dad-a231-3136e88e2849, ntc-clanses, NOSTATUS, false, , , , , , ntc-clanses.fc.usa.hp.com, NOTMANAGE
```

To observe a second example of `InventoryClient` functionality, invoke the `deleteIncidentByUUID()` method.

1. Open the NNMi console and go to the Incident **Browsing→All Incidents** view.
2. Select an incident of interest and open it as shown in **Error! Reference source not found.**. In the right pane of the incident form select the **Registration** ab. This shows the ID and the UUID of the incident.
   Note: The registration tab of *any* inventory object shows the ID and UUID of that object).
3. Select and copy the UUID of the incident.
4. Return to the `InventoryClient` in the jmx-console and enter the UUID as the single parameter to the `deleteIncidentByUUID()` method.
5. Invoke the `deleteIncidentByUUID()` method shown in Figure .

Figure 8: The deleteIncidentByUUID() Method



If successful, a `true` message is displayed. As with most of the SDK sample clients, the data presentation is very basic; the important part is the underlying web service calls. For this reason it is helpful and instructive to follow along with the sample method invocations by looking through the source code in `InventoryClient.java`.

6. After invoking the `deleteIncidentByUUID()` method, go back to the NNMi console and refresh the **All Incidents** view. Observe that the test incident disappears. Note that the incident is *not* closed (`lifecycle state = closed`); rather it is completely deleted from the database.

Generating a WS-I Web Service Client

In the last section, the WS-I SDK functionality was introduced using the InventoryClient example. Next, take a look at the basic steps used to generate a custom WS-I web service client using a standard web service toolkit. This is how most real-world web service integrations with NNMi will be done. The general approach is to generate Java or C stub code that can be programmatically included in other client code as a hook or entry point for invoking NNMi web service methods and returning NNMi data.

This example demonstrates how to generate NNMi web service stubs using the Apache Axis2 toolkit. Axis2 is open source and can be freely downloaded at http://axis.apache.org. The choice of a WS toolkit is up to the developer and toolkits other than Axis2 can certainly be used. In addition to Axis2, the NNMi team has developed and tested web service clients using the JBossWS toolkit (using the **wsconsume** command to generate stub code).

Note: The SDK sample clients that run using the `ovjboss` process, and whose java source code is available in <install_dir>/doc/nms-sdk-samples, use `JBossWS` web services libraries. As evidence of this, notice that the **java** classes import many classes from `org.jboss.ws.*` packages. However, all of the functions performed by the samples are also available to clients generated with Axis2. `JBossWS` is used for the sample clients so that they can be easily deployed using the `ovjboss` process without the need for additional libraries.

Important Note: Generating stubs for NNMi web services with Axis2 works well for the NNMi WS-I web services (NodeBean, InterfaceBean, IncidentBean, and others) but does *not* work well for the NNMi WS-E notification web services. There are multiple issues with the way that the `ovjboss` process hosts the WS-E WSDLs (mostly surrounding the import of other WS-E standard WSDLs: `ws-eventing.xml`, ws-eventing.xsd, and others) that the toolkit stub generation tools (**wsdl2java** in Axis2 and **wsconsume** in `JbossWS`) do not like. It *is* possible to manually modify the WS-E WSDL in such a way that Axis2 will generate code; however this paper describes an alternate, and perhaps easier, approach for developing a WS-E client.

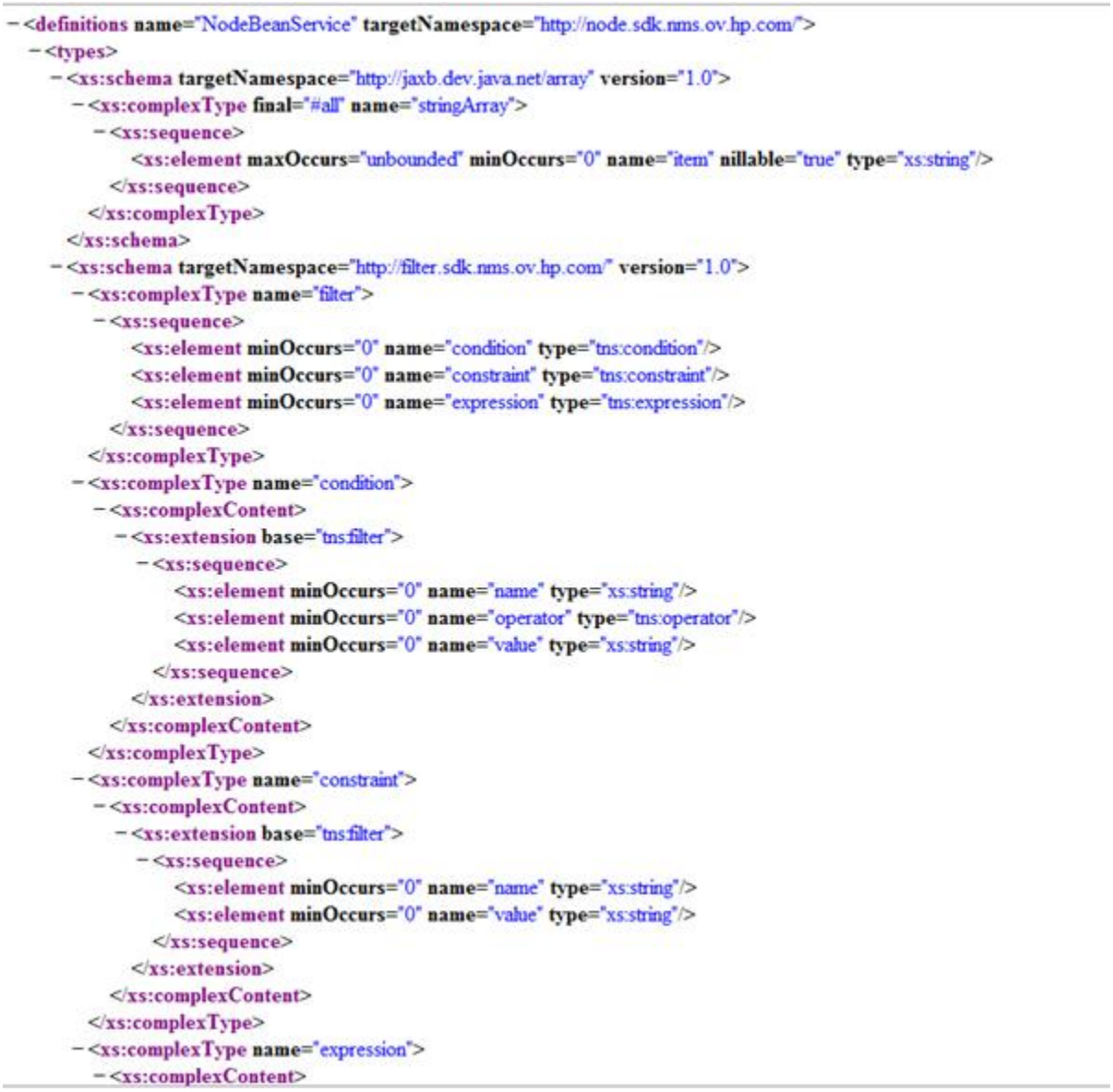
Axis2 and wsdl2java

In this example, you generate an Axis2 client stub for the `NodeBean` WS-I web service. After you have a working `NodeBean` client you add client stub classes for the `InterfaceBean` and `IncidentBean` WS-I services.

1. Check that you are able to access the NNMi Web Service Description Language (WSDL) documents from your desktop development system. As the name implies, WSDL documents describe web services: the inputs, outputs, and data types involved in each separate web service call. Tomcat running within `ovjboss` process hosts the WSDL documents just like any other web page or resource. The URIs for accessing all of the NNMi web service WSDLs are listed in NNM-SDK.pdf. In the case of the `NodeBean` WS-I web service, you can see the WSDL by pointing your browser to `http://<nnmServer>:<nnmPort>/NodeBeanService/NodeBean?wsdl`.

Try going to `http://<nnmServer>:<nnmPort>/NodeBeanService/NodeBean?wsdl` from a web browser on your desktop development system. If you are able to access the web service you will see the WSDL XML document displayed in your browser as shown in Figure 8.

Figure 8: The WSDL XML Document



