Oracle® Communications Offline Mediation Controller

Cartridge Development Kit Developer's Guide

Release 15.0

F86421-02

June 2024

Oracle Communications Offline Mediation Controller Cartridge Development Kit Developer's Guide, Release 15.0

F86421-02

Contents

Preface

This document describes the components and steps needed to develop a cartridge in the Oracle Communications Offline Mediation Controller framework using the Cartridge Development Kit (CDK) for Offline Mediation Controller 15.0.

Audience

This document is intended for developers responsible for developing cartridges and nodes that will be used within the Offline Mediation Controller product. Before reading this book, read *Offline Mediation Controller User's Guide*.

Documentation Accessibility

Diversity and Inclusion

1

CDK Design and Concepts

This chapter provides background information required for creating custom Oracle Communications Offline Mediation Controller nodes.

Overview

It is recommended that nodes perform tasks that are relatively small in scope to aid in faster processing. Two custom nodes (such as a CC node and an EP node) chained together may yield better throughput, flexibility, and validity, rather than writing a single node to achieve the same results. The most commonly customized nodes are CC and DC nodes.

The "generic" nodes that can be modified using the Cartridge Development Kit (CDK) are in the Offline Mediation Controller GUI in the "Cartridge Kit" market segment.

Life cycle of a node

Once the Offline Mediation Controller system is running, all nodes run within the Node Manager's VM. For debugging purposes, nodes can be run from the command line, provided that a main() method is implemented.

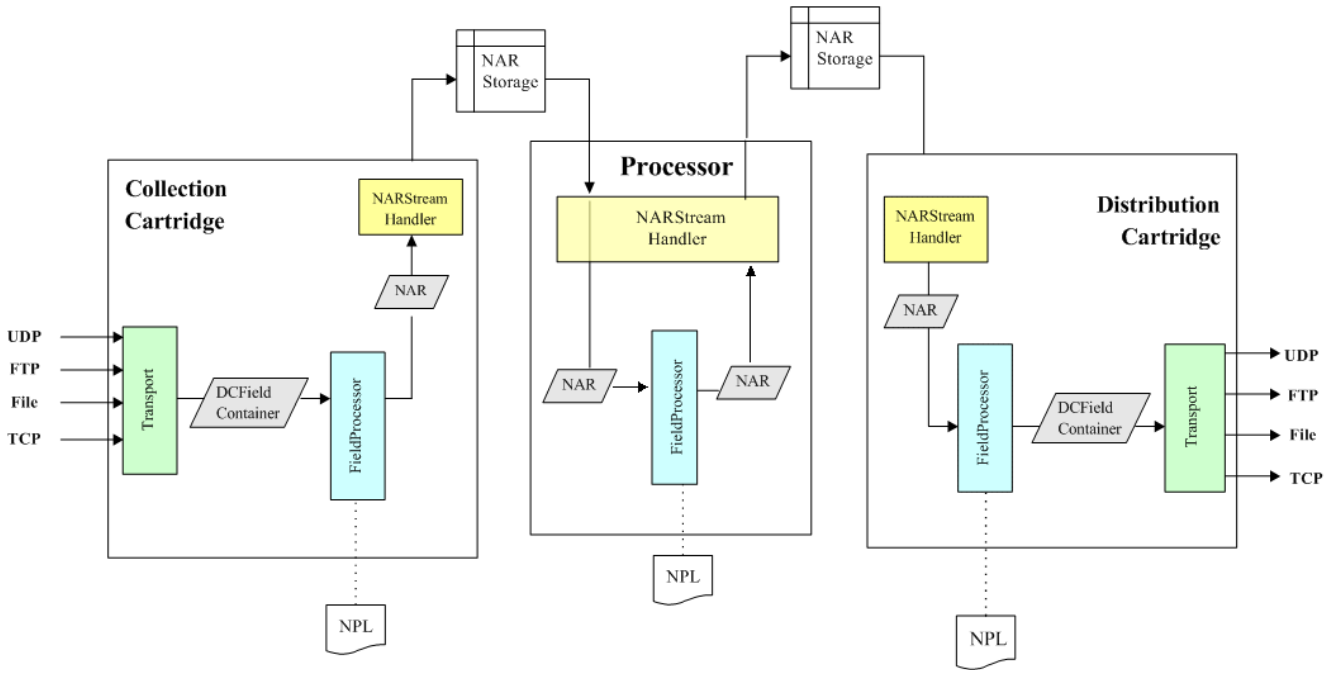
The Offline Mediation Controller system provides a mechanism for adding a node to the Node Manager and configuring and starting the node to process data via a Graphical User Interface (GUI). This process is described in more detail later in this document.

A node (object) is officially created when the user adds it to a Node Manager's node list and starts it. All necessary configuration information is captured and stored in the configuration file when the user creates the node. However, the node's constructor is not invoked until the user clicks the **Start** button in the Administration Client GUI.

Data flow diagram

Figure 1-1 shows the flow of data through any given node chain.

Figure 1-1 Data Flow Diagram



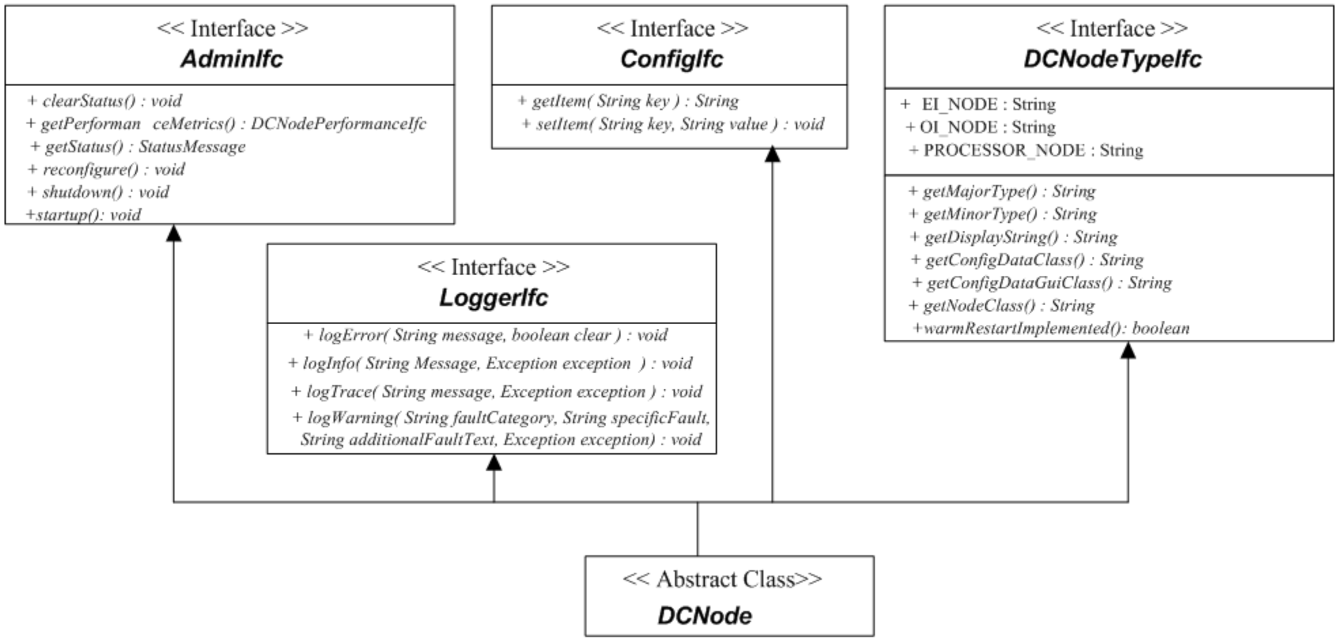
DCNode class hierarchy

Figure 1-2 and Figure 1-3 illustrate the class hierarchy of the base DCNode classes. Details of individual node components are included in later sections of this document. Methods that should be overridden are in italics.

Note:

All derivations of DCNode should provide a default (i.e., no-argument) constructor, which initializes the information necessary to implement DCNodeTypeIfc.

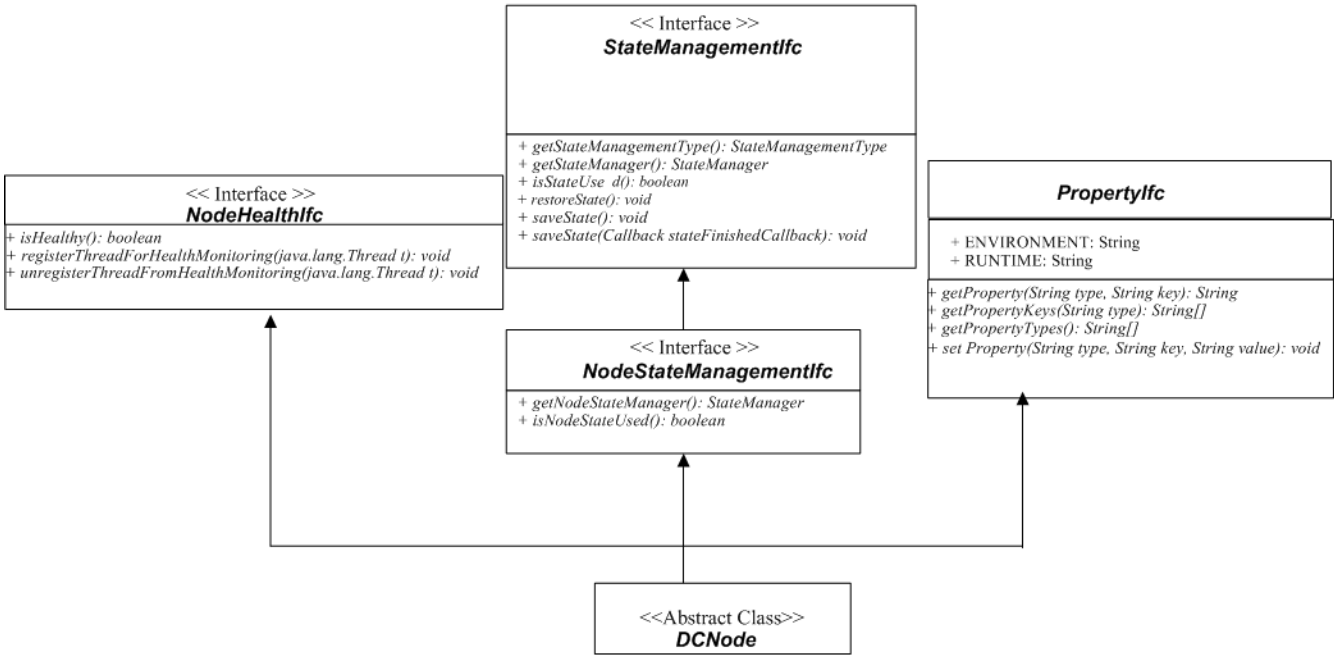
Figure 1-2   DCNode Class Hierarchy


<< Interface >>
AdminIfc
+ clearStatus() : void
+ getPerforman ceMetrics() : DCNodePerformanceIfc
+ getStatus() : StatusMessage
+ reconfigure() : void
+ shutdown() : void
+startup(): void
<< Interface >>
ConfigIfc
+ getItem(String key) : String
+ setItem(String key, String value) : void
<< Interface >>
DCNodeTypeIfc
+ EI_NODE : String
+ OI_NODE : String
+ PROCESSOR_NODE : String
+ getMajorType() : String
+ getMinorType() : String
+ getDisplayString() : String
+ getConfigDataClass() : String
+ getConfigDataGuiClass() : String
+ getNodeClass() : String
+warmRestartImplemented(): boolean
<< Interface >>
LoggerIfc
+ logError(String message, boolean clear) : void
+ logInfo(String Message, Exception exception) : void
+ logTrace(String message, Exception exception) : void
+ logWarning(String faultCategory, String specificFault, String additionalFaultText, Exception exception) : void
<< Abstract Class>>
DCNode

Note:

The StateManagement and NodeStateManagement interfaces are currently not supported.

