

## Chapter 8

# Developing Applications That Use NIC

This chapter describes how to develop an external application to interact with a network information collector (NIC). This chapter contains the following sections:

- External Application Requirements for NIC on page 163
- External Non-Java Applications That Use NIC on page 163
- External Java Applications That Use NIC on page 165

### External Application Requirements for NIC

---

If you write an external application to use NIC to perform a resolution, you can include NIC functionality in one of the following ways:

- For non-Java applications, use the interface module `NicAccess`, an IDL file that provides access to the NIC locator feature. The NIC locator can resolve the value of one or more keys.
- For Java applications, include the NIC proxy client libraries to use NIC in client/server mode.
- For Java applications, include the NIC proxy client libraries and the NIC host client libraries to use NIC in local host mode.

### External Non-Java Applications That Use NIC

---

If you write an application in a language other than Java, you can use the NIC access interface module, a simplified CORBA interface, to perform one or more resolutions.

For information about the NIC access interface module, see the API documentation in the SDX software distribution in the folder `SDK/doc/idl/nic` or on the Juniper Networks Web site at

<http://www.juniper.net/techpubs/software/management/sdx/api-index.html>

## Creating a NIC Locator to Include with a Non-Java Application

A NIC locator provides the same functionality as a NIC proxy, but is designed to work with non-Java applications.

You use the NIC access interface module to include NIC locators with your application by compiling the IDL file with your application files.

To use the NIC access interface module to create NIC locators:

1. Connect to the directory.
2. Obtain a CORBA reference to the NIC access interface from one of the following:

- The access IOR provided in the directory in the dynamic configuration DN under the hostname—typically, *host/demohost*.

You can read this information from SDX Admin from a host under *ou = dynamicConfiguration, ou = Configuration, o = Management, o = umc*.

- The path name of the IOR file (*nic.ior*) on the NIC host—typically, */opt/UMC/nic/var/run/nic.ior*.

- A corbaloc URL in the format:

```
corbaloc::<host>:8810/NicAccess
```

3. From the NIC access interface module, obtain a NIC locator, as identified by *NicFeature*. For example:

```
feature = access.getLocatorFeature(nicNameSpace); //nicNameSpace example "/NicLocator/ip"
```

4. Search for the key. For example:

```
feature.lookupSingle(NicLocatorKey key) //NicLocatorKey is coming from the IDL
```

For information about the NIC access interface module, see the API documentation in the SDX software distribution in the folder *SDK/doc/idl/nic* or on the Juniper Networks Web site at

<http://www.juniper.net/techpubs/software/management/sdx/api-index.html>

## External Java Applications That Use NIC

---

If you write an external Java application that interacts with a NIC, include NIC libraries in the application. These libraries are for NIC proxies and local NIC hosts. These libraries are located in the SDX distribution under *SDK/lib/nic*.

Typically, each NIC resolution process requires one NIC proxy. For example, the OnePopLogin sample data includes two resolution processes:

- Mapping of a subscriber's IP address to the subscriber's login name
- Mapping of the subscriber's login name to the SAE reference

An application that uses both these resolution processes would require two NIC proxies.

The NIC proxy provides a simple Java interface, the NIC application programming interface (API). You configure the NIC proxy to communicate with one resolver. For efficiency if you use NIC in client/server mode, the NIC proxy caches the results of resolution requests so it can respond to future requests for the same key without contacting the resolver.

The SDX software includes a factory interface, the NIC factory, to allow applications to instantiate, access, and remove NIC proxies. It also includes JAR files for NIC client and NIC host libraries.

### Developing a Java Application to Communicate with a NIC Proxy

You must configure an application to communicate with a NIC proxy.

