

Chapter 14

Developing Applications That Use a NIC

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

- Overview on page 257
- Providing the NIC Proxy Files on page 258
- Developing the Application on page 258

Overview

If you are writing an external application that interacts with a NIC, you must include a NIC proxy in the application. The NIC proxy is a client-side library that communicates with a single NIC resolver through Common Object Request Broker Architecture (CORBA).

Typically, each resolution process in the NIC 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 and 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, 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.

Providing the NIC Proxy Files

If you are using Java Runtime Environment (JRE) 1.5 or higher, you must provide the following Java archive (JAR) files, which are in the SDX software distribution in the folder */SDK/lib*, with your application:

- *des.jar*
- *gateway.jar*
- *infra.jar*
- *logger.jar*
- *nicproxy.jar*
- *jmxri.jar* (use only if you want administrators to be able to monitor the NIC proxy through the JMX management interface)

If you are using JRE 1.3, you must also include the following JAR files, which are in the SDX software distribution in the folder *SDK/lib-1.3*, with your application:

- *ldap.jar*
- *ldapbp.jar*
- *jndi.jar*
- *providerutil.jar*
- *smgtutil.jar*

Developing the Application

The application must perform the following functions when it communicates with the NIC proxy:

1. Instantiate the NIC factory.
2. Instantiate the Configuration Manager.
3. (Optional) Initialize logging.
4. Instantiate a NIC proxy.
5. Submit resolution requests.
6. Delete invalid results from the NIC proxy's cache.
7. Remove the NIC proxy.

The following sections describe the API calls that the application must use to perform these functions. For more information about the API calls, see the online documentation in the SDX software distribution in the folder */SDK/doc/nic*.

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 that 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 (see *Removing the NIC Proxies* on page 263), 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.

Instantiating a Configuration Manager

The application must instantiate a configuration manager to obtain a NIC instance from the NIC factory. To enable the application to do so, 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, that contains all the configuration properties for the NIC proxy.

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.sgmt.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 17, Customizing NIC Configuration*.

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 17, Customizing NIC Configuration*.
- `constValue`—Specific value of the constraint. For the allowed syntax for the data types, see *Chapter 17, Customizing NIC Configuration*.

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 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 (see *Chapter 17, Customizing NIC Configuration*) and returns as a string the first value it finds.

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

- Call the `getIntermediateValues` or `getAllIntermediateValues` method if the application expects multiple values. These methods take the name of a data type (see *Chapter 17, Customizing NIC Configuration*) and return values as follows:
 - The `getIntermediateValues` method returns a list of values as a string array.

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

- 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 perform this function, 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();
```