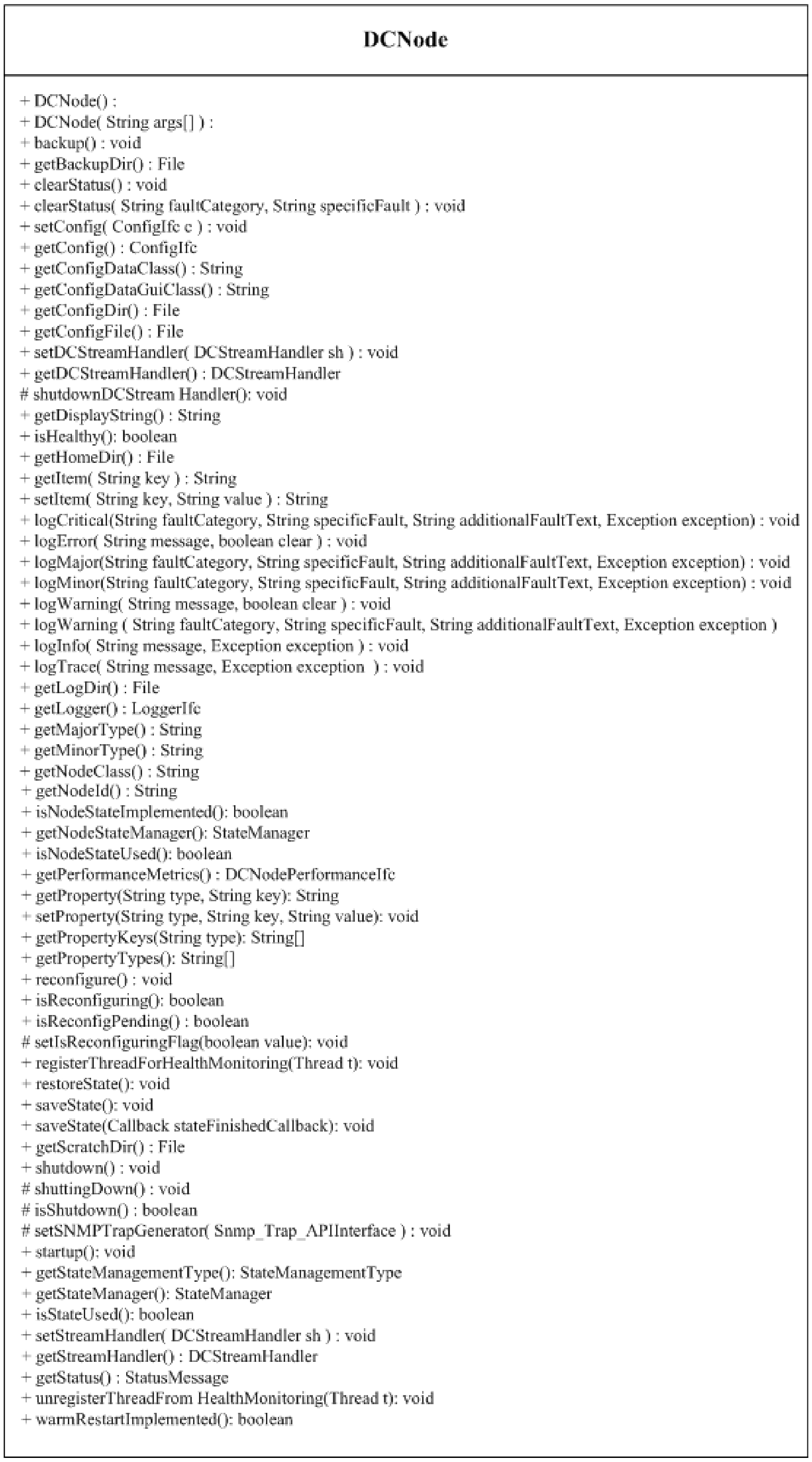
Figure 1-3   DCNode Class Hierarchy


<< Interface >>
StateManagementIfc
+ getStateManagementType(): StateManagementType
+ getStateManager(): StateManager
+ isStateUse d(): boolean
+ restoreState(): void
+ saveState(): void
+ saveState(Callback stateFinishedCallback): void
<< Interface >>
NodeHealthIfc
+ isHealthy(): boolean
+ registerThreadForHealthMonitoring(java.lang.Thread t): void
+ unregisterThreadFromHealthMonitoring(java.lang.Thread t): void
PropertyIfc
+ ENVIRONMENT: String
+ RUNTIME: String
+ getProperty(String type, String key): String
+ getPropertyKeys(String type): String[]
+ getPropertyTypes(): String[]
+ set Property(String type, String key, String value): void
<< Interface >>
NodeStateManagementIfc
+ getNodeStateManager(): StateManager
+ isNodeStateUsed(): boolean
<<Abstract Class>>
DCNode

DCNode class diagram

Figure 1-4 illustrates the DCNode classes.

Figure 1-4 DCNode Class Diagram

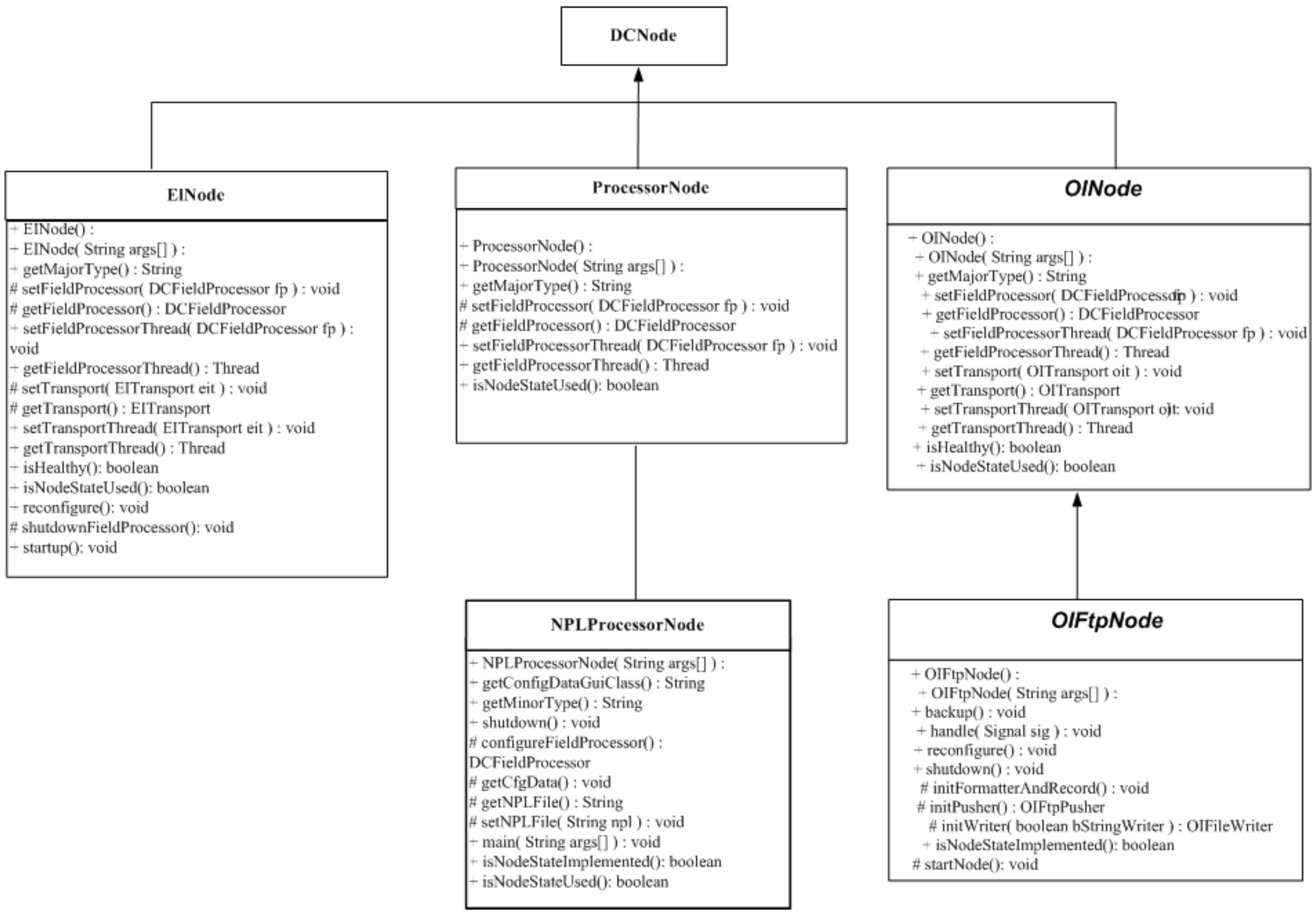


DCNode

```
+ DCNode() :
+ DCNode( String args[] ) :
+ backup() : void
+ getBackupDir() : File
+ clearStatus() : void
+ clearStatus( String faultCategory, String specificFault ) : void
+ setConfig( ConfigIfc c ) : void
+ getConfig() : ConfigIfc
+ getConfigDataClass() : String
+ getConfigDataGuiClass() : String
+ getConfigDir() : File
+ getConfigFile() : File
+ setDCStreamHandler( DCStreamHandler sh ) : void
+ getDCStreamHandler() : DCStreamHandler
# shutdownDCStream Handler(): void
+ getDisplayString() : String
+ isHealthy(): boolean
+ getHomeDir() : File
+ getItem( String key ) : String
+ setItem( String key, String value ) : String
+ logCritical(String faultCategory, String specificFault, String additionalFaultText, Exception exception) : void
+ logError( String message, boolean clear ) : void
+ logMajor(String faultCategory, String specificFault, String additionalFaultText, Exception exception) : void
+ logMinor(String faultCategory, String specificFault, String additionalFaultText, Exception exception) : void
+ logWarning( String message, boolean clear ) : void
+ logWarning ( String faultCategory, String specificFault, String additionalFaultText, Exception exception )
+ logInfo( String message, Exception exception ) : void
+ logTrace( String message, Exception exception  ) : void
+ getLogDir() : File
+ getLogger() : LoggerIfc
+ getMajorType() : String
+ getMinorType() : String
+ getNodeClass() : String
+ getNodeId() : String
+ isNodeStateImplemented(): boolean
+ getNodeStateManager(): StateManager
+ isNodeStateUsed(): boolean
+ getPerformanceMetrics() : DCNodePerformanceIfc
+ getProperty(String type, String key): String
+ setProperty(String type, String key, String value): void
+ getPropertyKeys(String type): String[]
+ getPropertyTypes(): String[]
+ reconfigure() : void
+ isReconfiguring(): boolean
+ isReconfigPending() : boolean
# setIsReconfiguringFlag(boolean value): void
+ registerThreadForHealthMonitoring(Thread t): void
+ restoreState(): void
+ saveState(): void
+ saveState(Callback stateFinishedCallback): void
+ getScratchDir() : File
+ shutdown() : void
# shuttingDown() : void
# isShutdown() : boolean
# setSNMPTrapGenerator( Snmp_Trap_APIInterface ) : void
+ startup(): void
+ getStateManagementType(): StateManagementType
+ getStateManager(): StateManager
+ isStateUsed(): boolean
+ setStreamHandler( DCStreamHandler sh ) : void
+ getStreamHandler() : DCStreamHandler
+ getStatus() : StatusMessage
+ unregisterThreadFrom HealthMonitoring(Thread t): void
+ warmRestartImplemented(): boolean
```

DCNode class hierarchy

Figure 1-5 DCNode Class Hierarchy


DCNode
ElNode
+ ElNode() :
+ ElNode(String args[]) :
+ getMajorType() : String
# setFieldProcessor(DCFieldProcessor fp) : void
# getFieldProcessor() : DCFieldProcessor
+ setFieldProcessorThread(DCFieldProcessor fp) : void
+ getFieldProcessorThread() : Thread
# setTransport(ElTransport eit) : void
# getTransport() : ElTransport
+ setTransportThread(ElTransport eit) : void
+ getTransportThread() : Thread
+ isHealthy(): boolean
+ isNodeStateUsed(): boolean
+ reconfigure(): void
# shutdownFieldProcessor(): void
+ startup(): void
ProcessorNode
+ ProcessorNode() :
+ ProcessorNode(String args[]) :
+ getMajorType() : String
# setFieldProcessor(DCFieldProcessor fp) : void
# getFieldProcessor() : DCFieldProcessor
+ setFieldProcessorThread(DCFieldProcessor fp) : void
+ getFieldProcessorThread() : Thread
+ isNodeStateUsed(): boolean
OINode
+ OINode() :
+ OINode(String args[]) :
+ getMajorType() : String
+ setFieldProcessor(DCFieldProcessor fp) : void
+ getFieldProcessor() : DCFieldProcessor
+ setFieldProcessorThread(DCFieldProcessor fp) : void
+ getFieldProcessorThread() : Thread
+ setTransport(OITransport oit) : void
+ getTransport() : OITransport
+ setTransportThread(OITransport oit: void
+ getTransportThread() : Thread
+ isHealthy(): boolean
+ isNodeStateUsed(): boolean
NPLProcessorNode
+ NPLProcessorNode(String args[]) :
+ getConfigDataGuiClass() : String
+ getMinorType() : String
+ shutdown() : void
# configureFieldProcessor() : DCFieldProcessor
# getCfgData() : void
# getNPLFile() : String
# setNPLFile(String npl) : void
+ main(String args[]) : void
+ isNodeStateImplemented(): boolean
+ isNodeStateUsed(): boolean
OIFtpNode
+ OIFtpNode() :
+ OIFtpNode(String args[]) :
+ backup() : void
+ handle(Signal sig) : void
+ reconfigure() : void
+ shutdown() : void
# initFormatterAndRecord() : void
# initPusher() : OIFtpPusher
# initWriter(boolean bStringWriter) : OIFileWriter
+ isNodeStateImplemented(): boolean
# startNode(): void

DataProvider and DataReceiver

The object-level representation of the data flow through a node can be expressed as a Data Provider and/or a Data Receiver.