If you are using Java Runtime Environment (JRE) 1.3 or higher, you must include in your application the Java archive (JAR) files, which are in the SDX software distribution in the folder */SDK/lib/*, with your application:

Configuration tasks that use the API calls to communicate with the NIC proxy are:

1. Instantiating a Configuration Manager on page 166
2. Passing a Reference to the Configuration Manager to the NIC Factory on page 166
3. Instantiating the NIC Factory Class on page 166
4. (Optional) Initializing Logging on page 167
5. Instantiating the NIC Proxy on page 168
6. Managing a Resolution Request on page 168
7. Deleting Invalid Results from the NIC Proxy's Cache on page 170
8. Removing the NIC Proxies on page 170

For more information about the API calls, see the online documentation in the SDX software distribution in the folder */SDK/doc/nic* or on the Juniper Networks Web site at

<http://www.juniper.net/techpubs/software/management/sdx/api-index.html>

### Instantiating a Configuration Manager

The application must instantiate a configuration manager.

To enable the application to instantiate a configuration manager to obtain a NIC instance from the NIC factory:

- Call one of the following methods:
  - For some applications (other than Web applications), in which you must define the system property `-DConfig.bootstrapFilename`, you can call the following method:

```
ConfigMgr configMgr = ConfigMgrFactory.getConfigMgr();
```

- For Web applications, you can instantiate the configuration manager as follows:

```
ConfigMgr configMgr = ConfigMgrFactory.getConfigMgr(properties);
```

- `properties`—`java.util.Properties` object, typically the bootstrap file, which contains all the configuration properties for the NIC proxy.

### Passing a Reference to the Configuration Manager to the NIC Factory

To pass a reference to the configuration manager to the NIC factory class:

- Call the following method in the application:

```
NicFactory.setConfigManager(configMgr);
```

### Instantiating the NIC Factory Class

The way you instantiate the NIC factory depends on the object request broker (ORB) configuration:

- If the NIC proxy uses the default ORB, call the following method in the application:

```
NicFactory nicFactory = NicFactory.getInstance();
```

This code instantiates a new NIC factory. Unless the `NicFactory.destroy` method has been called, subsequent calls to this method will return the instantiated NIC factory.

- If the NIC proxy does not use the default ORB, call the following method:

```
NicFactory.initialize(props);
NicFactory nicFactory = NicFactory.getInstance();
```

- props—java.util.Properties object, which contains the ORB properties for the NIC proxy. For example, if the NIC proxy uses JacORB but JacORB is not the default ORB, the ORB properties are:

```
org.omg.CORBA.ORBClass=org.jacorb.orb.ORB
org.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton
```

This code will instantiate a new NIC factory using the specified ORB. Unless the application has called the `NicFactory.destroy` method, subsequent calls to the `getInstance()` method will return the instantiated NIC factory. However, if the application has called the `destroy()` method, it must recall the `initialize()` method before it can call the `getInstance()` method.

For information about the `NicFactory.destroy` method, see *Removing the NIC Proxies* on page 170.

### Initializing Logging

You must initialize logging only if you want to view the logging information produced by the NIC proxy.

To enable the application to initialize logging:

- Call the following method:

```
Log.init(configMgr, configNameSpace);
```

- configMgr—Instance of the configuration manager, the value returned from the `getConfigMgr()` method
- configNameSpace—String that specifies the configuration namespace where you defined the logging properties

- If you define the logging properties in the bootstrap file, specify the root namespace, `"/`.

```
Log.init(configMgr, "/");
```

- If you define the logging properties in the directory, specify the namespace relative to the property `Config.net.juniper.sgmt.lib.config.staticConfigDN`, which you configure in the bootstrap file.

```
Log.init(configMgr, "/Applications/Quota");
```

### Instantiating the NIC Proxy

To enable the application to instantiate a NIC proxy:

- Call the following method:

```
NIC nicProxy = nicFactory.getNicComponent(nicNameSpace, configMgr)
```

Alternatively, if the expected data value (specified for the property `nic.value` in the NIC proxy configuration) is an SAE reference, you can call the following method:

```
SaeLocator nicProxy = nicFactory.getSaeLocator(nicNameSpace, configMgr);
```