```
- <definitions name="NodeBeanService" targetNamespace="http://node.sdk.nms.ov.hp.com/">
  - <types>
    - <xs:schema targetNamespace="http://jaxb.dev.java.net/array" version="1.0">
      - <xs:complexType final="#all" name="stringArray">
        - <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="item" nillable="true" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:schema>
    - <xs:schema targetNamespace="http://filter.sdk.nms.ov.hp.com/" version="1.0">
      - <xs:complexType name="filter">
        - <xs:sequence>
            <xs:element minOccurs="0" name="condition" type="tns:condition"/>
            <xs:element minOccurs="0" name="constraint" type="tns:constraint"/>
            <xs:element minOccurs="0" name="expression" type="tns:expression"/>
          </xs:sequence>
        </xs:complexType>
      - <xs:complexType name="condition">
        - <xs:complexContent>
          - <xs:extension base="tns:filter">
            - <xs:sequence>
                <xs:element minOccurs="0" name="name" type="xs:string"/>
                <xs:element minOccurs="0" name="operator" type="tns:operator"/>
                <xs:element minOccurs="0" name="value" type="xs:string"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      - <xs:complexType name="constraint">
        - <xs:complexContent>
          - <xs:extension base="tns:filter">
            - <xs:sequence>
                <xs:element minOccurs="0" name="name" type="xs:string"/>
                <xs:element minOccurs="0" name="value" type="xs:string"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      - <xs:complexType name="expression">
        - <xs:complexContent>
```

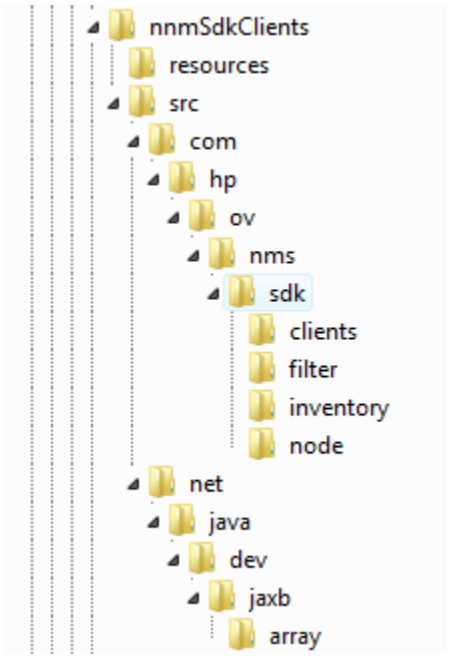
2. After you confirm that you can access this WSDL document from your development system, create the web service stub code using Axis2. If you do not yet have Axis2 installed on your development system, download the latest binary distribution (1.5.4 at the time of this writing) from http://axis2.apache.org and install it on your development system.
3. After verifying that Axis2 is installed on your development system, make sure the `AXIS2_HOME` environment variable points to the Axis2 home directory; then add `$AXIS2_HOME/bin` (UNIX) or `%AXIS2_HOME%\bin` (Windows) to your PATH environment variable. When properly configured, you should be able to execute the Axis2 **wsdl2java.sh** (UNIX) or **wsdl2java.bat** (Windows) scripts from a command prompt. Try running **wsdl2java.sh** or **wsdl2java.bat** with no arguments and confirm that the command usage message is displayed.
4. Create a new working root directory for your NNMi SDK client code. For this example, title the root directory `nnmSdkClients`. Change the working directory to the nnmSdkClients directory; then run **wsdl2java.sh** or **wsdl2java.bat** to generate the web service client stub code. Use the following parameters to generate synchronous methods only with extracted helper classes. Extracting the helper classes makes the stub code much more readable:

#> **wsdl2java.bat** -s -ss –g -o . -**uri** `http://<nnmServerFQDN>:<nnmPort>/NodeBeanService/NodeBean?wsdl`

Explanation of parameters:
-s = generate *synchronous* style code. I.e. code that blocks (waits) for a response from the web service.
-ss = split out the *server-side helper classes* rather than bundling them as inner classes of the main service stub class. If you fail to specify this option, the result will be the creation of a gigantic main service stub class with many inner helper classes.
-g = generate all classes. Required to generate the actual Axis2 service stub class with the –ss option.
-o = specify the output directory for the stub code. The '.' dot value means the current directory and assumes that you are executing wsdl2java from the 'nnmSdkClients' root directory of the project.
-uri = Endpoint Reference for the web service WSDL document.

Running the **wsdl2java.sh** or **wsdl2java.bat** command as shown above generates client stub code for invoking WS operations on the `NodeBean` WS-I web service. The stub code will be split into the directories shown in Figure 9.

Figure 9 Directories Generated with wsdl2java.sh or wsdl2java.bat.



5. Manually add the `clients` subdirectory after the **wsdl2java.sh** or **wsdl2java.bat** command completes. This is the subdirectory where the main client entry point classes will be created. These main client entry point classes will invoke the Axis2 service stub class which in turn will call the NNMi web services. The main Axis2 service stub for the `NodeBean` service is in the `node` subdirectory and is called `NodeBeanServiceStub.java`. NodeBeanServiceStub.java contains public methods for each of the node operations exposed through the `NodeBean` web service.

Basic HTTP Authentication

There is one crucial piece of code missing from the Axis2-generated service stubs. This is the code to authenticate a web service request to the `ovjboss` process. You need to modify the service stubs to supply basic HTTP authentication (a user and encrypted password) in the SOAP request sent to the NNMi web service.
Note: This example uses the `getNodes()` method as an example, however, the following code applies to all web service clients and methods. Every web service request sent to NNMi must contain basic HTTP authentication.
To do this, modify the contents of the `NodeBeanServiceStub.java` file.

- Add the code shown in red.
- Replace `nnmUser` and `nnmPw` with NNMi user credentials for an *administrator* or *web services role* user.
- `HttpTransportProperties` and `HTTPConstants` are **Axis2** classes that will be pulled in if the Axis2 lib directory is in the classpath (detailed below).

```
public  com.hp.ov.nms.sdk.node.GetNodesResponse getNodes(com.hp.ov.nms.sdk.node.GetNodes
getNodes178) throws java.rmi.RemoteException,com.hp.ov.nms.sdk.node.NmsNodeFaultException {

org.apache.axis2.context.MessageContext _messageContext = null;
  try{
       org.apache.axis2.client.OperationClient _operationClient =
       _serviceClient.createClient(_operations[9].getName());

       _operationClient.getOptions().setAction("http://node.sdk.nms.ov.hp.com/NmsNode/getNodesReques
       t");

       _operationClient.getOptions().setExceptionToBeThrownOnSOAPFault(true);

       HttpTransportProperties.Authenticator basicAuth = new
       HttpTransportProperties.Authenticator();

       basicAuth.setPreemptiveAuthentication(true);
       basicAuth.setUsername(nnmUser);
       basicAuth.setPassword(nnmPw);

       _operationClient.getOptions().setProperty(HTTPConstants.AUTHENTICATE, basicAuth);
...
```

WS-I Filters

In addition to the above HTTP authentication code, another common problem developers encounter when writing WS-I clients is constructing filters for the `get*` methods used to retrieve inventory objects. The relevant Java classes for constructing filters are located in the src/com/hp/ov/nms/sdk/filter directory generated by the Axis2 wsdl2java utility.

In most cases, the WS-I client wants to use the **Expression** class to combine filter constraints and conditions subfilters. The canned filters contained in the `InventoryClient.java` source code are the best examples to use when constructing Expression filters that combine multiple conditions and constraints. NNM-SDK.pdf also has a section that formally describes the structure of NNMi SDK filters.

Whereas conditions are used to filter inventory objects according to specified field values, constraints are used to attach meta-instructions to the filter, such as how many objects to retrieve and whether or not to retrieve optional fields. Historically, developers have found that the most important constraints to include as subfilters in their Expression filter are the following:

- **includeCustomAttributes/includeCias**: setting the `includeCustomAttributes` constraint to `true` tells an NNMi WS-I service to return the custom attributes associated with the inventory objects that match the other conditions in the filter. Setting the `includeCias` constraint to `true` tells the WS-I `IncidentBean` service to return the custom incident attributes associated with the incidents that match the other conditions in the filter.
- **offset**: this constraint is an integer index value that tells NNMi *where to start* when retrieving a batch of inventory objects. This constraint is used to iterate through a large set of inventory objects with multiple WS-I get* calls (see example below).
- **maxObjects**: this constraint is an integer that tells NNMi *how many* objects to retrieve in each get* call.

When retrieving a large set of inventory objects with a WS-I get* call, it is best to retrieve the large set in many small batches rather than making a single large get call that has the potential to timeout. The `offset` and `maxObjects` constraints permit these small, iterative batch calls. For example, suppose a WS-I client wanted to retrieve the entire set of nodes in the NNMi inventory; further suppose that NNMi contained 4,523 nodes in the inventory. Attempting to retrieve all 4,523 nodes in a single NodeBean.getNodes() call would likely lead to a timeout. A smarter approach is to retrieve the objects by looping through smaller calls to NodeBean.getNodes(). Each call to getNodes() would include the `offset` and `maxObjects` constraints to control which batch of nodes were returned. The series of WS calls would look like this:

```
getNodes() call 1: offset = 0, maxObjects = 500
getNodes() call 2: offset = 500, maxObjects = 500
getNodes() call 3: offset = 1000, maxObjects = 500
getNodes() call 4: offset = 1500, maxObjects = 500
getNodes() call 5: offset = 2000, maxObjects = 500
…..
getNodes() call 10: offset = 4500, maxObjects = 500
```

Assuming that the rest of the filter matches all 4,523 nodes in the inventory, calls 1 through 9 will return 500 nodes each. Call 10 will return the remaining 23 nodes (offset value 4500-4522; 4522 rather than 4523 because we began with offset=0, not offset=1).

The client code making the sequence of calls keeps making calls until the returned result set contains *fewer objects than the value assigned to* `maxObjects`. In other words, if NNMi returns the maxObjects number of inventory objects, then it is supposed that NNM has more objects to retrieve. 500 objects per call tends to be a nice batch size for WS-I get* calls to the NNMi SDK..

Run the WS-I Client

After modifying `NodeBeanServiceStub.java` to do basic HTTP authentication, write a client class (in the `clients` subdirectory) to invoke the methods in `NodeBeanServiceStub.java`. Your client can contain a `main(args[])` method and serve as the entry point to your java SDK client project or you can create another class to act as the entry point and call your client class.

After you code client and entry point classes for your java client project, compile the project by running the **ant** command from the root `nnmSdkClients` project directory. The **ant** command looks for the default `build.xml` file created by Axis2, and uses that file to determine how to build the project. To have the **ant** command available, set the correct entry point to your java project in the resulting compiled archive. You must specify a `Main-Class` parameter in the `build.xml` file to be placed in the resulting JAR's manifest.

1. Edit the `build.xml` file and add the following property at the end of the existing properties near the top of the `build.xml` file:

   ```
   <property name="main-class" value="com.hp.ov.nms.sdk.client.WsClientStartUp"/>
   ```

   The value of this property is the fully qualified (package+class) name of the Java class that contains a `main(args[])` method, and that will serve as the entry point to your client project. In this example, a class called WsClientStartUp.java was added to the `clients` subdirectory to serve as the program entry point. WsClientStartUp.java instantiates other client classes and delegates web service operations to them based on command line input parameters.
2. After adding this property, modify the **Ant** JAR task that creates the eventual client Axis archive (`.aar`) to add this main-class property value as the `Main-Class` in the archive's manifest. Note that a .aar archive is, for all practical purposes, identical to a .jar:

   ```
   <jar destfile="${lib}/${name}.aar">
               <fileset excludes="**/Test.class" dir="${classes}"/>
               <manifest>
             <attribute name="Main-Class" value="${main-class}"/>
               </manifest>
   </jar>
   ```

3. After modifying the `build.xml` file to denote the client entry point, save & close the `build.xml` file.
4. Run the **Ant** command from the project root directory.
5. Debug any compiler errors until the **Ant** command builds successfully. The result of a successful build will be the presence of an Axis archive (.aar) in the build/lib directory. In this example the archive is called `nnmSdkClients.aar`.
6. Run the client archive with **java –jar** and include the Axis2 lib directory in the classpath with the `java.ext.dirs` JVM parameter. There are many **Axis2** classes that the client stub needs to access, so pulling in the entire lib directory is easier than specifying each library by name in the classpath:

   java -Djava.ext.dirs="C:\axis2\lib" -jar build/lib/nnmSdkClients.aar *<param 1> <param 2> <param 3>… <param n>*
   Assumption: `axis2` has been installed on the development system in `C:\axis2`. In other words, the `AXIS2_HOME` environment variable value is set to `C:\axis2`.

Adding additional WS-I clients

After you finish generating the initial Axis2 client, adding additional WS-I service stubs is relatively easy. For example, to add a client for the InterfaceBean service, run the following **wsdl2java** command from the `nnmSdkClients` project root directory:

wsdl2java.bat -s -ss –g -o . -uri
`http://<nnmServerFQDN>:<nnmPort>/InterfaceBeanService/InterfaceBean?wsdl`

This adds a new `iface` subdirectory that contains the main `InterfaceBeanServiceStub.java` class and associated helper classes.

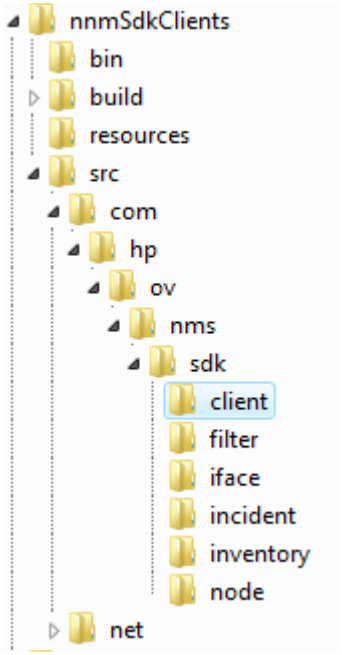
Run the following **wsdl2java** command to further add stub code for the `IncidentBean` web service:

wsdl2java.bat -s -ss –g -o . -uri `http://<nnmServerFQDN>:<nnmPort>/IncidentBeanService/IncidentBean?wsdl`

This adds a new `incident` subdirectory that contains the **IncidentBeanServiceStub.java** class and associated helper classes.

After running **wsdl2java** commands to generate stub code for the `NodeBean`, `InterfaceBean`, and `IncidentBean` WS-I services, the `nnmSdkClients` directory should resemble the file structure shown in **Error! Reference source not found.**.

Figure 10: The nnmSdkClients Directory



Notice the addition of the `iface` and `incident` subdirectories. Run additional **wsdl2java** commands to add stubs for any of the other NNMi WS-I web services. For a complete list of the WS-I web services, see NNM-SDK.pdf.

NNMi WS-Eventing (WS-E) Web Services

The NotifyClient Example

The `NotifyClient` example is the primary example client for exercising the NNMi WS-Eventing web services. These services include the incident notification service, the node notification service, the SNMP change notification service, and, new to NNMi 9.10, the security change notification service. Unlike the WS-I web services, which operate on a request/response model, the WS-E services operate on a publish/subscribe model. In the WS-I services (`NodeBean` and others), a web service client requests particular atomic operations on a web service to get or set data in NNMi. The WS-I web service performs the requested operation; then returns response data. In the WS-E notification services a web service client *subscribes* to a notification service and waits; the WS-E web service then asynchronously publishes events to the subscribed clients. To get a feel for the WS-E web services using the NotifyClient example, do the following, using Figure 11 as a guide.

1. To enter the `NotifyClient` mbean in the jmx-console, click the **mbean=NotifyClient** link from the main jmx-console screen.
2. Fill in the `Password`, `User`, `WSURLSource`, and `WSURLTarget` fields in the Mbean attributes; then click **Apply Changes**.
3. To begin using the sample client, set `WSURLSource` and `WSURLTarget` to the HTTP URL of the NNMi console. The mbean attributes section contains counter fields that are incremented when the sample WS-E target receives specific types of notifications from the NNMi notification WS-E services.

Figure 11

List of MBean attributes:

Name	Type	Access	Value	Description
Password	java.lang.String	RW	nnm	Attribute exposed for management
User	java.lang.String	RW	webservices	Attribute exposed for management
NodeCount	java.lang.Integer	R	0	Attribute exposed for management
WSURLSource	java.lang.String	RW	http://localhost:80/	Attribute exposed for management
WSURLTarget	java.lang.String	RW	http://localhost:80/	Attribute exposed for management
IncidentCount	java.lang.Integer	R	0	Attribute exposed for management
LastIncidentReceived	java.lang.String	R		Attribute exposed for management
ChangeCount	java.lang.Integer	R	0	Attribute exposed for management
LastChangeReceived	java.lang.String	R		Attribute exposed for management
CorrelationCount	java.lang.Integer	R	0	Attribute exposed for management
LastCorrelationReceived	java.lang.String	R		Attribute exposed for management
LastNodeReceived	java.lang.String	R		Attribute exposed for management
SnmpNotificationCount	java.lang.Integer	R	0	Attribute exposed for management
LastSnmpNodeUpdatedReceived	java.lang.String	R		Attribute exposed for management

Apply Changes

4. After filling in the appropriate `user`, `password`, and `URL` values, scroll down to the `subscribeIncidents` method; then click **Invoke**. This causes the `NotifyClient` to send a SOAP subscribe request to the NNMi `nms-sdk-notify` incident notification service running at `http://<nnmFQDN>:<nnmPort>/nms-sdk-notify/subscribe?wsdl`. After a successful subscription, NNMi returns a subscription response SOAP message with a subscription UUID, the notification endpoint reference (EPR) to which NNMi will send notifications, and an expiration time for the subscription. The `NotifyClient` displays the subscription response details shown in Figure 12.

Figure 12: Displayed Subscription Response Details

JMX MBean Operation Result `subscribeIncidents()`

Back to Agent View Back to MBean View Reinvoke MBean Operation