DataProvider

A Data Provider is a component that provides data to another component within the node. The DataProviderIfc interface defines this component, which provides methods that allow data to flow to another component within the node.

DataReceiver

A Data Receiver accepts data from another component within the node. A Data Receiver is defined by the DataReceiverIfc interface which provides methods that allow data to be obtained from another component within the node.

Relationship between DataProvider and DataReceiver

Data may flow through a node using either a "push" or "pull" algorithm. It is up to the individual node developer to determine which algorithm is used within the node. If data comes in very slowly, the developer may want to"push" the data from the DataProvider to the DataReceiver, as it becomes available. Conversely, a DataReceiver may "pull" data from its DataProvider by periodically asking the DataProvider if data is available and retrieving it for processing. Both of these situations are illustrated below.

DataProvider Push

First, the DataProvider's DataReceiver must be set. In this case, **transport** (an EITransport) is the DataProvider and **fp** (NPLFieldProcessor) is the DataReceiver.

```
·// Set up the DataReceiver/DataProvider mappings
·transport.setDataReceiver( fp );
```

Then, the DataProvider calls the DataReceiver's processData() method directly.

```
·    MyEIRecord eiRecord = new MyEIRecord( data );
·
·    try
·    {
·        // Give this record to the data provider for
·        // further processing.
·        getDataReceiver().processData( eiRecord );
·    }
·    catch( Exception e )
·    {
·        System.out.println( "Processing error: " + e.getMessage() );
·        getLogger().logError( "Error processing record: " + e, false );
·    }
```

Another option would be for the DataProvider to call the DataReceiver's **dataIsAvailable()** method, and then the DataReceiver would make a call back to the DataProvider's **getData()** method to retrieve the data. Again, the **setDataReceiver()** method would need to be called first.

Example 1-1   Push using getData()

```
·    public void dataIsAvailable()
·    {
·        dataAvailable = true;
·
·        DataProviderIfc provider = getDataProvider();
·
·        if( provider != null )
·        {
·            DCFieldContainer data = getDataProvider().getData();
·
·            while( data != null )
·            {
·                try
·                {
·                    processData( data );
·                }
·                catch( NodeProcessingException npe )
·                {
·                    // Do something…
·                   }
·                   data = provider.getData();
·               }
·           }
·
·           dataAvailable = false;
·       }
```

DataReceiver Pull

In a “pull" scenario, it is likely that the data is coming in quickly and that the DataProvider will utilize some sort of queue in order to store the data temporarily until the DataReceiver is ready for it. This setup is particularly useful for CCs which are using real-time (or near real-time) transports, such as UDP or TCP, and need to be able to collect the data as it comes in “off the line".

The DataProvider must first be set using the setDataProvider() method, similar to the above call to setDataReceiver() in the “push" scenario.

The DataReceiver would probably be running in a thread, and do the following as part of the run() method:

Example 1-2 Implementing DataReceiver Pull

```
·     while( running )
·     {
·         while( getDataProvider().isDataAvailable() )
·         {
·             DCFieldContainer data = getDataProvider().getData();
·             try
·             {
·                 processData( data );
·             }
·             catch( NodeProcessingException npe )
·             {
·                 System.out.println( "NodeProcessingException caught while " +
·                              "processing record.\n    " + npe.getMessage() );
·             }
·         }
·         if( running )
·         {
·             try
·             {
·                 Thread.currentThread().sleep( 2000 );
·             }
·             catch( InterruptedException ie ) { }
·         }
·     }
```

Transport

A Transport is responsible for moving data into and out of the Offline Mediation Controller system. Transports are only associated with CC or DC nodes.

EITransport

An EITransport is a DataProvider that accepts data from outside of the Offline Mediation Controller system and creates the appropriate EIRecord objects for that data. Therefore, the EITransport needs to know the record delimiter, as well as the EIRecord object to be populated from each raw data record. An EITransport may contain a factory (described later), which would be responsible for creating the appropriate EIRecord objects when needed, or receive this class type information as a parameter. In cases where data may be coming in rapidly (for example, via UDP packets in a CC node), the EITransport could utilize a revolving queue to store the incoming records until the DataReceiver is ready for them.

OITransport

An OITransport is a DataReceiver that receives OIRecord objects specific to the destination of the data and is responsible for transmitting that data to the desired destination. The OITransport will extract the formatted data from the OIRecord and send the data out of the Offline Mediation Controller system via the appropriate medium. The OITransport will typically receive its data from a “push". It is bound by the frequency of the incoming data and the speed of its DataProvider.

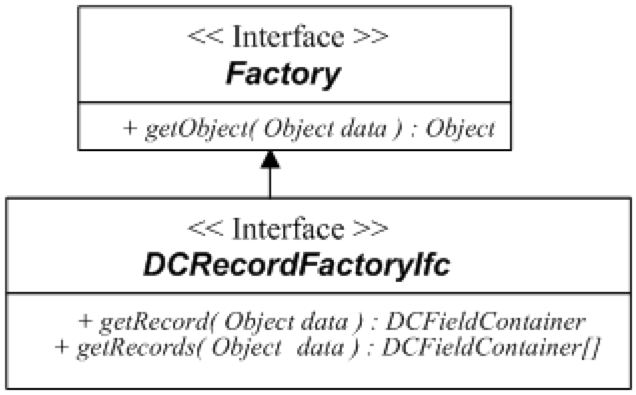
DCRecordFactoryIfc

The DCRecordFactoryIfc is a generic interface that can be used which will accept data and generate the appropriate EIRecord or OIRecord objects. This allows a Transport or FieldProcessor to behave in a generic way without having to explicitly know what type of EIRecord or OIRecord object that needs to be created based on the incoming or outgoing data.

The only time a user of the CDK would need to write their own DCRecordFactoryIfc would be if they intended to use an existing transport with a custom EIRecord.

Figure 1-6 shows the DCRecordFactoryIfc interface.

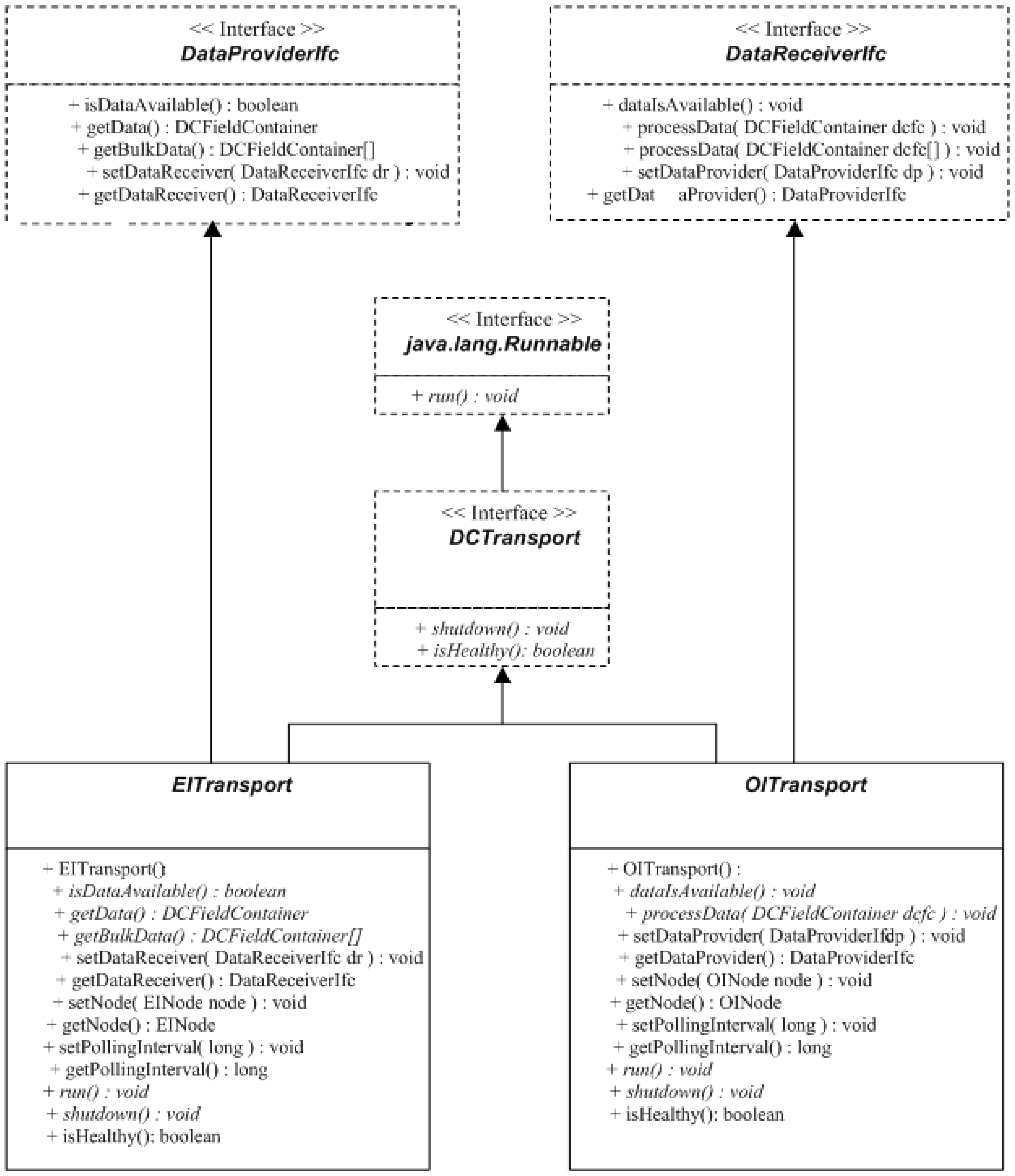
Figure 1-6   DCRecordFactoryIfc


<< Interface >>
Factory
+ getObject(Object data) : Object
<< Interface >>
DCRecordFactoryIfc
+ getRecord(Object data) : DCFieldContainer
+ getRecords(Object data) : DCFieldContainer[]

Transport class hierarchy

Figure 1-7 shows the hierarchy of the Transport class.