- `nicFactory`—Instance of the NIC factory
- `nicNameSpace`—String that specifies the configuration namespace where you defined the properties for the NIC proxy
  - If you define the NIC properties in the bootstrap file, specify the root namespace, `"/`.

```
NIC nicProxy = nicFactory.getNicComponent("/", configMgr)
```

  - If you define the properties in the directory, specify the namespace relative to the property `Config.net.juniper.smgmt.lib.config.staticConfigDN`, which you specified in the bootstrap file.
 

```
NIC nicProxy = nicFactory.getNicComponent("/Applications/Quota", configMgr)
```
- `configMgr`—Instance of the configuration manager, the value returned from the `getConfigMgr()` method

### Managing a Resolution Request

To enable the application to submit a resolution request and obtain the associated values:

1. Construct a `NicKey` object to enable the application to pass the data key to the NIC proxy:

```
NicKey nicKey = new NicKey(stringKey);
```

- `stringKey`—Data key for which you want to find corresponding values.

For the syntax of allowed data types, see *Chapter 10, NIC Resolution Process*.

2. If the resolution process specifies constraints that you wish to provide in the resolution request, add them to the `NicKey` object:

```
NicKey.addConstraint(constName, constValue);
```

- `constName`—Name of the constraint.

For the allowed data types and their syntax, see *Chapter 10, NIC Resolution Process*.

- `constValue`—Specific value of the constraint.

For the allowed syntax for the data types, see *Chapter 10, NIC Resolution Process*.

3. Call a method that starts the resolution process.

For example, you can call a method specified in the NIC interface:

```
NicValue val = nicProxy.lookupSingle(nicKey);
```

Alternatively, if the expected data value is an SAE reference, you can call the following method:

```
Saeld saeld = nicProxy.lookupSae(nicKey);
```

4. Call the `getValue` method to access the string representation of the data value obtained by the NIC proxy.

```
String val=val.getValue();
```

Alternatively, if the expected data value is an SAE reference:

```
String val=saeld.getValue();
```

5. (Optional) Call a method to get intermediate values obtained during a resolution. For example, applications that use the `AssignedIP` realm can map the IP address to the SAE reference and can obtain the intermediate values of the interface identifier and virtual router (VR) name during the process. The SAE can use the interface identifier and VR name to dynamically create a subscriber session.

- Call the `getIntermediateValue` method if the application expects only one value. This method takes the name of a data type and returns as a string the first value it finds.

```
String getIntermediateValue(String dataTypeName){};
```

For information about data types, see *Chapter 10, NIC Resolution Process*.

- Call the `getIntermediateValues` or `getAllIntermediateValues` method if the application expects multiple values. These methods take the name of a data type and return values as follows:
  - The `getIntermediateValues` method returns a list of values as a string array.

```
String[] getIntermediateValues(String dataTypeName);
```

For information about data types, see *Chapter 10, NIC Resolution Process*.

- The `getAllIntermediateValues` method returns a map of all intermediate values for the request. The key for the map is the name of the network data type, and the value of the map is a string array of the intermediate values.

```
Map getAllIntermediateValues();
```

### Deleting Invalid Results from the NIC Proxy's Cache

If the application receives an exception when using values that the NIC proxy returned for a specific key, it must inform the NIC proxy to delete this entry from its cache.

To enable the application to inform the NIC proxy to delete an entry from its cache:

- Call the following method:

```
nicProxy.invalidateLookup(nicKey, nicValue);
```

- `nicKey`—Data key that you want to remove from the cache
- `nicValue`—Optional data value that corresponds to this key

If the application passes a null data value to the NIC proxy, the NIC proxy removes all the values associated with the data key from its cache.

### Removing the NIC Proxies

Make sure that before your application shuts down, it removes the NIC proxy instances to release resources for other software processes.

To remove one NIC proxy instance:

- Call the following method:

```
NicProxy.destroy();
```

To remove all NIC proxy instances, call the following method:

```
NicFactory.destroy();
```