```
SubscriptionManager: http://nnmsaw1vm14/nms-sdk-notify/manage
SubscriptionID: urn:jbwse:41cca329-b4b9-4b55-a596-7e2ac7eb7845
Expires: Tue Apr 19 16:05:30 MDT 2011
Notify To: EPR {address=http://localhost:80/SampleSinkBeanService/IncidentSinkBean, refParam=null}
```

5. If the jmx-console displays an error message, examine the bottom of the following files for clues about what went wrong:
   - *Windows*: `<dir>:\ProgramData\HP\HP BTO Software\log\nnm\ovjboss.log`
   - *Windows*: `<dir>:\ProgramData\HP\HP BTO Software\log\nnm\nnm-trace.log`
   - *UNIX*: `/var/opt/OV/log/nnm/ovjboss.log`
   - *UNIX*: `/var/opt/OV/log/nnm/nnm-trace.log`.

If the jmx-console shows a response resembling the one shown in Figure 12, the `NotifyClient` has successfully subscribed to the `nms-sdk-notify` incident notification service.

The `NotifyClient's` subscription request tells the `nms-sdk-notify` service to send incident notifications to an endpoint reference at `http://localhost:80/SampleSinkBeanService/IncidentSinkBean` This is the URI where the sample notification endpoint class, `IncidentSinkBean`, is listening for incident notifications. The `IncidentSinkBean` is deployed in the `ovjboss` process alongside the `NotifyClient` (both are included in the `nms-sdk-samples.ear`). The java source code for both the `NotifyClient` and the `IncidentSinkBean` is included in the `nms-sdk-samples` directory:

- *Windows*: `<install_dir>\doc\nms-sdk-samples/wsi-inventory-client/src/main/java/com/hp/ov/nms/sdk/notification`

- *UNIX*: `/opt/OV/doc/nms-sdk-samples/wse-notification-client/src/main/java/com/hp/ov/nms/sdk/notification`

As with the WS-I `InventoryClient`, it is instructive to follow along in the code in `NotifyClient.java` and `NnmSubscriber.java` while invoking the `NotifyClient` sample methods in the jmx-console.
After `NotifyClient` is subscribed to the incident notification web service, go back in the web browser to the main `NotifyClient` mbean screen and invoke the `testIncidentNotification` method. After doing this, the jmx-console will display a generic success page stating that the `Operation completed successfully without a return value`. Now go back to the main `NotifyClient` mbean screen and press F5 to refresh the webpage. Observe the list of mbean attributes at the top of the page. The `IncidentCount` attribute has increased by one and the `LastIncidentReceived` attribute shows "`testIncident''`. Review the following steps to see what happened:

1. The `NotifyClient` subscribed to the `nms-sdk-notify` WS-E incident notification web service and requested that the web service send incident notifications to an EPR where the example `IncidentSinkBean` is listening (`http://localhost:80/SampleSinkBeanService/IncidentSinkBean`).
2. The `NotifyClient` sent a test Incident to NNMi (by invoking the testIncidentNotification method).
3. NNMi received the test incident and the `nms-sdk-notify` WS-E service sent out a notification of that new incident to all subscribed listener endpoints, including `IncidentSinkBean` at `http://localhost:80/SampleSinkBeanService/IncidentSinkBean`.
4. `IncidentSinkBean` received this notification.
5. `IncidentSinkBean` updated the `IncidentCount` and `LastIncidentReceived` attributes on `NotifyClient` to denote its newly received incident notification. When the `NotifyClient` mbean page was refreshed, the latest values updated by `IncidentSinkBean` were displayed in the `NotifyClient` mbean attributes.

In step 5, `IncidentSinkBean` updated the attributes on `NotifyClient` for the sake of convenience. This way the incident subscription, test incident injection, and notification can all be executed and observed from the same `NotifyClient` mbean view in the jmx-console. There is no other reason why the code running at the notification EPR (`IncidentSinkBean`) needs to access the subscriber code (`NotifyClient`). In other words, the code that handles the notification subscription can be separate from the code running at the notification EPR which receives, and acts on, the incident notifications.
Run similar tests using the `nms-sdk-node-notify` and `nms-sdk-snmp-notify` web services to receive node change and SNMP change notifications. To do this, invoke the appropriate subscribe operations in `NotifyClient`; then invoke the `testNodeNotification` or `testSnmpNotification` methods.

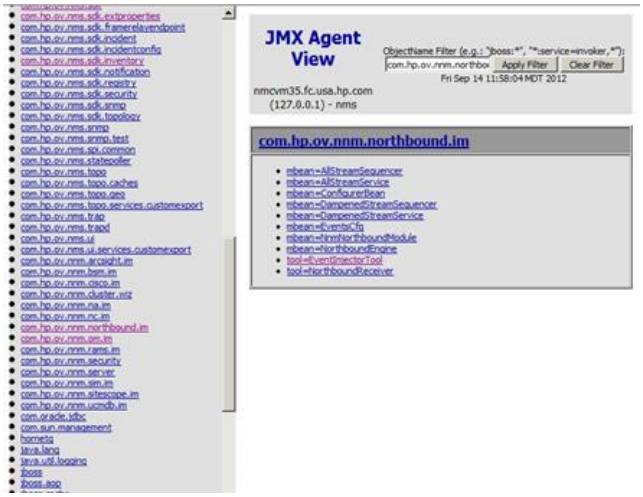
The Northbound EventInjectorTool

In addition to the `NotifyClient`, another helpful jmx-console mbean for WS-E notification testing is the `EventInjectorTool`. The `EventInjectorTool` is part of the NNMi Northbound interface (NBI) and was originally introduced as a testing tool for NBI. Use the `EventInjectorTool` to send test `NodeDown`, `InterfaceDown`, or `ConnectionDown` incidents to NNMi. In comparison, while you can use the `nnmsnmpnotify.ovpl` command line script to send test *traps*, using the `EventInjectorTool` is the only easy way to send in test *management events*. The `EventInjectorTool` tool is very handy for testing the NNMi WS-E `nms-sdk-notify` incident notification web service.

Do the following to use this tool:

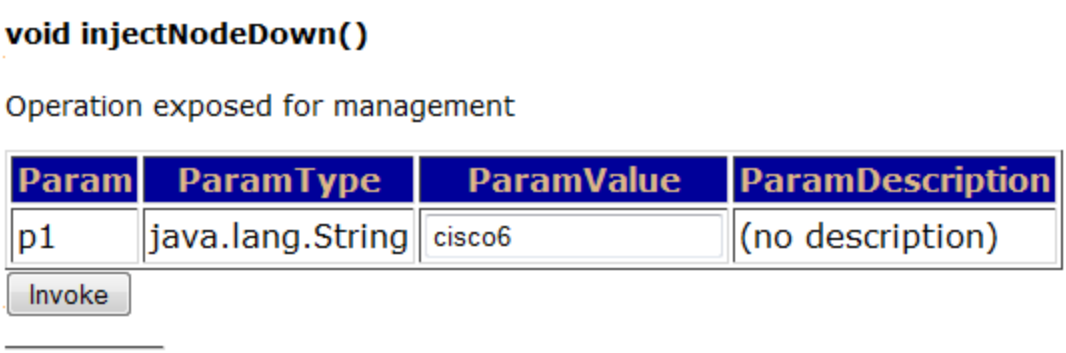
1. From the main jmx-console page, scroll down to the `Northbound` mbeans; then click the link titled, **tool=EventInjectorTool** shown in Figure 13.

Figure 13: Click tool=EventInjectorTool



2. From the `EventInjectorTool` mbean page, invoke the `listNodes()` method. This method displays a list of the nodes currently in the NNMi inventory. The `listNodes()` method displays each node by short name, then by fully qualified domain name (FQDN). The *short name* will be used to invoke the other methods of the `EventInjectorTool`.
3. Copy the short name of a particular test node from the left column of the `listNodes()` output; then use the browser window to view the main `EventInjectorTool` mbean page.
4. Invoke the `injectNodeDown()` method, shown in Figure 14, sending the short name of the test node as the single parameter.

Figure 14: Invoke the injectNodeDown() Method



Invoking this method injects a `NodeDown` incident on node cisco6. The incident will appear in the NNMi console in the **Incident Browsing→All Incidents** table as shown in (or in **Root Cause Incidents**, since `NodeDown` is a root cause management event).

Note: In NNMi 9.0x and above, many root cause incidents like `NodeDown` and `InterfaceDown` come configured with a default six minute dampening period and the incident table views come pre-configured to filter out incidents that are in a lifecycle state of `Dampened`. To see incidents in the `Dampened` lifecycle state, right-click the **Lifecycle State** (LS) column; then select, **Remove Filter**. Ensure that there are no other filters in place (such as a node group filter) that might block the test `NodeDown` incident from being displayed. After the filters are removed, the newly-injected `NodeDown` should be visible (with a `Dampened` lifecycle state if the default Incident Configuration is in place).

A notification of the new `NodeDown` incident will now be sent out using the `nms-sdk-notify` web service. If the `NotifyClient` is still subscribed to that `nms-sdk-notify` incident notification service, the `NodeDown` incident will be seen in the `lastIncidentReceived` attribute of `NotifyClient` in the jmx-console.
Note: The NotifyClient's subscription to `nms-sdk-notify` might have timed out and might need to be renewed.
Currently, only the `injectNodeDown()`, `injectInterfaceDown()`, and `injectConnectionDown()` methods on the `EventInjectorTool` mbean are hooked up. The other methods were stubbed out but never implemented. The `injectInterfaceDown()` and `injectConnectionDown()` methods inject incidents on the first ordered interface or connection associated with a given node. The input parameter to all three of these methods is the test node *short name* (for example, it uses `cisco6`, not `cisco6.hp.com`).

Generating a WS-E Notification Client

Now that you better understand the `NotifyClient` example, you can better understand the creation of a custom WS-E subscriber client. Unfortunately, creating a client to subscribe to one of the NNMi WS-E notification services is a more manual task than creating a WS-I client as described above. The reason for this is that WS toolkit stub code generation tools (both the Axis2 `wsdl2java` tool and the `JBossWS wsconsume` tool) fail to create stub code when given the NNMi WS-E WSDL as input. The reason for the code generation failures is the nature by which the `ovjboss` process hosts the NNMi WS-E WSDL. The `ovjboss` process uses `wsdl:import` statements to reference other elements and namespaces (`ws-evening.wsdl`, `ws-eventing.xsd`, and others) from the main notification `wsdl`. The stub code generation tools do not interpret these WSDL/XSD imports correctly.

Note: It *is* possible to generate subscriber stub code using Axis2 `wsdl2java`, but the process requires manually modifying the NNMi WS-E WSDL and also manually correcting a bug in Axis2 1.5.x which leads to a NullPointerException when given the *corrected* WSDL. Also, even after the Axis2 WS-E subscriber stub is generated, the subscriber does not send all info in its SOAP subscription request that the WS-E service requires. For all these reasons, it is much easier to use the included sample code as a starting point for writing a subscriber client rather than using Axis2 `wsdl2java`.

The dom4j Servlet Sample Client

The `NotifyClient` sample in the jmx-console described in the previous section demonstrates how the NNMi WS-E notification services work; however, the code behind the `NotifyClient` sample needs to be deployed in the `ovjboss` process to run, and is therefore not a very practical example for a developer who will create a subscriber client that runs on a separate system (the typical integration use case). A more practical example for most developers is the dom4j Notification Servlet Client example located in the following directory:

- *Windows:* `<install_dir>\doc\nms-sdk-samples\dom4j-notification-servlet-client`

- *UNIX*: `/opt/OV/doc/nms-sdk-samples/dom4j-notification-servlet-client`

This example is a very simple Java HTTP servlet that can be deployed in a web container like Tomcat. The servlet subscribes to an NNMi WS-E notification service (the `nms-sdk-node-notify` service by default), and hosts an endpoint for receiving notifications. You can change the default notification service to the `nms-sdk-notify` incident service or any other WS-E service. The hosting of an endpoint reference URI is the big advantage to running this code inside a web container like Tomcat. Tomcat handles the plumbing from the notification EPR to the receiving servlet java code.

To run the dom4j Notification Client servlet, download Tomcat from `http://tomcat.apache.org`, and install it on a development system that has HTTP access to the NNMi console. The process described in this white paper uses Tomcat version 7.0.14.

1. Download a Tomcat binary distribution for your platform and unzip the contents to a convenient location on the development system.
2. After Tomcat is extracted, make sure to set the `$CATALINA_HOME` environment variable to the Tomcat root directory. You might also want to add `$CATALINA_HOME/bin` to your `PATH` environment variable.

Note: Before proceeding with the NNMi example, try a test startup of Tomcat with `$CATALINA_HOME/bin/startup`. Confirm that Tomcat starts up cleanly without any port conflicts. By default, Tomcat uses port 8080 for HTTP and port 8443 for HTTPS. If either of these ports conflict with other applications on the development system change them in the `$CATALINA_HOME/conf/server.xml` file. The ports are set in the `<connector>` elements. Keep working on the port configuration until Tomcat starts up cleanly without any stack traces in the log.

After Tomcat is installed and configured on the development system, it is time to build the dom4j notification servlet. Like all the sample code in the `nms-sdk-samples` directory, the dom4j notification servlet comes prebuilt. You can find the prebuilt web archive in the following location:

- *Windows:* `<install_dir>\doc\nms-sdk-samples\dom4j-notification-servlet-client/target/NnmNotificationClient.war`
- UNIX: `/opt/OV/doc/nms-sdk-samples/dom4j-notification-servlet-client/target/NnmNotificationClient.war`

The main **NnmNotificationClient.java** class contains several hard coded parameters that need to be modified for your specific environment (WS source URL, WS Target URL, user, and password). You will need to modify these parameters and rebuild the code to change the servlet for your environment. You will also need to rebuild NnmNotificationClient.war to change the servlet to receive incident notifications rather than node notifications. For these reasons you need to set up a build environment to be able to modify the client code and rebuild `NnmNotificationClient.war`.

Setting up a Maven Build Environment

All of the example code in the `nms-sdk-samples` directory was built using Maven 2 and this is the easiest way to rebuild the examples, including the dom4j Notification Servlet Client. There are two basic options for setting up a Maven build environment to modify and rebuild the nms-sdk-samples:

Option 1: Install Maven 2 or later on the *NNMi management server*. The root `pom.xml` file assumes that Maven will be installed on the NNMi system and is written to pull JAR dependencies from the local `ovjboss` directory structure. Look for the `pom.xml` file in the following location:

- Windows: `<install_dir>\doc\nms-sdk-samples\pom.xml`
- UNIX: `/opt/OV/doc/nms-sdk-samples/pom.xml`

Those are the positive points for installing Maven on the NNMi management server. The challenges are that every time you change and rebuild a sample client like `NnmNotificationClient.war` you need to pull the rebuilt war archive down to your development system to run and test in Tomcat.

Option 2: Install Maven on the *desktop development system* running Tomcat. This approach requires a little initial setup (described below) and has the advantage of building the `NnmNotificationClient.war` on the same machine where Tomcat is running. Using this approach, you can copy each newly rebuilt `NnmNotificationClient.war` locally into the Tomcat webapps directory for deployment.

Wherever you decide to install Maven and build the nms-sdk-samples, do the following:

1. Download Maven 2 from http://maven.apache.org.
2. Unzip Maven to a convenient location.
3. Set the `$M2_HOME` environment variable and add `$M2_HOME/bin` to your $PATH environment variable.
4. If you are running Maven on the NNMi system try to run **mvn install** from the `nms-sdk-samples` root directory.

Note: In NNMi 8.1x and 9.0x the pom.xml files need a version number adjustment (there is a mismatch between the root `pom.xml` version number and the version numbers referenced in the child project pom.xml files). In NNMi 9.20 and newer the build should succeed without any `pom.xml` modification.

5. If you are running Maven on a separate development system do the following:
    a. Copy the entire `nms-sdk-samples` directory from the NNMi management server to the developer system.
    b. After the `nms-sdk-samples` is copied to the developer system, run **mvn install** from the root `nms-sdk-samples` directory. The first Maven build will fail and display a list of dependency archives that need to be installed in the local Maven repository on the development system. These dependency installs into the local Maven repository are a one-time setup step. All of the required library archives can be found on the NNMi system at the following locations:

Windows:

```
<install_dir>\NNM\lib\nms-sdk.jar
<install_dir>\nmsas\client\jbossall-client.jar
<install_dir>\nmsas\client\jbossws-client.jar
<install_dir>\nmsas\jbossws-common.jar
<install_dir>\nmsas\client\jbossws-spi.jar
<install_dir>\nmsas\client\servlet-api.jar
<install_dir>\nmsas\client\jboss-annotations-ejb3.jar
<install_dir>\nmsas\client\jboss-system-client.jar
<install_dir>\NNM\server\deploy\jbossws.sar\jbossws-core.jar
<install_dir>\nmsas\client\jboss-ejb3x.jar
<install_dir>\nmsas\client\jboss-jaxrpc.jar
<install_dir>\nmsas\client\jboss-jaxws.jar
<install_dir>\nmsas\lib\jboss-jmx.jar
```

UNIX:

```
/opt/OV/NNM/lib/nms-sdk.jar
/opt/OV/nmsas/client/jbossall-client.jar
/opt/OV/nmsas/client/jbossws-client.jar
/opt/OV/nmsas/client/jbossws-common.jar
/opt/OV/nmsas/client/jbossws-spi.jar
/opt/OV/nmsas/client/servlet-api.jar
/opt/OV/nmsas/client/jboss-annotations-ejb3.jar
/opt/OV/nmsas/client/jboss-system-client.jar
/opt/OV/NNM/server/deploy/jbossws.sar/jbossws-core.jar
/opt/OV/nmsas/client/jboss-ejb3x.jar
/opt/OV/nmsas/client/jboss-jaxrpc.jar
/opt/OV/nmsas/client/jboss-jaxws.jar
/opt/OV/nmsas/lib/jboss-jmx.jar
```

c. Pull these JAR files down to the development system and install each one by running the **mvn install:install-file** command for each dependency JAR in the failed build display from step b. Use the following as an example:

mvn install:install-file -DgroupId=com.hp.ov.nms.sdk -DartifactId=nms-sdk-jar -Dversion=1.0 -Dpackaging=jar –Dfile nms-sdk.jar
This will install the nms-sdk.jar into the local maven repository at ~/.m2/repository.

6. After all of the above archives are installed in the developer system's local maven repository, edit the root level `pom.xml` file in the `nms-sdk-samples` directory.
   d. In the `<dependencies>` section, change each of the `<scope>` elements from `system` or `provided` to `compile`.
   e. Comment out the `<systemPath>` elements if they exist.
   f. **Save & close** the `pom.xml` file and try to build from the root `nms-sdk-samples` directory using the **mvn install** command.

Receiving WS-E *Node* Notifications

1. After the **mvn install** command is building successfully from the root nms-sdk-samples directory, edit the `nms-sdk-samples/dom4j-notification-servlet-client/src/main/java/com/hp/ov/nms/sdk/notification/NnmNotificationClient.java` file. You can view the file contents in Figure 15.

Figure 15: Contents of the NnmNotificationClient.java File

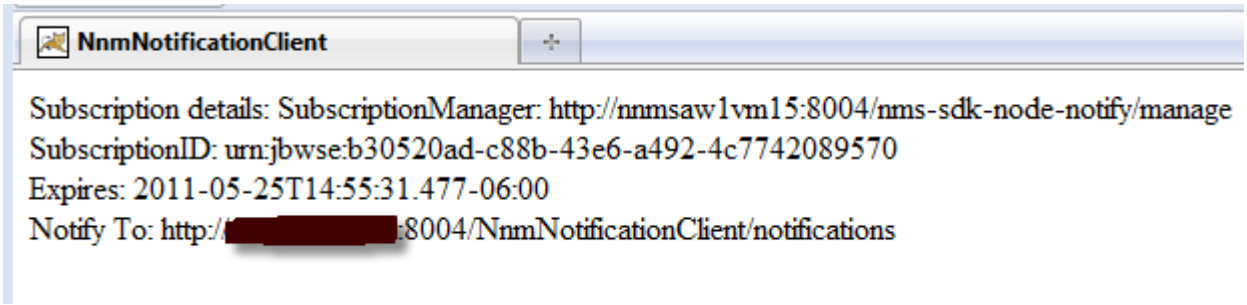
```
 2⊕ * (C) Copyright 2005 Hewlett-Packard Development Company, L.P.
19 package com.hp.ov.nms.sdk.notification;
20
21⊕import java.io.BufferedReader;
42
43⊖/**
44  * Client for Nnm notifications via WS-Eventing
45  *
46  * @author Rocky
47  *
48  */
49 public class NnmNotificationClient extends HttpServlet {
50     private static final Logger log = Logger.getLogger(NnmNotificationClient.class.getName());
51     private final static String nnmNotificationNamespace="http://notification.sdk.nms.ov.hp.com/";
52     private String source="nnm.company.com:80";
53     private String target="tomcatListenerSystem.company.com:8080";
54     private String user="system";
55     private String password="openview";
56     private static NnmSubscriber notificationSubscriber=null;
57     private final static String notificationSource="/IncidentNotificationSource";
58     private final static String notificationContext="nms-sdk-notify";
59     private final static String notificationNamespace=nnmNotificationNamespace+notificationContext;
60     private final static String notificationSourceUri=notificationNamespace+notificationSource;
61     private final static String notificationEndPoint="NnmNotificationClient/notifications";
62     private ArrayList<String> notifications=null;
63
```

2. Change the source, target, user, and password class variables to valid values in your environment. The source is the FQDN and port where the NNMi console is accessed. The target is where notifications should be sent. This is the development system where Tomcat is running plus the port at which Tomcat is listening for http connections. The user should be an NNMi administrator or web services role user.
3. After making these changes, run the **mvn install** command from the root `nms-sdk-samples` directory to rebuild the `NnmNotificationClient.war` file.
4. Stop Tomcat if it is running (run **$CATALINA_HOME/bin/shutdown**); then deploy the newly rebuilt `NnmNotificationClient.war` file in Tomcat by copying the `.war` file from the `nms-sdk-samples/dom4j-notification-servlet-client/target/NnmNotificationClient.war` file into `$CATALINA_HOME/webapps`.
5. Start Tomcat (run **$CATALINA_HOME/bin/startup**). Tomcat will automatically expand the `NnmNotificationClient.war` file into a `NnmNotificationClient` directory with `META-INF` and `WEB-INF` subdirectories. The `WEB-INF` directory contains the compiled code (in the classes directory), dependency jars (in the lib directory), and the `web.xml` deployment descriptor.
6. Open the `web.xml` file and observe the URIs that Tomcat is using to host services for the `NnmNotificationClient` servlet.