Figure 1-7 Transport Class Hierarchy


<< Interface >>
DataProviderIfc
+ isDataAvailable() : boolean
+ getData() : DCFieldContainer
+ getBulkData() : DCFieldContainer[]
+ setDataReceiver(DataReceiverIfc dr) : void
+ getDataReceiver() : DataReceiverIfc
<< Interface >>
DataReceiverIfc
+ dataIsAvailable() : void
+ processData(DCFieldContainer dcfc) : void
+ processData(DCFieldContainer dcfc[]) : void
+ setDataProvider(DataProviderIfc dp) : void
+ getDataProvider() : DataProviderIfc
<< Interface >>
java.lang.Runnable
+ run() : void
<< Interface >>
DCTransport
+ shutdown() : void
+ isHealthy(): boolean
EITransport
+ EITransport()
+ isDataAvailable() : boolean
+ getData() : DCFieldContainer
+ getBulkData() : DCFieldContainer[]
+ setDataReceiver(DataReceiverIfc dr) : void
+ getDataReceiver() : DataReceiverIfc
+ setNode(EINode node) : void
+ getNode() : EINode
+ setPollingInterval(long) : void
+ getPollingInterval() : long
+ run() : void
+ shutdown() : void
+ isHealthy(): boolean
OITransport
+ OITransport() :
+ dataIsAvailable() : void
+ processData(DCFieldContainer dcfc) : void
+ setDataProvider(DataProviderIfc dp) : void
+ getDataProvider() : DataProviderIfc
+ setNode(OINode node) : void
+ getNode() : OINode
+ setPollingInterval(long) : void
+ getPollingInterval() : long
+ run() : void
+ shutdown() : void
+ isHealthy(): boolean

FieldProcessor

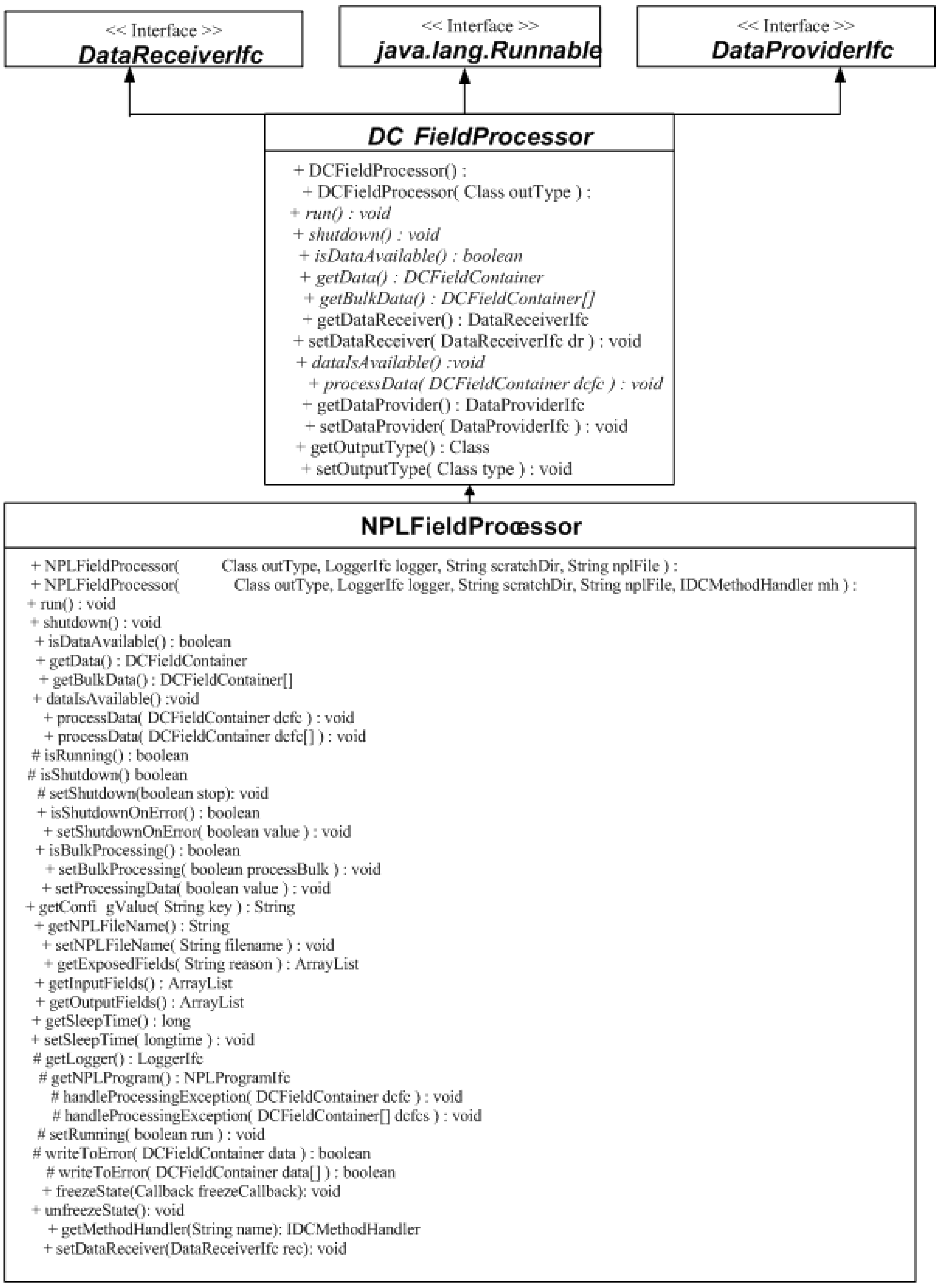
There is an NPLFieldProcessor provided, which should be used by all nodes. This class uses information contained in an ASCII text file, which contains NPL commands. (See the *NPL Reference Guide* for more information). The FieldProcessor utilizes the information in the NPL file to transform, translate, enhance, and/or route information from the incoming

DCFieldContainer object into the appropriate outgoing DCFieldContainer object. A FieldProcessor is both a DataProvider and a DataReceiver.

FieldProcessor class hierarchy

Figure 1-8 shows the FieldProcessor class hierarchy.

Figure 1-8 FieldProcessor Class Hierarchy


<< Interface >>
DataReceiverIfc
<< Interface >>
java.lang.Runnable
<< Interface >>
DataProviderIfc
DC FieldProcessor
+ DCFieldProcessor() :
+ DCFieldProcessor(Class outType) :
+ run() : void
+ shutdown() : void
+ isDataAvailable() : boolean
+ getData() : DCFieldContainer
+ getBulkData() : DCFieldContainer[]
+ getDataReceiver() : DataReceiverIfc
+ setDataReceiver(DataReceiverIfc dr) : void
+ dataIsAvailable() :void
+ processData(DCFieldContainer dcfc) : void
+ getDataProvider() : DataProviderIfc
+ setDataProvider(DataProviderIfc) : void
+ getOutputType() : Class
+ setOutputType(Class type) : void
NPLFieldProcessor
+ NPLFieldProcessor(Class outType, LoggerIfc logger, String scratchDir, String nplFile) :
+ NPLFieldProcessor(Class outType, LoggerIfc logger, String scratchDir, String nplFile, IDCMethodHandler mh) :
+ run() : void
+ shutdown() : void
+ isDataAvailable() : boolean
+ getData() : DCFieldContainer
+ getBulkData() : DCFieldContainer[]
+ dataIsAvailable() :void
+ processData(DCFieldContainer dcfc) : void
+ processData(DCFieldContainer dcfc[]) : void
# isRunning() : boolean
# isShutdown() boolean
# setShutdown(boolean stop): void
+ isShutdownOnError() : boolean
+ setShutdownOnError(boolean value) : void
+ isBulkProcessing() : boolean
+ setBulkProcessing(boolean processBulk) : void
+ setProcessingData(boolean value) : void
+ getConfi gValue(String key) : String
+ getNPLFileName() : String
+ setNPLFileName(String filename) : void
+ getExposedFields(String reason) : ArrayList
+ getInputFields() : ArrayList
+ getOutputFields() : ArrayList
+ getSleepTime() : long
+ setSleepTime(longtime) : void
# getLogger() : LoggerIfc
# getNPLProgram() : NPLProgramIfc
# handleProcessingException(DCFieldContainer dcfc) : void
# handleProcessingException(DCFieldContainer[] dcfcs) : void
# setRunning(boolean run) : void
# writeToError(DCFieldContainer data) : boolean
# writeToError(DCFieldContainer data[]) : boolean
+ freezeState(Callback freezeCallback): void
+ unfreezeState(): void
+ getMethodHandler(String name): IDCMethodHandler
+ setDataReceiver(DataReceiverIfc rec): void

DCFieldContainer

The DCFieldContainer is a generic interface that is used for passing the data within and between nodes. A DCFieldContainer represents a single record of data. It is responsible for converting data from its raw format (for example from UDP packets, ASCII, or binary files) and extracting specific pieces as fields for use by the DataReceiver.

The DCFieldContainer is where most of the work is done for a particular node. This class is responsible for knowing the format and order of the incoming (or outgoing) data, and how to extract (or assemble) that data on a per field basis.

Adapter classes are provided (DCEIRecordAdapter and DCOIRecordAdapter), which “no-op" those methods that are not needed based on the type of DCFieldContainer. That is, a DCFieldContainer in a CC node would most likely use only the get… methods from DCFieldContainer for extracting data fields. Conversely, a DCFieldContainer in a DC node would most likely use only the set… methods.

EIRecord

EIRecord is derived from DCEIRecordAdapter and is the class that developers should derive from when developing a DCFieldContainer object for use in a CC node.

OIRecord

OIRecord is derived from DCOIRecordAdapter and is the class that developers should derive from when developing a DCFieldContainer object for use in a DC node. Note that derivations of OIRecord should include a default (i.e. no-argument) constructor in order to be utilized properly by the NPLFieldProcessor.

EIRecord and OIRecord Storage

Both the EIRecord and OIRecord classes are responsible for storing attributes. The various field functions will pass attribute IDs to distinguish between the attributes. These IDs originate from the NPL, and may be either the attribute values or names, depending on which is used in the NPL. The EIRecord and OIRecord should refrain from hard coding the possible values, as the NPL can be independently changed. Instead, more generic means, such as hash tables, should be used for storage and retrieval of these values.

NAR

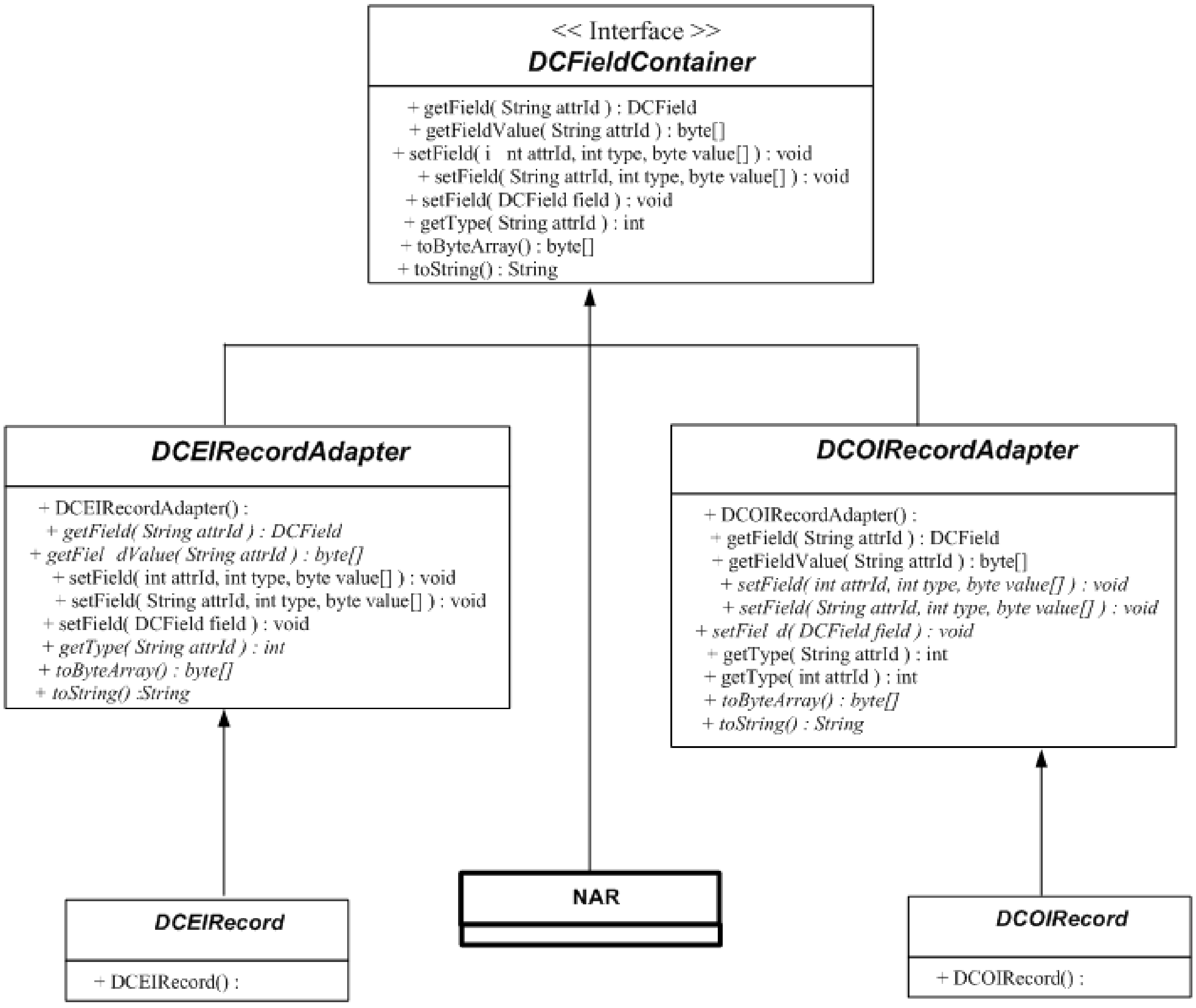
As mentioned previously, the NAR is the DCFieldContainer object that is used internally within the Offline Mediation Controller system. The FieldProcessor of a CC node generates a NAR, which then may be passed through one or more processor nodes for further modification and is received by the FieldProcessor in an DC node.

A Data Dictionary provides definitions for the type of data in the fields (potentially) within a NAR. This information is currently available in a text file, located at *OMC_Home*/datadict/Data_Dictionary, where *OMC_Home* is the directory in which you installed Offline Mediation Controller.

DCFieldContainer class hierarchy

Figure 1-9 shows the DCFieldContainer class hierarchy.

Figure 1-9 DCFieldContainer Class Hierarchy

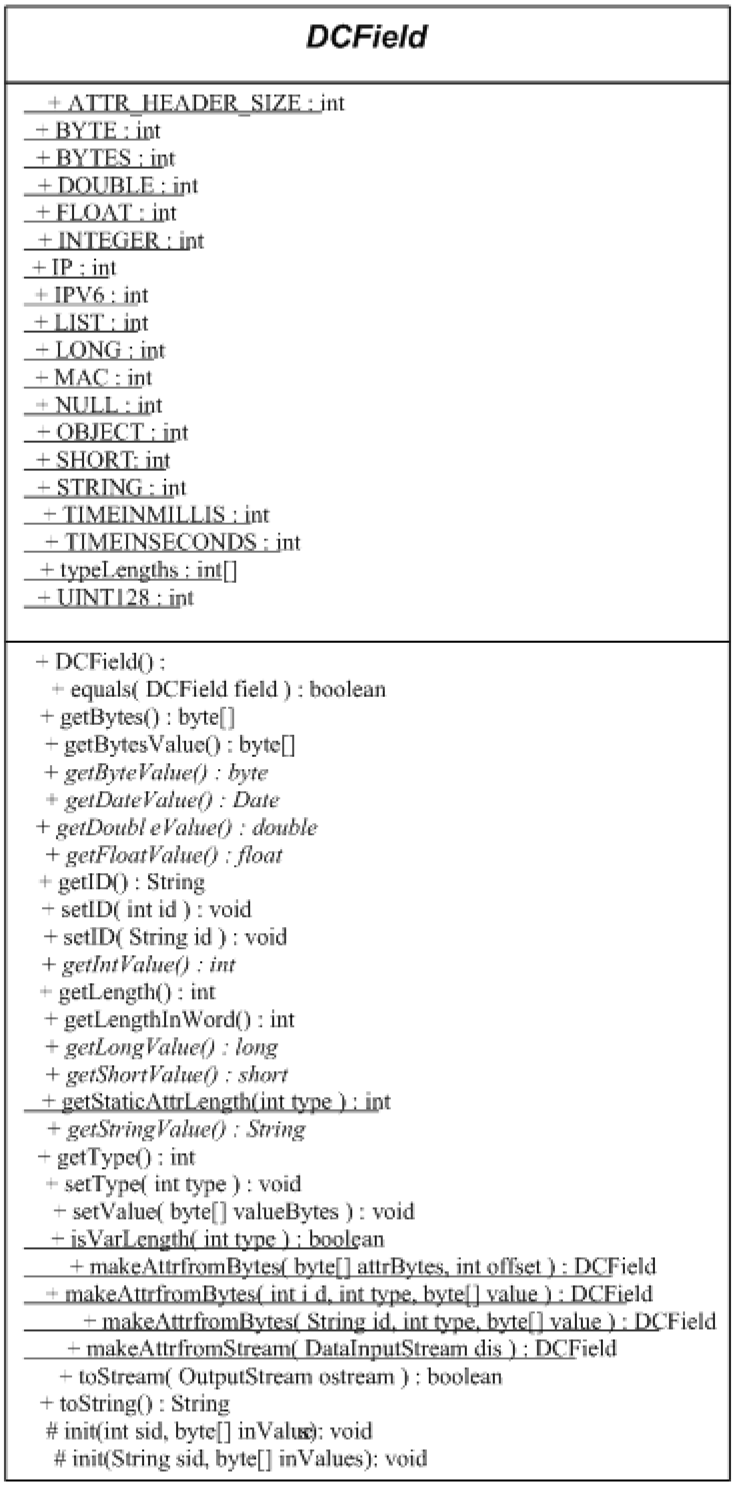

<< Interface >>
DCFieldContainer
+ getField(String attrId) : DCField
+ getFieldValue(String attrId) : byte[]
+ setField(i nt attrId, int type, byte value[]) : void
+ setField(String attrId, int type, byte value[]) : void
+ setField(DCField field) : void
+ getType(String attrId) : int
+ toByteArray() : byte[]
+ toString() : String
DCEIRecordAdapter
+ DCEIRecordAdapter() :
+ getField(String attrId) : DCField
+ getFiel dValue(String attrId) : byte[]
+ setField(int attrId, int type, byte value[]) : void
+ setField(String attrId, int type, byte value[]) : void
+ setField(DCField field) : void
+ getType(String attrId) : int
+ toByteArray() : byte[]
+ toString() :String
DCOIRecordAdapter
+ DCOIRecordAdapter() :
+ getField(String attrId) : DCField
+ getFieldValue(String attrId) : byte[]
+ setField(int attrId, int type, byte value[]) : void
+ setField(String attrId, int type, byte value[]) : void
+ setFiel d(DCField field) : void
+ getType(String attrId) : int
+ getType(int attrId) : int
+ toByteArray() : byte[]
+ toString() : String
DCEIRecord
+ DCEIRecord() :
NAR
DCOIRecord
+ DCOIRecord() :

DCField

A DCField is an object that represents the attributes (fields) of a particular record. DCField objects are stored in a DCFieldContainer. Most data types can be represented with the DCField classes that are provided.

Figure 1-10 shows the DCField object and attributes.

Figure 1-10 DCField

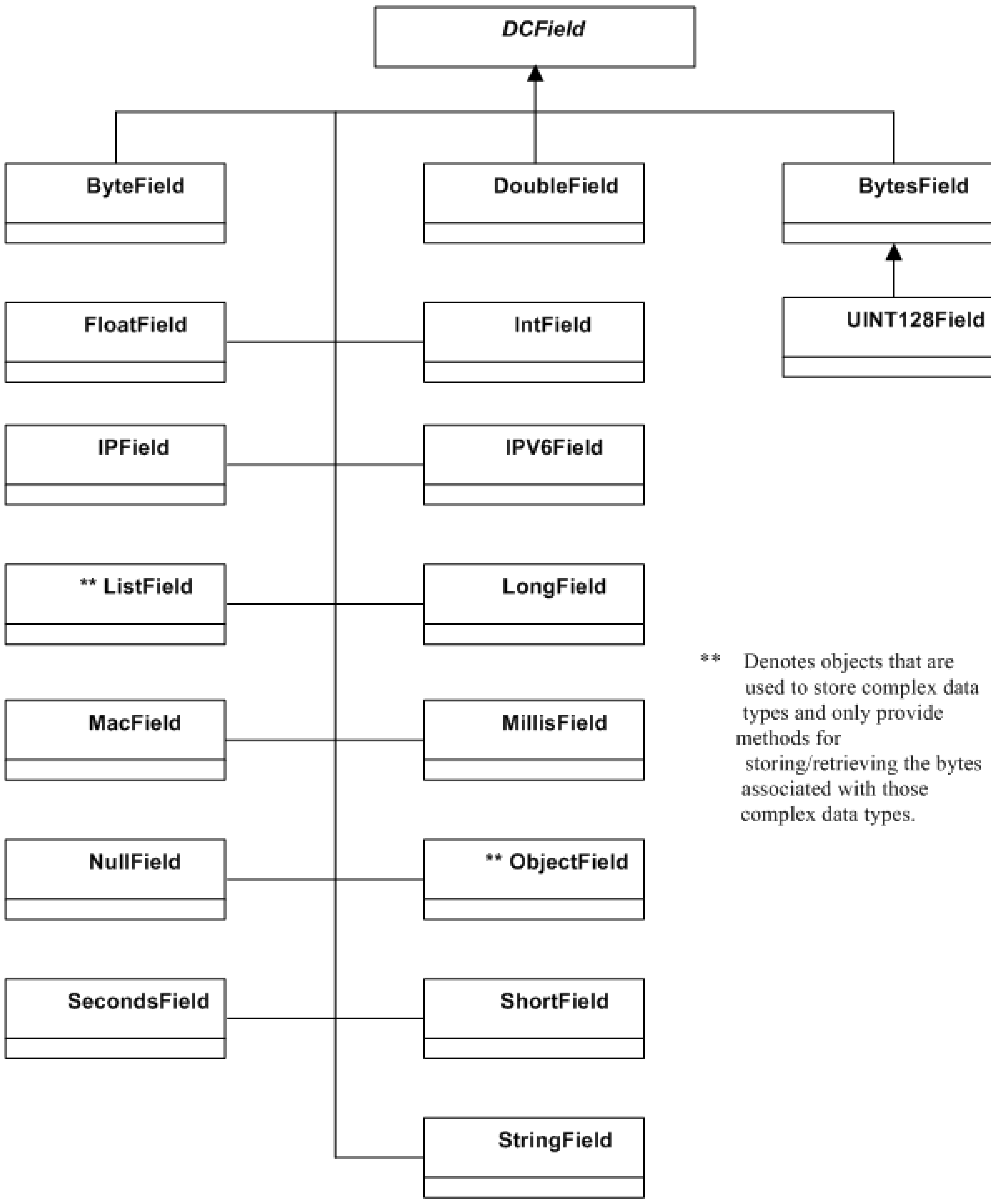


```
DCField

+ ATTR_HEADER_SIZE : int
+ BYTE : int
+ BYTES : int
+ DOUBLE : int
+ FLOAT : int
+ INTEGER : int
+ IP : int
+ IPV6 : int
+ LIST : int
+ LONG : int
+ MAC : int
+ NULL : int
+ OBJECT : int
+ SHORT: int
+ STRING : int
+ TIMEINMILLIS : int
+ TIMEINSECONDS : int
+ typeLengths : int[]
+ UINT128 : int

+ DCField() :
+ equals( DCField field ) : boolean
+ getBytes() : byte[]
+ getBytesValue() : byte[]
+ getByteValue() : byte
+ getDateValue() : Date
+ getDoubl eValue() : double
+ getFloatValue() : float
+ getID() : String
+ setID( int id ) : void
+ setID( String id ) : void
+ getIntValue() : int
+ getLength() : int
+ getLengthInWord() : int
+ getLongValue() : long
+ getShortValue() : short
+ getStaticAttrLength(int type ) : int
+ getStringValue() : String
+ getType() : int
+ setType( int type ) : void
+ setValue( byte[] valueBytes ) : void
+ isVarLength( int type ) : boolean
+ makeAttrfromBytes( byte[] attrBytes, int offset ) : DCField
+ makeAttrfromBytes( int i d, int type, byte[] value ) : DCField
+ makeAttrfromBytes( String id, int type, byte[] value ) : DCField
+ makeAttrfromStream( DataInputStream dis ) : DCField
+ toStream( OutputStream ostream ) : boolean
+ toString() : String
# init(int sid, byte[] inValues): void
# init(String sid, byte[] inValues): void
```

DCField class hierarchy

Figure 1-11 shows DCField class hierarchy.