```
<servlet>
      <servlet-name>NnmNotificationClient</servlet-name>
      <servlet-class>com.hp.ov.nms.sdk.notification.NnmNotificationClient</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>NnmNotificationClient</servlet-name>
      <url-pattern>/subscribe</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>NnmNotificationClient</servlet-name>
      <url-pattern>/unsubscribe</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>NnmNotificationClient</servlet-name>
      <url-pattern>/notifications</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>NnmNotificationClient</servlet-name>
      <url-pattern>/results</url-pattern>
   </servlet-mapping>
```

7. Look at the `NnmNotificationClient.doRequest()` method in the Java code to see the actions taken when pointing your browser to each of these URIs:
   - Pointing your browser to `http://<tomcatFQDN>:<tomcatPort>/NnmNotificationClient/subscribe` will cause the servlet to subscribe to the NNMi WS-E notification service (the `nms-sdk-node-notify` service by default).
   - Pointing your browser to `http://<tomcatFQDN>:<tomcatPort>/NnmNotificationClient/unsubscribe` will cause the servlet to unsubscribe from the NNMi WS-E notification service.
   - Pointing your browser to `http://<tomcatFQDN>:<tomcatPort>/NnmNotificationClient/results` will show the notifications that the servlet has received from NNMi.
   - http://<tomcatFQDN>:<tomcatPort>/NnmNotificationClient/notification is the endpoint reference (EPR) URI to which NNMi sends node change notifications.
      a. Open a web browser on your system running Tomcat and point the browser to http://<TomcatFQDN>:<TomcatHTTPPort>/NnmNotificationClient/**subscribe**. This causes the Notification Client to subscribe to the NNMi nms-sdk-node-notify WS-E web service. After a successful subscription, the servlet displays the details of the subscription as shown in Figure 16.

Figure 16: Successful Subscription to the nms-sdk-node-notify WS-E Web Service



The Notification Client servlet is now subscribed to the `nms-sdk-node-notify` WS-E service and has asked NNMi to send node notifications to the Tomcat system at the Tomcat HTTP port (port 8004 in Figure 16). Notice that, by default, the subscription will expire within five minutes of the time of subscription. You can override this by specifying a different `<expire>` element in the SOAP subscription request.

8. Now that the servlet client is subscribed to the `node notification` service, go to the **Inventory→Nodes** table in the NNMi console. Select a node (preferably an SNMP node, that is, a node with a valid device profile set); then change the management mode of the node from **managed** to **not managed** or **out of service** (or change it back to **managed** from a **non-managed** state). Doing this causes a node notification message to be sent from the `nms-sdk-node-notify` service to the `subscribed listener` servlet. You can see some basic details about the notification by going to `http://<TomcatFQDN>:<TomcatHTTPPort>/NnmNotificationClient/results`.

Receiving WS-E *Incident* Notifications

Do the following to configure `NnmNotificationClient.java` to receive incident notifications rather than node notifications:

1. Change the class variables at the top of the `NnmNotificationClient.java` file. These variables configure the URI the Tomcat servlet uses when subscribing to the WS-E service. Change the values as shown in Figure 17.

Figure 17: Change the Class Variables

```
private final static String notificationSource="/IncidentNotificationSource";
private final static String notificationContext="nms-sdk-notify";
```

   Completing the changes shown in Figure 17 points the Tomcat servlet subscribe request to the `nms-sdk-notify` incident notification service rather than the `nms-sdk-node-notify` node notification service.
2. Change all references to the **com.hp.ov.nms.sdk.node.NodeNotification** class to the **com.hp.ov.nms.sdk.incident.IncidentNotification** class. Look for these helper classes in `WEB-INF/lib/nms-sdk-jar-1.0.jar`. Make these changes in both the `NnmNotificationClient.java` and `NnmSubscriber.java` files.
3. Change the `NnmNotificationClient.onNotification()` and `NnmSubscriber.onNotification()` methods as shown in Figure 18 and Figure 19Figure 18 to handle incident notifications rather than node notifications:

Figure 18: NnmNotificationClient.onNotification() Changes

```
protected String onNotification(HttpServletRequest request, HttpServletResponse response) throws Serv
    StringBuffer sb = new StringBuffer();
    BufferedReader reader = request.getReader();

    String s = reader.readLine();
    sb.append(s);
    while (s != null) {
        s = reader.readLine();
        if (s != null) sb.append(s);
    }

    reader.close();

    String results = "";
    if(sb!=null && sb.length()!=0 && !sb.toString().equals("null")) {
        IncidentNotification incNotification=notificationSubscriber.onNotification(sb.toString());
        results=new Date(System.currentTimeMillis()).toString()+": ";
        results+=incNotification.getSourceNodeName() + ": " + incNotification.getFormattedMessage();
    }
    if(notifications==null) {
        notifications=new ArrayList<String>();
    }
    notifications.add(results);

    return results;
}
```

Figure 19: NnmSubscriber.onNotification() Changes:

```
public IncidentNotification onNotification(String notification) {
    IncidentNotification incidentNotification=null;
    try {
        SAXReader saxReader = new SAXReader();
        Document document = saxReader.read(new StringReader(notification));

        incidentNotification=new IncidentNotification();
        incidentNotification.setFormattedMessage(document.selectSingleNode("//formattedMessage").getText());
        incidentNotification.setSourceNodeName(document.selectSingleNode("//sourceNodeName").getText());
    } catch(Exception e) {
        log.log(Level.SEVERE,"Error parsing notification: ",e);
    }
    return incidentNotification;
}
```

4. By completing the previous steps, you rewrote the dom4j Notification Client servlet to receive incident notifications from `nms-sdk-notify` rather than node notifications from `nms-sdk-node-notify`. Now rebuild the `.war` using the **mvn install** command; then redeploy it in the Tomcat `webapps` directory.

Important Note: When redeploying in Tomcat, first stop Tomcat by running the **$CATALINA_HOME/bin/shutdown** command, then delete both the existing `$CATALINA_HOME/webapps/NnmNotificationClient.ear` file *and* the entire expanded `NnmNotificationClient` directories.

WS-E Filters

A common question among developers is how to specify a filter in the SOAP subscribe message so that NNMi will only send out notifications that match the filter. To do this, specify an **XPath expression** in the <Filter> element of the SOAP subscribe message. If you specify an XPath <Filter> element, NNMi (actually the ovjboss process) only sends out notifications where the notification XML matches the XPath expression.

The first step in writing a WS-E XPath filter expression is to examine the SOAP notification messages that are sent from NNMi to learn the structure of the XML. A good way to do this is to setup Tcpmon as a proxy between a subscribed listener client and NNMi. Setting up Tcpmon is described later in this document.

Below is a typical NodeDown incident notification observed in Tcpmon.

```
<env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope'
xmlns:wsa='http://www.w3.org/2005/08/addressing'
xmlns:wse='http://schemas.xmlsoap.org/ws/2004/08/eventing'>
   <env:Header>
      <wsa:Action>http://notification.sdk.nms.ov.hp.com/nms-sdk-
notify/IncidentNotificationSource/Notification</wsa:Action>
      <wsa:To>http://localhost:9899/SampleSinkBeanService/IncidentSinkBean</wsa:To>
   </env:Header>
   <env:Body>
      <sys:onNotification xmlns:sys='http://notification.sdk.nms.ov.hp.com/nms-sdk-notify'>
         <arg0>
            <id>150324833045</id>
            <uuid>1408799e-2dbf-4daa-b19f-b22c6444eae8</uuid>
            <name>NodeDown</name>
            <sourceUuid>02836032-7626-4334-85ab-a26d9ecd6237</sourceUuid>
            <sourceName>bay450sw</sourceName>
            <sourceNodeUuid>02836032-7626-4334-85ab-a26d9ecd6237</sourceNodeUuid>
            <sourceNodeName>bay450sw</sourceNodeName>
            <sourceNodeLongName>bay450sw.company.com</sourceNodeLongName>
            <lifecycleState>com.hp.nms.incident.lifecycle.Registered</lifecycleState>
            <severity>CRITICAL</severity>
            <priority>com.hp.nms.incident.priority.None</priority>
            <category>com.hp.nms.incident.category.Fault</category>
            <family>com.hp.nms.incident.family.Node</family>
            <nature>ROOTCAUSE</nature>
            <origin>MANAGEMENTSOFTWARE</origin>
            <duplicateCount>0</duplicateCount>
            <rcaActive>true</rcaActive>
            <formattedMessage>Node Down</formattedMessage>
            <originOccurrenceTime>2011-08-24T15:11:52.684-06:00</originOccurrenceTime>
            <firstOccurrenceTime>2011-08-24T15:11:52.684-06:00</firstOccurrenceTime>
            <lastOccurrenceTime>2011-08-24T15:11:52.684-06:00</lastOccurrenceTime>
```

```
            <incidentResent>0</incidentResent>
            <created>2011-08-24T15:11:52.715-06:00</created>
            <previousLifecycleState/>
            <previousRcaActive/>
            <cias>
               <name>cia.sourceNode.OdbId</name>
               <type>STRING</type>
               <value>70a9cf8db9a2dcd8a2db2a0888055d548</value>
            </cias>
            <cias>
               <name>cia.sourceNode.UcmdbId</name>
               <type>STRING</type>
               <value>8088cb7722c6a1d72c845239b52e3a198</value>
            </cias>
         </arg0>
      </sys:onNotification>
   </env:Body></env:Envelope>
```

To subscribe to the nms-sdk-notify incident service to *only* receive NodeDown incidents one could use the following XPath Filter:

```
//arg0[name='NodeDown']
```

This filter will cause the notification service to only send out notifications that contain an XML node named 'arg0' with a child node named 'name' which in turn has the value 'NodeDown'.

As a second example, to subscribe to all incidents *except* those with a severity or NORMAL or WARNING, specify the following XPath filter:

```
//arg0[not(severity='NORMAL' or severity='WARNING')]
```

This filter will cause the notification service to only send notifications of MINOR, MAJOR, and CRITICAL severities.

As a third example, to subscribe to all incidents *except* trap OID .1.3.6.1.6.3.1.1.5.3.1.3.6.1.4.1.9 (CiscoLinkDown trap) specify the following XPath filter:

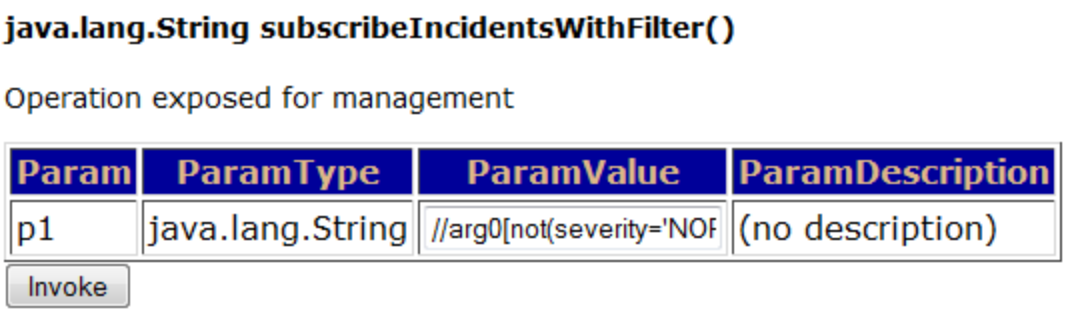
```
//arg0[not(cias) or not(cias/name='cia.snmpoid' and
cias/value='.1.3.6.1.6.3.1.1.5.3.1.3.6.1.4.1.9')]
```

This filter matches any notification XML that 1) does not contain a <cias> element, or 2) contains one or more <cias> elements but none of which have a name value 'cia.snmpoid' and a value of '.1.3.6.1.6.3.1.1.5.3.1.3.6.1.4.1.9'

There are many good resources on the web for learning or refreshing on XPath. The formal XPath specification is documented at http://www.w3.org/TR/xpath.

The best way to test XPath WS-E filters with the NNM notification services is to use the previously described NotifyClient jmx-console sample. NotifyClient contains methods for subscribing with filters that take in an XPath expression as the single String parameter. For example, one can test an incident filter XPath expression using the subscribeIncidentsWithFilter() method:

Figure 20 Using the subscribeIncidentsWithFilter() Method to Test an Incident Filter XPath Expression



Important Note: When typing XPath expressions into the subscribe*WithFilters() methods do not copy and paste single or double quote characters. This will insert some invalid ASCII characters into the XPath expression that is sent to the ovjboss process and the ovjboss process will not be able to correctly interpret the filter. To ensure that your XPath expressions are syntactically correct, run the **tail-f /var/opt/OV/log/nnm/ovjboss.log** command as you invoke the subscribe*WithFilters() methods. If the ovjboss process is unable to interpret the XPath expression you will see errors similar to the following logged to the ovjboss.log:

2011-08-25 11:25:12,415 ERROR [org.jboss.ws.extensions.eventing.mgmt.Subscription] Failed to evalute xpath expression javax.xml.transform.TransformerException: A location path was expected, but the following token was encountered: Â'WARNINGÂ'

The strange-looking Â characters are the result of copying and pasting single or double quote characters into the jmx-console input text field. The presence of these characters will cause the ovjboss process to fail to apply the XPath filter. Watch the ovjboss.log file for these types of error messages when you invoke subscribe operations that specify an XPath filter.

Specifying an Expiration Time

A second common question among developers is how to specify an expiration time in the SOAP subscribe message and how to renew a subscription once it has expired. The maximum WS-E subscription time permitted by the `ovjboss` process is ten minutes. In the `NnmSubscriber.createSubscribeMsg()` method, you can specify this as shown in

Figure 21.

Figure 21 Specify the Maximum WS-E Subscription Time Allowed by the ovjboss Process