Figure 1-11 DCField Class Hierarchy



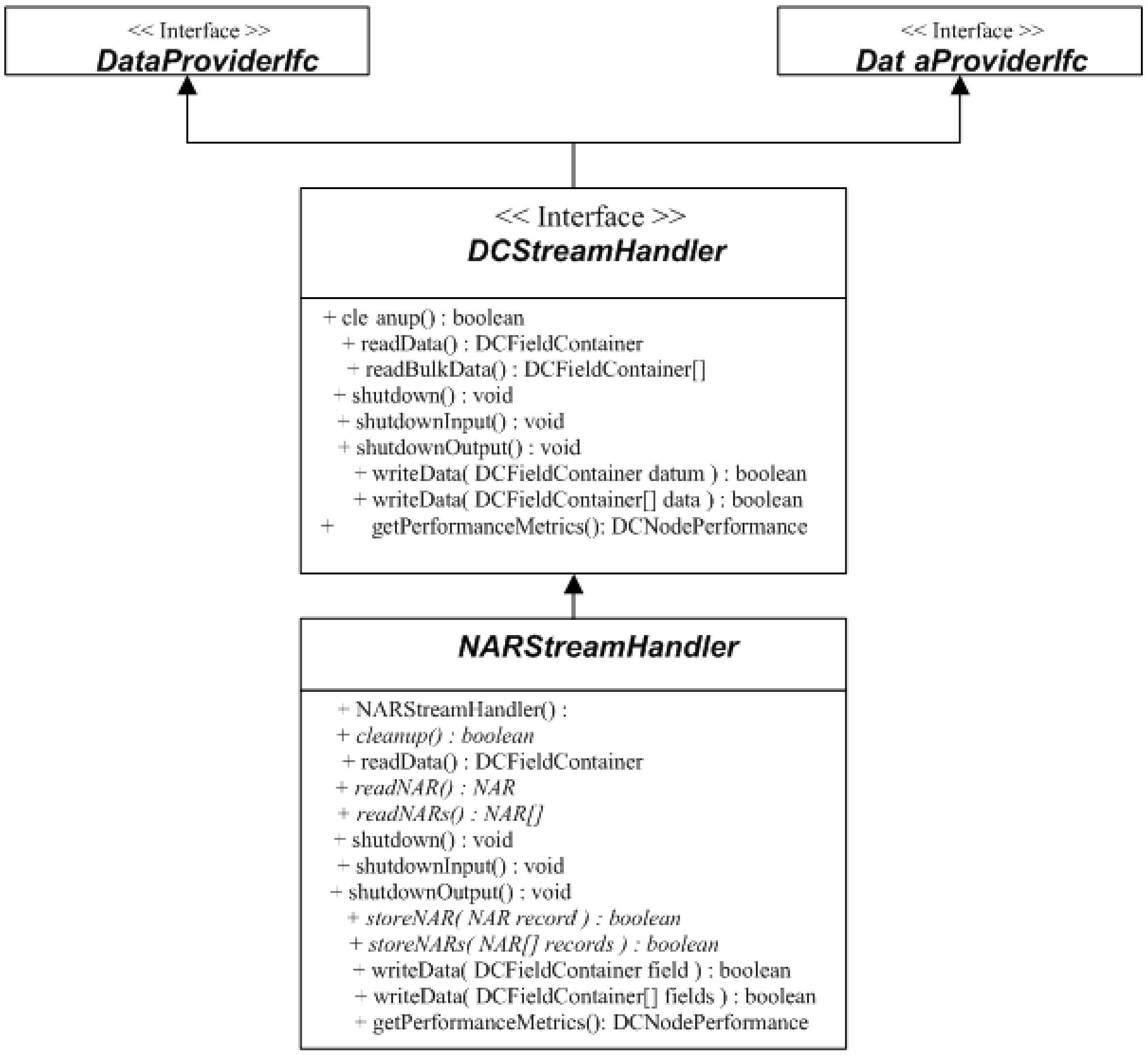
DCStreamHandler

The DCStreamHandler provides functionality to move data records from one node to the next in the node chain. The current implementation of the Offline Mediation Controller system utilizes a NARFileManager. The base node classes provide this object, and the moving of the data is transparent to the user and developer. In other words, a node developer (in general) does not need to be concerned about getting data from one node to another, this is all handled by the system. Developers only need to be concerned with moving data inside the node.

DCStreamHandler class hierarchy

Figure 1-12 shows the DCStreamHandler class hierarchy.

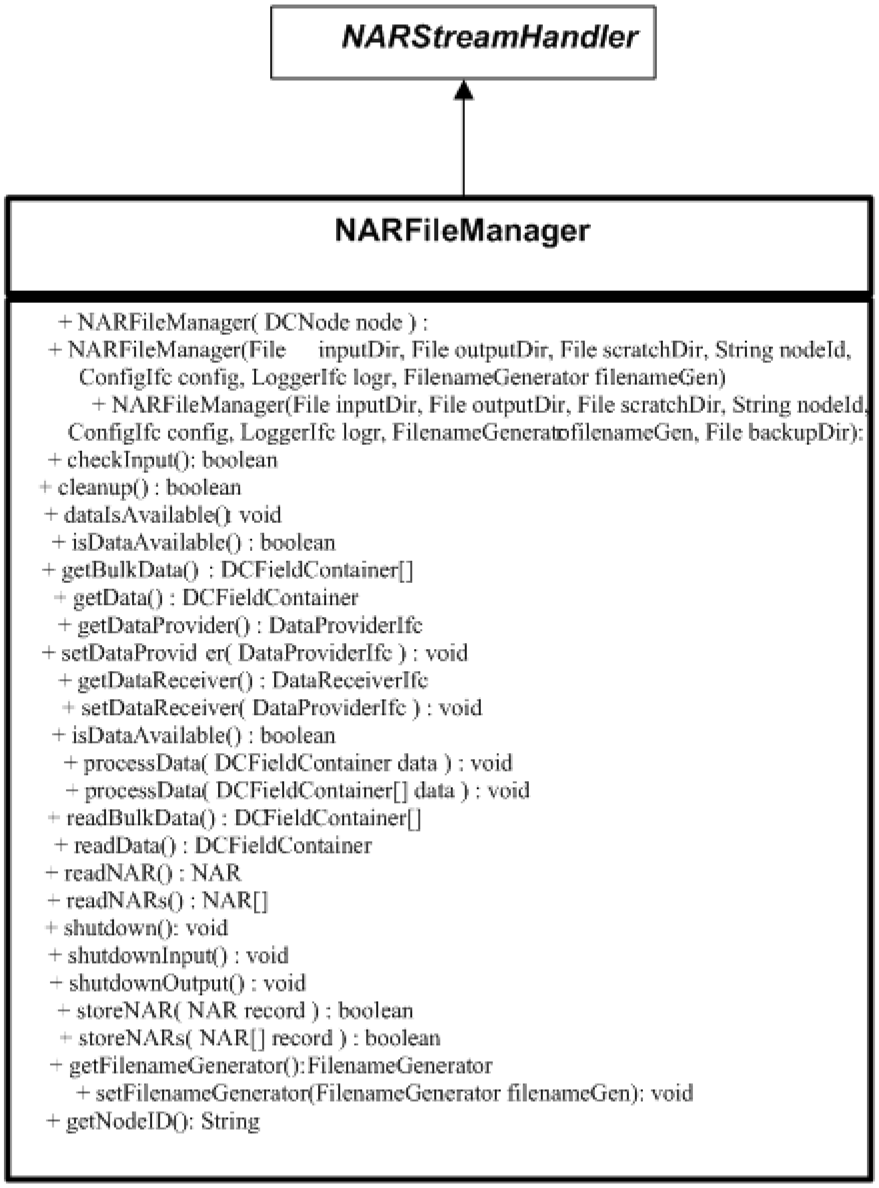
Figure 1-12 DCStreamHandler Class Hierarchy


<< Interface >>
DataProviderIfc
<< Interface >>
Dat aProviderIfc
<< Interface >>
DCStreamHandler
+ cle anup() : boolean
+ readData() : DCFieldContainer
+ readBulkData() : DCFieldContainer[]
+ shutdown() : void
+ shutdownInput() : void
+ shutdownOutput() : void
+ writeData(DCFieldContainer datum) : boolean
+ writeData(DCFieldContainer[] data) : boolean
+ getPerformanceMetrics(): DCNodePerformance
NARStreamHandler
+ NARStreamHandler() :
+ cleanup() : boolean
+ readData() : DCFieldContainer
+ readNAR() : NAR
+ readNARs() : NAR[]
+ shutdown() : void
+ shutdownInput() : void
+ shutdownOutput() : void
+ storeNAR(NAR record) : boolean
+ storeNARs(NAR[] records) : boolean
+ writeData(DCFieldContainer field) : boolean
+ writeData(DCFieldContainer[] fields) : boolean
+ getPerformanceMetrics(): DCNodePerformance

DCStreamHandler class hierarchy (continued)

Figure 1-13 shows the NARStreamHandler class.

Figure 1-13 NARStream Handler


NARStreamHandler
NARFileManager
+ NARFileManager(DCNode node) :
+ NARFileManager(File inputDir, File outputDir, File scratchDir, String nodeId, ConfigIfc config, LoggerIfc logr, FilenameGenerator filenameGen)
+ NARFileManager(File inputDir, File outputDir, File scratchDir, String nodeId, ConfigIfc config, LoggerIfc logr, FilenameGeneratorfilenameGen, File backupDir):
+ checkInput(): boolean
+ cleanup() : boolean
+ dataIsAvailable(): void
+ isDataAvailable() : boolean
+ getBulkData() : DCFieldContainer[]
+ getData() : DCFieldContainer
+ getDataProvider() : DataProviderIfc
+ setDataProvid er(DataProviderIfc) : void
+ getDataReceiver() : DataReceiverIfc
+ setDataReceiver(DataProviderIfc) : void
+ isDataAvailable() : boolean
+ processData(DCFieldContainer data) : void
+ processData(DCFieldContainer[] data) : void
+ readBulkData() : DCFieldContainer[]
+ readData() : DCFieldContainer
+ readNAR() : NAR
+ readNARs() : NAR[]
+ shutdown(): void
+ shutdownInput() : void
+ shutdownOutput() : void
+ storeNAR(NAR record) : boolean
+ storeNARs(NAR[] record) : boolean
+ getFilenameGenerator():FilenameGenerator
+ setFilenameGenerator(FilenameGenerator filenameGen): void
+ getNodeID(): String

2

Node Attributes

This chapter describes attributes of the Oracle Communications Offline Mediation Controller nodes.

Collection Cartridges (CCs)

CC nodes should contain the following:

- EITransport
- NPLFieldProcessor
- Knowledge of (or Factory for) the appropriate DCFieldContainer object to generate for the incoming data

The EITransport class reads the data from outside the system and creates the appropriate DCFieldContainer objects. The NPLFieldProcessor would then map the data from the DCFieldContainer into a NAR based on the commands in the NPL file. The DataProvider and DataReceiver relationships are shown in Table 2-1.

Table 2-1 DataProviders and DataReceivers for Collection Cartridges

DataProvider	DataReceiver
EITransport	NPLFieldProcessor
NPLFieldProcessor	NARFileManager

Processor Cartridges

There is a base ProcessorNode class. However, most developers should not have to derive new Processor Node classes. Instead, one of the provided Processor Nodes (see Table 2-2) or a generic NPLProcessorNode (which contains an NPLFieldProcessor) should be used and the desired functionality should be performed using NPL commands.

In a ProcessorNode, the node's DCStreamHandler (i.e., NARFileManager) is both DataProvider and DataReceiver for the FieldProcessor.

Table 2-2 DataProviders and DataReceivers for Processor Cartridges

DataProvider	DataReceiver
NARFileManager	NPLFieldProcessor
NPLFieldProcessor	NARFileManager

Several generic Processor Nodes are provided as part of the Offline Mediation Controller system and are described below.

NPLProcessorNode

This is a generic Processor that can be used for adding or removing fields from a NAR or filtering records based on a particular condition or calculation.

FileEnhancerNode

This node uses a lookup file that can be used to add additional data to a NAR. For example, a file containing a listing of port numbers and their associated applications can be used to add an application description field to a NAR, based on the port number contained in one of the other fields in the NAR.

Lookup file

To use this node you will need to provide a lookup file that complies to the following format:

```
key_value_separator = 'seperator1'
pair_separator = 'seperator2'
keyPart=valuePart1
keyPart2=valuePart2
...
keyPartN=valuePartN
```

where the 1st and 2nd lines are optional. When either of the two lines is omitted, the following defaults will be used:

```
key_value_separator = "="
pair_separator = "/n"
```

key_value_separator and pair_separator values should be wrapped in single quotes, the only special characters that will be recognized as such will be '\n', '\t' and '\r'. All other characters included between the single quotes will be interpreted as independent characters.

```
pair_separator='\n###NEW PAIR###\n'
```

This would be interpreted as a new line followed by ###NEW PAIR### followed by a

new line.

```
key_value_separator='\t\f'
```

This would be interpreted as a tab followed by \f (NOT form feed).

NPL File

Following is an example of NPL that you could use with the FileEnhancerNode to do “application" enhancement.

- Input field in.1 is a port number
- Input field in.2 is the host name of the application
- The variable tablename represents a cross-reference table or file named "apps" containing mappings from port numbers to application names. This specifically is generated from the contents of your lookup file.
- Output field out.3 is the attribute which will receive the application name.
- Output field out.4 is the attribute which will receive the module name.

- The import statement imports the interface corresponding with the Method Handler implementation that will be used by the NPL for the Java hook (the lookup call).

Example 2-1 FileEnhancer NPL

```
·// NPL file to do simple app enhancement on a 1 attribute nar.
·
·import com.nt.udc.processor.FileEnhancer.FileEnhMethodHandlerIfc;
·
·String tablename = "apps";
·
·InputRec {
· String 1;        // port number
  String  2;        //host
·} in;
·
·OutputRec {
· String 3;        // application name
· String 4;        // module name
·}
·
·out.3 = Java.lookup(tablename,in.1);
·out.4 = Java.lookup(tablename,in.1+"|"+in.2);
·
·write(out);
```

NodeConfigGUI Fields

Currently there are only two fields in the node specific portion of the GUI. These are Lookup File and Lookup Table Name. Lookup file is the absolute path to the file you want to use to generate the lookup table. Lookup table name is the tag you will use from NPL to access this table. This attribute is included because in the future we will provide support for lookup from multiple tables in one FileEnhancer. In the near future there will be added support in the Node Config GUI for the concept of configuring loaders. Loaders are objects that populate a lookup file for you from some source (e.g. an LDAP directory).

FtpFileEnhancerNode

Similar to the FileEnhancerNode except this provides functionality to obtain the lookup file from a remote location via FTP.

LDAPEnhancerNode

Directory information is gathered from an LDAP directory and stored in a file, which can then be used to add or remove fields in the NAR.

Multithreaded Programmable Aggregation Processor Node

You can customize the Aggregation Processor node by its NPL file, which contains functions that perform the common aggregation tasks, including record storage and retrieval and attribute aggregation.

Note:

All aggregator NPL rule files must start with the following statement:

```
import com.metasolv.nm.processor.MXAggregator.MXJavahookHandler
```

Configuring the Aggregator NPL Rule File

The following configuration variables are available in the Aggregator NPL rule file:

ModulusAttribute

This multithreaded aggregator distributes input records to its processor threads by using a modulus routing algorithm. The algorithm is almost identical to the one used for routing NARs between cartridges.

The only difference is that the multithreaded aggregator allows two additional types of NAR fields to be used as selector values for modulus routing. Normal modulus routing allows only IntFields and LongFields to be used as selector values. The multithreaded aggregator allows BytesFields and StringFields in addition to IntFields and LongFields.

FlushOnStartup

This variable indicates if any valid records in the stored file hash table(s) should be flushed to the output on startup. If records were stored in this table when the node was stopped, they will be restored on the next startup. This method provides a simple means of ensuring any old items in the table are immediately flushed. The valid values for this item are “true" and “false". The default is false.

Hash Table Definitions

The Aggregation Processor supports multiple tables. Each table is mapped to a file in the Aggregation Processor's scratch directory. The configuration items and Java hooks that operate on a specific hash table use an index. The indexes start at 1.

NumberOfHashTables

This variable tells the node how many file hash tables to create.

The following variables are repeated, one for each table. The GUI specifies the timer value for the tables. This value must be the same for all tables.

HashTableKeysX

Specifies the attribute IDs that are used as keys for a specific file hash table. The X is the index of the hash table. For example, HashTableKeys3. The node constructs the key using the attribute IDs specified in the HashTableKeysX variable. The value of this variable is a string of IDs separated by spaces. For example, "20001 20232 10036".

HashTableFlushX

Specifies the behavior of the flush timer for a table, where X is the table index. The three possible values for this field and their associated behaviors are:

1. write - the record is output to the next node in the chain
2. delete - the record is dropped from the node chain
3. off - the flush timer is not active for the table

FlushImmediate

This variable indicates whether any valid records in the stored file hash table(s) should be flushed immediately to the output if Invoice_Date_Time has passed the current time, without waiting for the volume and flush timer conditions to be met. The flushImmediate(out, tableIndex) method introduced in the MxAggregator NPL is invoked as shown below.

```
String DATE_FORMAT="yyyy-MM-dd HH:mm:ss";
currTime = currentTime();
cdrTime = str2TimeInMilliSecs(in.Invoice_Date_time, DATE_FORMAT);

if(currTime - cdrTime > 0) {
        Java.flushImmediate(out, tableIndex);
}
```

FlushInactiveHashKey

The **flushInactiveHashKey** variable clears any inactive keys in the stored file hash table(s), regardless of whether the volume and flush timer conditions have been met. The valid values for this variable are "true" and "false", with a default of "false". In the **general.cfg** file of the aggregator node, setting the **flushInactiveHashKey** as “true” and setting the **inactiveperiod** parameter activates the overridden **storeNARWithTimer(out, tableIndex)** method, which triggers a timer to periodically check for inactive keys.

Traffic Volume Configurations

The traffic volume container Java hooks require you to set the following configuration variables:

TVMChangeTime

Specifies the string name for the time change component of a traffic volume container. For example, “ChangeTime".

TVMUplinkVolume

Specifies the name for the item containing the uplink volume in the traffic volume container. For example, “DataVolumeGPRSUplink".

TVMDownlinkVolume

Specifies the name for the item containing the downlink volume in the traffic volume container. For example, “DataVolumeGPRSDownlink".

Convenience Attribute Sets

There are Java hook functions that contain sets of attributes. These sets are specified in the configuration file, and can use any unused name. The Java hooks perform the appropriate lookup for the name they receive. This is a more optimized approach than using in the full string attribute list each time a Java hook is called. In the string list approach, the string must be parsed and tokenized each time the Java hook is called but in the optimized approach, each attribute set is loaded the first time it is needed, and then an internal representation is stored for subsequent uses.

For example, the following line can appear in the configuration:

```
config {
…
MyAttributeSet "1 5 20001 32075";
}
```

And then this attribute set could be called from some Java hooks:

```
…
Java.replaceAttributeSet(in, out, "MyAttributeSet");
…
Java.removeAttributeSet(out, "MyAttributeSet");
```

Java Hooks

The following are all the Java hooks provided by the Aggregation Processor. When a parameter type of NAR is used, the node expects to receive an InputRec or OutputRec type.

```
void appendLists(NAR source, NAR dest, String attrList)
```

Searches the attributes in the attrList and for each, appends it from the source NAR to the corresponding attribute in the destination NAR.

void appendListsWithoutRepeat(NAR source, NAR dest, String attrList)

Performs the same function as appendLists with one exception: when appending, a value will not be repeated in succession. For example, appending the following:

Source: “1", “5", “7", “7", “2"

Dest: “9", “5", “5", “6", “1"

Produces the following: “9", “5", “5", “6", “1", “5", “7", “2"

Integer compareBytes(NAR A, NAR B, Integer attrId)

Performs a byte comparison of the attributes specified in the two NARs.

Returns: “1" if the two are the same or “0" otherwise.

void concatenateStrings(NAR source, NAR dest, String attrList)

Searches the attributes specified in the string and concatenates the corresponding source attribute onto the destination.

```
void concatenateStrings(NAR source, NAR dest, String separator, String attrList)
```

Searches the attributes specified in the string and concatenates the corresponding source attribute onto the destination similar to the above function. This variation also inserts the separator between each destination and source when concatenating.

```
Integer distributeTrafficVolumeSetPerDay(NAR in, String trafficVolumesID, NAR dest,
String uplinkSetName, String downlinkSetName)
```

Parses the traffic volume containers, indicated by the trafficVolumesID, and distributes the uplink and downlink volumes into other attributes according to the time. The destination attributes used for the volume distribution are specified by the uplinkSetName and downlinkSetName. Each of these must always contain 24 attributes, each corresponding to one hour of the day.

For example, if the NAR contains two traffic volumes, one for 8am and another for 2pm, the corresponding uplink and downlink values will be put into the 9th and 15th attribute Ids from the corresponding sets.

This function only distributes the traffic volumes for one day. For example, if there is a traffic volume container that is for the next day, that traffic volume and any following it are not processed. If there are any unprocessed traffic volume containers, the processed ones will be removed from the incoming NAR so only the unprocessed ones will remain. The incoming NAR is not modified if all the volume containers are processed.

The traffic volume configuration information must be set for this function to operate. The function returns “1" if all the volume containers were processed and “0" if some were not processed and have been indicated in the incoming NAR.

```
Integer distributeTrafficVolumesPerDay(NAR in, String trafficVolumesID, NAR dest, String
uplinkVolumeAttrs, String downlinkVolumeAttrs)
```

This method is the same as the one above, with the exception that the uplink and downlink attribute IDs are listed as a string parameter rather than using a named set.

Bytes generateOpeningTimeFromTrafficVolume(List trafficVolumes)

Returns a byte time representation for the beginning of the day for the first traffic volume container in the list. More specifically, it is the bytes representation of the same timestamp, but with the hour, minute and seconds all set to “0".

The traffic volume configuration information must be set for this function to operate.

```
Integer getBytesValueFromListMapIp(List inData, Integer index, nar dest, String attrId)
```

This specialized function sets the bytes in the attrId of the destination to the IP of the first valid IP type in a map, where the map comes from the specified item, as per index, in the list. For example, in the Wireless market segment, the SGSN IP Address List field is a list of maps, with an IP specified in these maps. Using this function on this field sets the bytes in the destination to be the bytes representation of the IP address of one of the SGSNs in the list, using index to determine which SGSN.

The index is specified starting at 0. This will return 1 if an IP was found and subsequently set in the destination, otherwise 0 will be returned.

```
Integer getDayOfYear(Bytes date)
```

Determines the current day of the year. For example, if the date is March 20, 2003, the method returns “20".

```
Integer getNAR(NAR out)
```

Searches for a NAR in the default hash table, 0. Upon finding the NAR, the method returns it in the out parameter and removes it from the table. The method also removes any timers. To search, the method uses the key created by the last call to setKey for this hash table.

The method returns “1" if it finds a NAR and “0" otherwise.

```
Integer getNAR(NAR out, Integer index)
```

Performs the same function as getNAR above, but also supplies an index to indicate the hash table to use.

```
Integer getPreviousDayOfYear(Bytes date)
```

Returns the day of the year for the day previous to the current date. This is the same as “getDayOfYear(date)-1" except in the case where the current date is the first day of the year. In this case, the method returns the last day of the previous year.

```
void keepMaxAttributes(NAR source, NAR dest, String attrList)
```

Searches the attributes specified in attrList, and puts the greater value of the source or destination in the dest field. This method supports the following types:

- byte
- short

- int
- long

`void keepMinAttributes(NAR source, NAR dest, String attrList)`

Searches the attributes specified in attrList, and puts the lesser value of the source or destination in the dest field. This function supports the following types:

- byte
- short
- int
- long

`void removeAttributes(NAR in, String attrList)`

Removes all attributes specified in attrList from the incoming NAR.

`void removeAttributeSet(NAR in, String setName)`

This method performs the same function as above, except the attribute list is contained in the set specified by setName.

`Integer removeNAR()`

Deletes the record matching the key set for hash table 0 from the table.

The method returns “1" if it finds and removes a matching record or “0" if it does not find a matching record.

`Integer removeNAR(Integer index)`

Deletes the record matching the key set for the hash table specified by index.

The method returns “1" if it finds and removes a matching record or “0" if it does not find a matching record.

`void replaceAttributes(NAR source, NAR dest, String attrList)`

Takes each attribute specified in attrList from NAR source and puts it in NAR destination. The method overrides any values previously set in dest.

`void replaceAttributeSet(NAR source, NAR dest, String setName)`

This method is the same as above, but the attribute list is in the set specified by setName.

`void setKey(NAR in)`

Creates a key for the default hash table, 0, from the in NAR. The attributes used to create the key are specified in the configuration.

`void setKey(NAR in, Integer index)`

This method is the same as the function above, except the key is created for the hash table specified by index, where index starts from 0.