```
Element expires = factory.createElement(new QName("Expires", EVENTING_NAMESPACE));
String expireTime=format(new Date(System.currentTimeMillis() + 10*60*1000L));
expires.setText(expireTime);
subscribe.add(expires);
```

Most integrators will want ongoing subscriptions that last indefinitely until explicitly cancelled. The solution is to implement a re-subscription algorithm in some type of timed loop. The NotifyClient jmx-console example shows a method for doing this using the **java.util.TimerTask** class.

Open both the `NotifyClient.java` and `NnmSubscriber.java` files. There are two alternatives for making a subscription continuous:

1. The subscriber client can continually send a WS-E Renew SOAP message to the NNM notification service before the current subscription expires.
2. The subscriber client can continually cancel the existing subscription before it expires with a WS-E `SubscriptionEnd` SOAP message; then send a new Subscribe SOAP message.

For either alternative, the <*expires*> element in the SOAP message must be continually increased to a time that is a maximum of ten minutes in the future.

Note: the NotifyClient example has its own **NnmSubscriber** class which is slightly different than the dom4j Notification Servlet Client example's **NnmSubscriber** class.

Tcpmon

Tcpmon is a useful tool for seeing web service XML traffic flowing over HTTP between NNMi and a web service client. Tcpmon acts as a *proxy* between an NNMi web service and the invoking client:

1. The client sends an XML SOAP message to Tcpmon.
2. Tcpmon displays the message; then forwards the SOAP message to NNMi.
3. The web service response flows back from NNMi, through Tcpmon, then to the client.

Seeing the sequence and content of web service SOAP message is helpful in understanding how communication between NNMi and the WS client happens, and what data is being exchanged.
Tcpmon is free and open source. It is available at http://ws.apache.org/commons/tcpmon.

The easiest way to start using Tcpmon with the NNMi web services is to direct the jmx-console SDK examples to send traffic *to Tcpmon* rather than NNMi, then have Tcpmon forward the traffic to NNMi.
To do this, do the following:

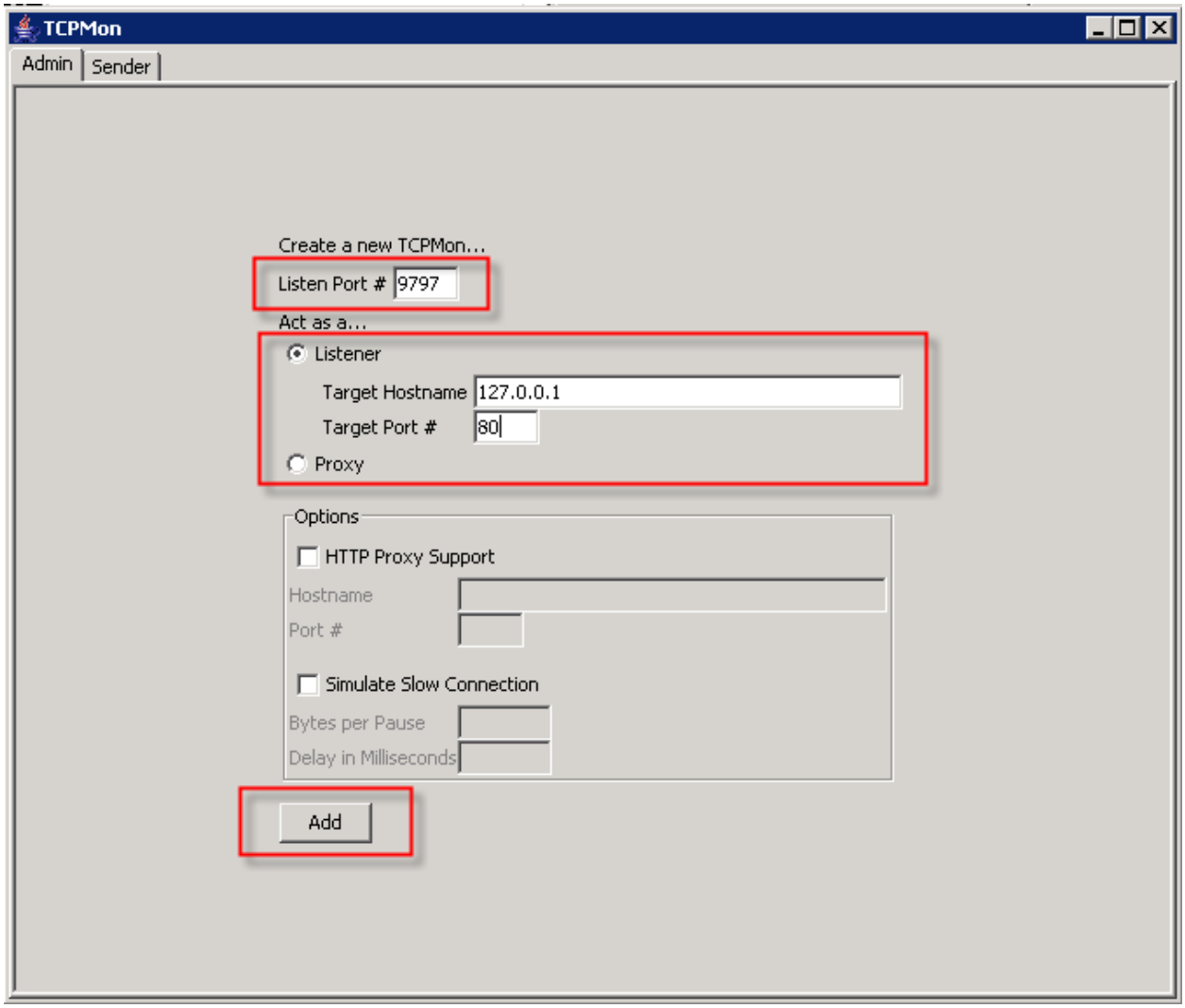
1. Download and extract Tcpmon to a convenient location on either a desktop development system or the NNMi management server.
2. Run Tcpmon by running **build/tcpmon.sh** or **build/tcpmon.bat**. Doing so brings up the Tcpmon user interface.

Note: If you run Tcpmon on a UNIX server, make sure your X-windows display is set up properly to run a graphical application.

To set up a Tcpmon listener to act as a proxy between the WS-E NotifyClient example and the NNMi management server, do the following using Figure 22 as a guide.

1. Click the **Admin** tab; then configure a new Tcpmon listener.
2. The `Listen Port #` is the port where Tcpmon listens for incoming traffic.
3. The `Target Hostname` and `Target Port #` are where Tcpmon forwards the traffic it receives. In Figure 22, Tcpmon is running on the NNMi management server and listening for incoming traffic on local port 9797. It then forwards the traffic it receives on port 9797 to NNMi listening at 127.0.0.1:80.

Figure 22: Setting up a Listener to Act as a Proxy between the WS-E NotifyClient Example and the NNMi Management Server



4. Click the **Add** button to start this listener. Doing so creates a new top level tab for the listener.
5. Create a second listener with Listen Port = 9798 forwarding to 127.0.0.1:80. The result is two Tcpmon listener tabs: one for listen port 9797 and one for listen port 9798, both forwarding to NNMi on the local system at port 80.
6. On each of the two new Tcpmon tabs, select the XML Format checkbox at the bottom of the screen as shown in Figure 23.

Figure 23: Select the XML Format Checkbox

7. Navigate to the `NotifyClient` mbean in the NNMi jmx-console.
8. In the top mbean attributes set the `WSURLSource` to the `system:port,` as shown in Figure 24, where the first Tcpmon listener is running.
9. Set the `WSURLTarget` to the `system:port,` as shown in Figure 24, where the second Tcpmon listener is listening.
10. Make sure to click **Apply Changes**.

Figure 24: Set WSURLSource and WSURLTarget

WSURLSource	java.lang.String	RW	http://tcpmonSys:9797/
WSURLTarget	java.lang.String	RW	http://tcpmonSys:9798/

Changing the `WSURLSource` tells `NotifyClient` to send its WS-E Subscribe request to Tcpmon listening on some remote system at port 9797. Tcpmon then forwards the Subscribe request to NNMi. In Figure 22, Tcpmon is running on the NNMi management server itself, so it forwards the traffic to NNMi at 127.0.0.1:80.

Changing the `WSURLTarget` changes the `<NotifyTo>` element in the NotifyClient's WS-E Subscribe request. This tells the NNMi WS-E service to send event notification to Tcpmon listening at port 9798. Tcpmon then forwards those notifications to NNMi port 80 where the example `SinkBean` is waiting to receive the notification. If you want NNMi to forward the notifications through Tcpmon to a separate development system, change the Tcpmon 9798 port listener to forward to the development system instead of back to the NNMi jmx-console examples.

11. After changing the `WSURLSource` and `WSURLTarget` attributes and applying the changes, invoke the `subscribeIncidents()` method.
12. Open Tcpmon and look at the port 9797 listener. The WS-E subscribe request from NotifyClient should be displayed in the upper window and the NNMi WS-E service's response in the lower window as shown in Figure 25.

Figure 25: The WS-E Subscribe Request from NotifyClient and the NNMi WS-E Service's Response

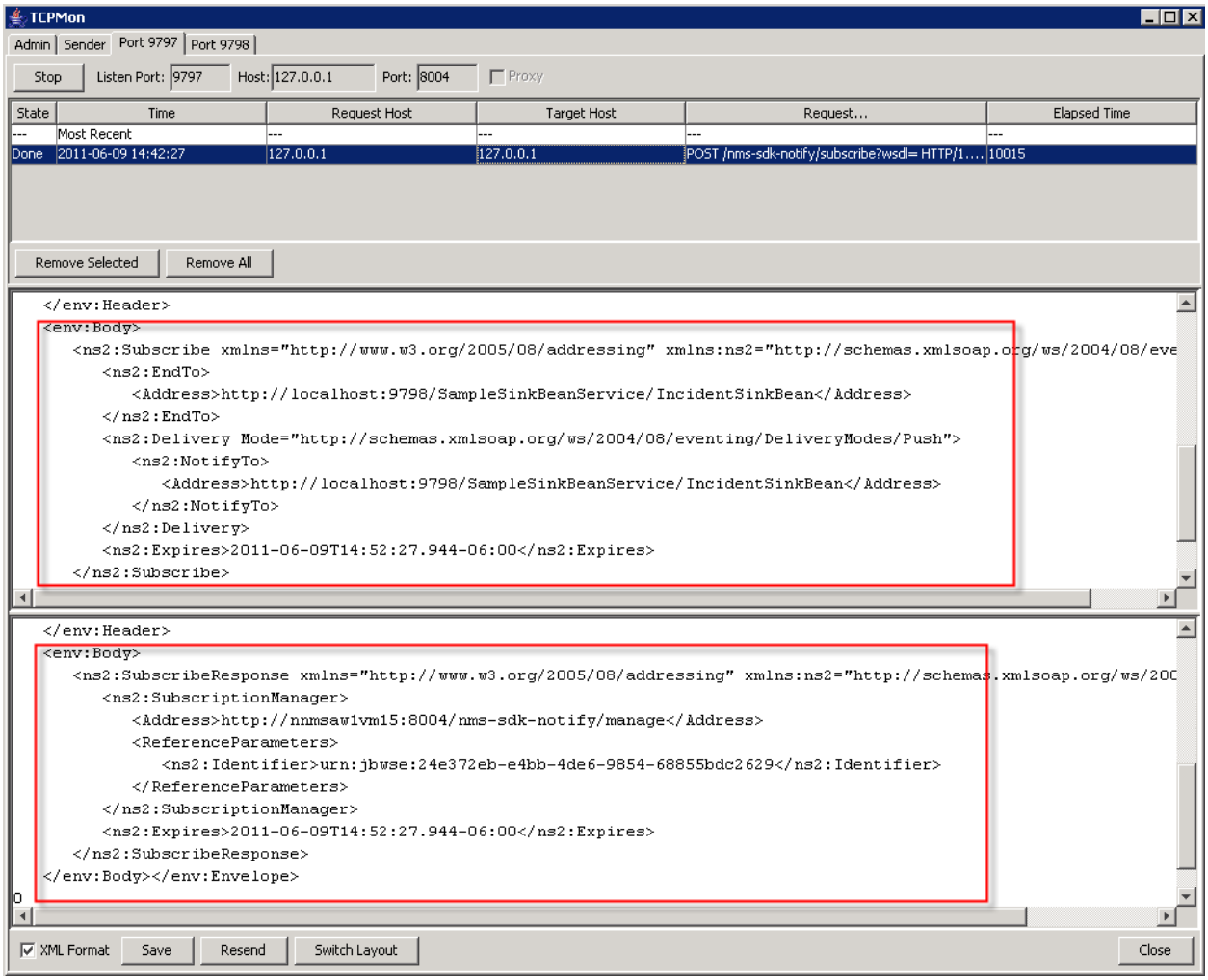
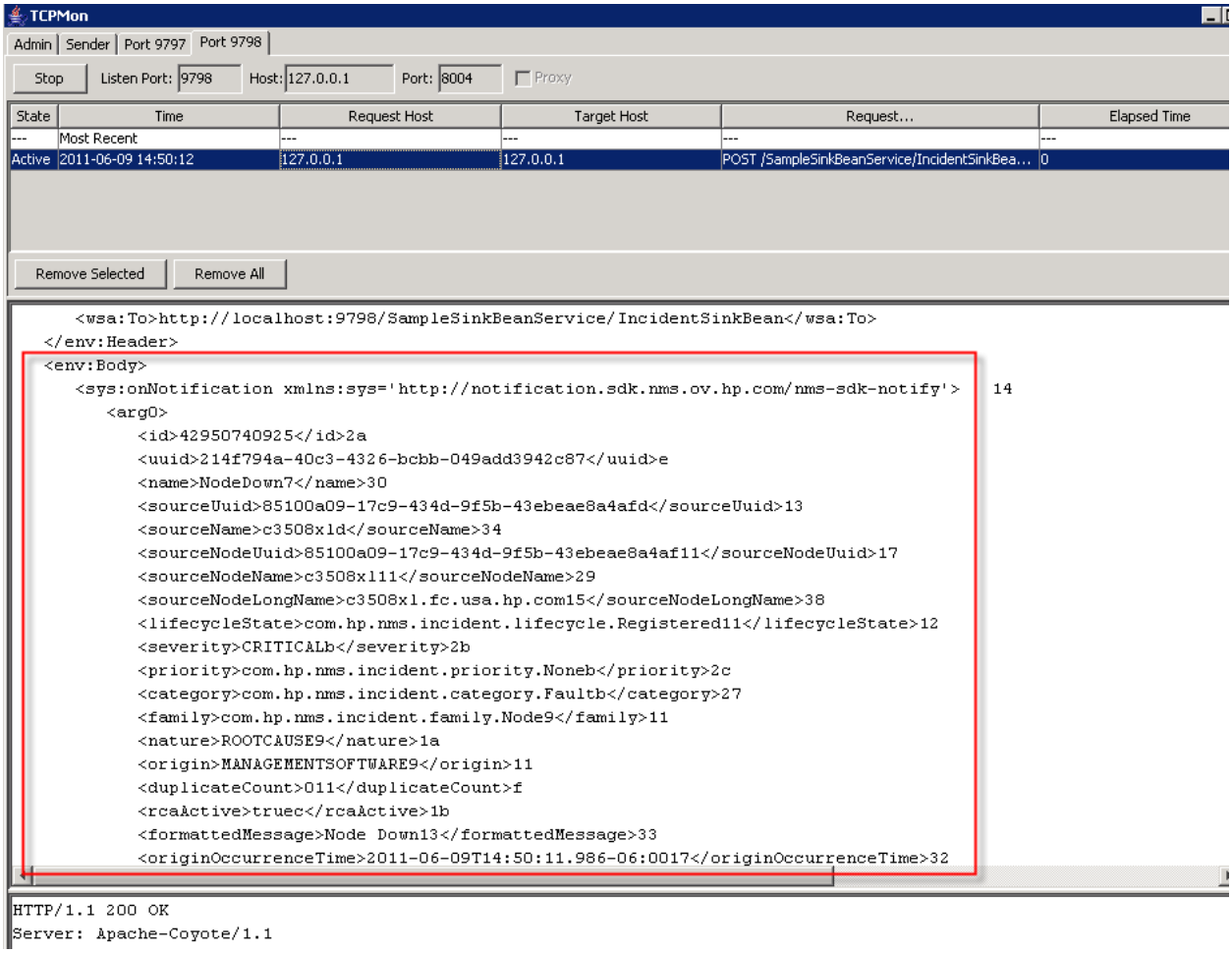


Figure 25 shows the XML traffic that passes across HTTP to subscribe to an NNMi WS-E notification service. The nms-sdk-notify incident notification service is now set up to send incident notifications to localhost:9798 where the second Tcpmon listener is listening.

13. Return to the main `NotifyClient` mbean view and invoke the `testIncidentNotification()` method, or go to the Northbound EventInjectorTool and send in a test `NodeDown`, `InterfaceDown`, or `ConnectionDown` incident.
14. Click the Tcpmon **Port 9798** listener tab as shown in Figure 26. NNMi creates a new incident and the WS-E notification service sends out a notification of that incident to subscribed listeners. The incident notification SOAP messages should be displayed as they flow from the NNMi WS-E Service, through Tcpmon at port 9798, to the NotifyClient IncidentSinkBean (or whichever notification endpoint you told the port 9798 Tcpmon listener to forward to.

Figure 26: Click the Tcpmon Port 9798 Listener Tab



All of the incident fields available to a subscribed client are visible in the Tcpmon-displayed message. The subscribed client can extract any of these data fields from the notification message.

Hewlett Packard
Enterprise