`void storeNAR(NAR in)`

Stores the incoming NAR in the default hash table, 0, with the last key specified for this hash table. The method does not start a timer for this record even if the configuration indicates timers are enabled for this table.

```
void storeNAR(NAR in, Integer index)
```

This method has the same function as above, except the hash table is specified by index.

```
void storeNARWithTimer(NAR in)
```

This method is the same as the storeNAR(in) function, except it starts a timer for this record if the option is enabled in the configuration for the default table, 0. If the record already exists in the table and the timer is already set, the method overrides both items.

```
void storeNARWithTimer(NAR in, Integer index)
```

This method is the same as the above function, except it is for the hash table specified by index.

```
void sumAttributes(NAR source, NAR dest, String attrList)
```

This method sums the source and dest attributes specified by attrList. The results are stored in dest.

```
Integer sumAttributesNoOverflow(NAR source, NAR dest, String attrList)
```

This method is the same as the above function, except it does not sum the source and dest attributes if it will result in an overflow condition.

The method returns “1" if it performs the summation and “0" if it detects an overflow condition and does not perform the summation.

```
void sumValue(Long sourceValue, NAR dest, String attrList)
```

Adds the source value to each of the attributes in dest specified by attrList.

```
Integer sumValueNoOverflow(Long source, NAR dest, String attrList)
```

This method is the same as the above function, except it does not perform a summation if it detects the attributes would overflow.

The method returns “1" if it performs the summation and “0" if it detects an overflow condition and does not perform the summation.

```
storeNARWithTimer(out, tableIndex)
```

This method triggers a timer to periodically check for inactive keys in the table and flush aggregated NARs without waiting for volume or flush timer conditions to be met.The timer activation depends on the following segmentation criteria:'

- • **Segmentation based on volume:** Only the inactivity timer is activated.
- • **Segmentation based on time or both time and volume:** Two timers are activated. One timer operates based on the flush interval (existing behavior). The other timer operates based on the inactivity period. The aggregated record is flushed when either timer's condition is satisfied, whichever occurs first.

Usage

This section will show some examples of how you can use the Java hooks.

Simple Session Aggregation Example

The following examples demonstrate a simple way to aggregate sessions for various record types. The examples show you how to aggregate partial records from the same device but do not demonstrate how to combine record types. The examples use Wireless attributes but do not include the record declarations for readability.

The following are the major attributes:

20001 - The session identifier. Unique for a specific GGSN and shared by the SGSNs.

20200 - The record type.

20232 - The GGSN IP address

20233 - The SGSN IP address, for S-CDRs only.

Due to the different keys used to uniquely identify sessions on the GGSN and SGSN, separate tables are used in the example below.

Here is the sample NPL with comments:

```
// Declare the AP java hooks
import com.metasolv.nm.processor.MXAggregator.MXJavahookHandler;

// Configure the file hash tables we're going to need

Config {
// Distribute records to threads based on session ID.
ModulusAttribute "20001";

    // Don't arbitrarily flush all existing records on startup.
    FlushOnStartup "false";

    // Indicate that there will be configuration information
    // for two different tables.
    HashTables "2";

    // Set the attributes used for the key for table 0
    HashTableKeys0 "20001 20200 20232";
    // Indicate that we want flushed records to be written
    // to the output for this table.
    HashTableFlush0 "write";

    // Add the SGSN IP to the key for the second table
    HashTableKeys1 "20001 20200 20232 20233";
    // We also want flushed records for this table to be
    // written to the output.
    HashTableFlush1 "write";

    // Declare a set of attribute Ids to use later on.
ReplaceSet "20001 20002 20005 20121 20200 20201 20202 20203 20204 20205 20206 20207
20210 20211 20212 20214 20216 20217 20218 20219 20220 20221 20222 20223 20224 20225
20226 20227 20228 20229 20232 20233 20234 20235 20300 20301 20302 20303 20304 20305
20306 20310";
}

Integer returnCode = 0;
Integer tableIndex = 0;

// The input and output record declarations would be here.
// The SMS CDR types are not aggregated, so just write them out.
if ((in.20200 == 21) || (in.20200 == 22)) {
    out = in;
    write(in);
}
else {
    // Get the index to the appropriate table. Use table 0 for G-CDRs,
    // and table 1 for M and S-CDRs, as they require a key with the
    // SGSN IP address also.
    if (in.20200 == 19) {
        tableIndex = 0;
    }
    else {
        tableIndex = 1;
    }

    // Now set the key for the table we will be using. This will be
    // in place for all further operations on this table.
    Java.setKey(in, tableIndex);

    // Try to find an already existing record in the table
    // which matches our key. The returnCode variable will
    // indicate if the lookup was successful.
    returnCode = Java.getNAR(out, tableIndex);

    if (returnCode == 1) {
        // Found one. We now need to do the aggregation between the
        // found record (out) and the input record.

        // First, try to sum the duration if the variable will not
        // overflow.
        returnCode = Java.sumAttributesNoOverflow(in, out, "20004");
        if (returnCode == 0) {
        // Overflow condition. Just flush out the old record and
            // start over with the new one.
            write(clone(out));
            out = in;
        }
        else {
            // The sum worked, so continue with all the rest.
            // These two items are lists which are aggregated by
            // appending the incoming onto the end of the stored
            // attribute.
            Java.appendLists(in, out, "20209 20213");

            // This is a convenient way to perform bulk assignment
            // of attributes from in to out. This could be done as
// separate assignments (e.g. out.20001 = in.20001;), but
            // it would be less efficient than this call and would
            // also be less readable.
            // This function also makes use of the named set
            // "ReplaceSet", which identifies a set of attributes
            // declared in the configuration block at the top.
            Java.replaceAttributeSet(in, out, "ReplaceSet");
        }
    }
    else {
        // There is no matching record already in the table, so this
        // is the first. Rather than doing separate assignments for
        // all the needed attributes, this is just doing a bulk copy
        // of all of them.
        out = in;
    }

    // Now check to see if the record should be written to the output
    // or stored back in the table. We'll only ever get here for the
    // G-CDR, S-CDR and M-CDRs, so just check the cause for record
    // closing value to see if the session is closed.
    if ((out.20202 == 0) || (out.20202 == 4)) {
        write(out);
    }
    else {
        Java.storeNARWithTimer(in, tableIndex);
    }
}
```

Distribution Cartridges (DCs)

DC nodes should contain the following:

- NPLFieldProcessor
- OITransport
- Knowledge of (or Factory for) the appropriate DCFieldContainer object to generate.

The NPLFieldProcessor maps the data from the NAR into the appropriate DCFieldContainer object based on the commands in the NPL file. The OITransport class then takes the data and transmits it via the appropriate medium outside the system. The DataProvider and DataReceiver relationships are shown in Table 2-3.

Table 2-3 DataProviders and DataReceivers for Distribution Cartridges

DataProvider	DataReceiver
NARFileManager	NPLFieldProcessor
NPLFieldProcessor	OITransport

Several DC nodes are provided as part of the Offline Mediation Controller system and are outlined below.

ASCII DC Node

Flat file based DC node which will produce output in ASCII format. Also has the capability to put the completed files on a remote machine via FTP.

IPDR DC Node

Produces file-based output in IPDR format.

XML DC Node

Produces file-based output in XML.

JDBC DC Node

The JDBC DC node is a generic node that inserts data into a relational database. You can find this DC node in the Offline Mediation Controller Cartridge Kit market segment, which is part of the Database Storage and Reporting solution. The DC node outputs DMS-MSC data (GCDR and GHOT records) into an Oracle database. Still, it can also be configured to work with other types of data and databases, providing the proper NPL rules files are written. The JDBC DC node can connect to any type of relational database without modification of the existing Java code. This DC node uses a Java JDBC interface to insert data into a database. Therefore, any database that supports JDBC works with this DC node.

The basic node chain that uses the JDBC DC node to insert data into a database contains a DMS-MSC CC node and a JDBC DC node.

This DC node obtains the necessary database information from the NPL file it is configured to use. The NPL file must have a configuration clause containing a configuration key of DBTables and an associated configuration value representing a list of one or more database table names. This comma-separated list of names corresponds to the database tables where the incoming NAR attributes are inserted. For each table in the list, there must also be a corresponding Expose clause, which will associate attributes of the output records to the appropriate column within the specified database table. The data type of the output attribute must be compatible with the data type for the database column. The DC node issues a processing exception if the data types are incompatible. Refer to the sample NPL code listed at the end of this section for more details.

The NPL file must have a configuration clause containing two configuration keys: “JDBCDriver" and “JDBCUrl". The two configuration keys must have a configuration value associated with them.

The configuration key, “JDBCDriver", is the class name of the JDBC driver provided by the specific database. In the case of the Oracle database, the class name is “oracle.jdbc.driver.OracleDriver".

The configuration key, “JDBCUrl", identifies a data source so the appropriate driver recognizes and establishes a connection with it. Different JDBC drivers require different JDBC URLs. You must provide the appropriate JDBC URL information as the configuration value for the “JDBCUrl" clause in the NPL rules file. The values of the database host, the port that the database server is listening to, and the database SID appear in the node configuration window in the Offline Mediation Controller GUI. If the JDBC URL requires any of these three fields, you must replace these fields with “%DBHOST%", “%DBPORT%", and “%DBSID%" in the “JDBCUrl" configuration value.

For the Oracle database, the “JDBCUrl" configuration value is: “jdbc:oracle:thin:@%DBHOST%:%DBPORT%:%DBSID%".

For example, if the database host is “MyHost", the port number is “1521", and the SID is “ORCL", the JDBC DC node uses “jdbc:oracle:thin:@MyHost:1521:ORCL" as the JDBC URL to connect to the database.

The configuration values “DBCatalog" and “DBSchemaPattern" are the additional criteria used to validate the database table. You only need to specify these two values in the NPL rules file when there are two tables with the same table name (in different schema patterns). In Offline Mediation Controller, the “DBSchemaPattern" is “NMUSER1" and the “DBCatalog" is “SYS.ALL_TABLES".

The following is a sample NPL rules file for the JDBC DC node working with an Oracle database. There are two tables: “tableName1" and “tableName2". Each table has five columns: “column1", “column2", “column3", “column4", and “column5". Columns 1 through 4 are all INTEGER types. Column 5 is a VARCHAR type.

```
Config {
    DBTables   "tableName1,tableName2";
    JDBCDriver   "oracle.jdbc.driver.OracleDriver";
   JDBCUrl    "jdbc:oracle:thin:@%DBHOST%:%DBPORT%:%DBSID%";
    DBCatalog "SYS.ALL_TABLES";
               DBSchemaPattern "NMUSER1";
}

InputRec {
// input record fields
             Integer 1000;
```

```
  Integer 1001;
Integer 1002;
Integer 1003;
String 1004;
} in;

OutputRec {
// output record fields
Integer attribute1;
Integer attribute2;
Integer attribute3;
Integer attribute4;
String attribute5;
} out;
Expose for tableName1 {
      out.attribute1      "column1";
      out.attribute2      "column2";
      out.attribute3      "column3";
      out.attribute4      "column4";
      out.attribute5      "column5";
}

Expose for tableName2 {
      out.attribute1      "column1";
      out.attribute2      "column2";
      out.attribute3      "column3";
      out.attribute4      "column4";
      out.attribute5      "column5";
}
out.**** = in.****;
…

write(out);
```

Table 2-4 displays the required configurable parameters for the JDBC DC node.

Table 2-4   Configurable Parameters for JDBC DC Node

NPL	Description
DBTables	List of table names separated by quotes: “ "
JDBCDriver	Class name of JDBC driver
JDBCUrl	String of JDBC URL
DBCatalog	Database catalog (optional)
DBSchemaPattern	Database schema pattern (optional)

Table 2-5 displays the required configurable parameters for the JDBC DC node.

Table 2-5   Node Config GUI Fields

Node Config GUI	Description
UserId	Database user ID
Passwd	Database user password
Batch Size	Number of rows in one batch insertion
DB Host	Host name or IP address of the database host

Table 2-5 (Cont.) Node Config GUI Fields

Node Config GUI	Description
DB Port	Port that the database is listening to
DB SID	Database SID

The JDBC DC node uses batch insertion to insert a certain number of records at one time into the database. The batch size is configurable through the node configuration window in the Offline Mediation Controller GUI. However, a driver is not required to implement the batch execution. For those drivers that do not support batch execution, the JDBC DC node singularly inserts the records into the database. The DC node can call the DatabaseMetaData method and supportsBatchUpdates value to find out whether the driver supports batch updates.

3
Customizing the Administration Client GUI

This chapter provides an overview of customizing the Oracle Communications Offline Mediation Controller Administration Client GUI.

DCNodeConfigGUI and Related Classes

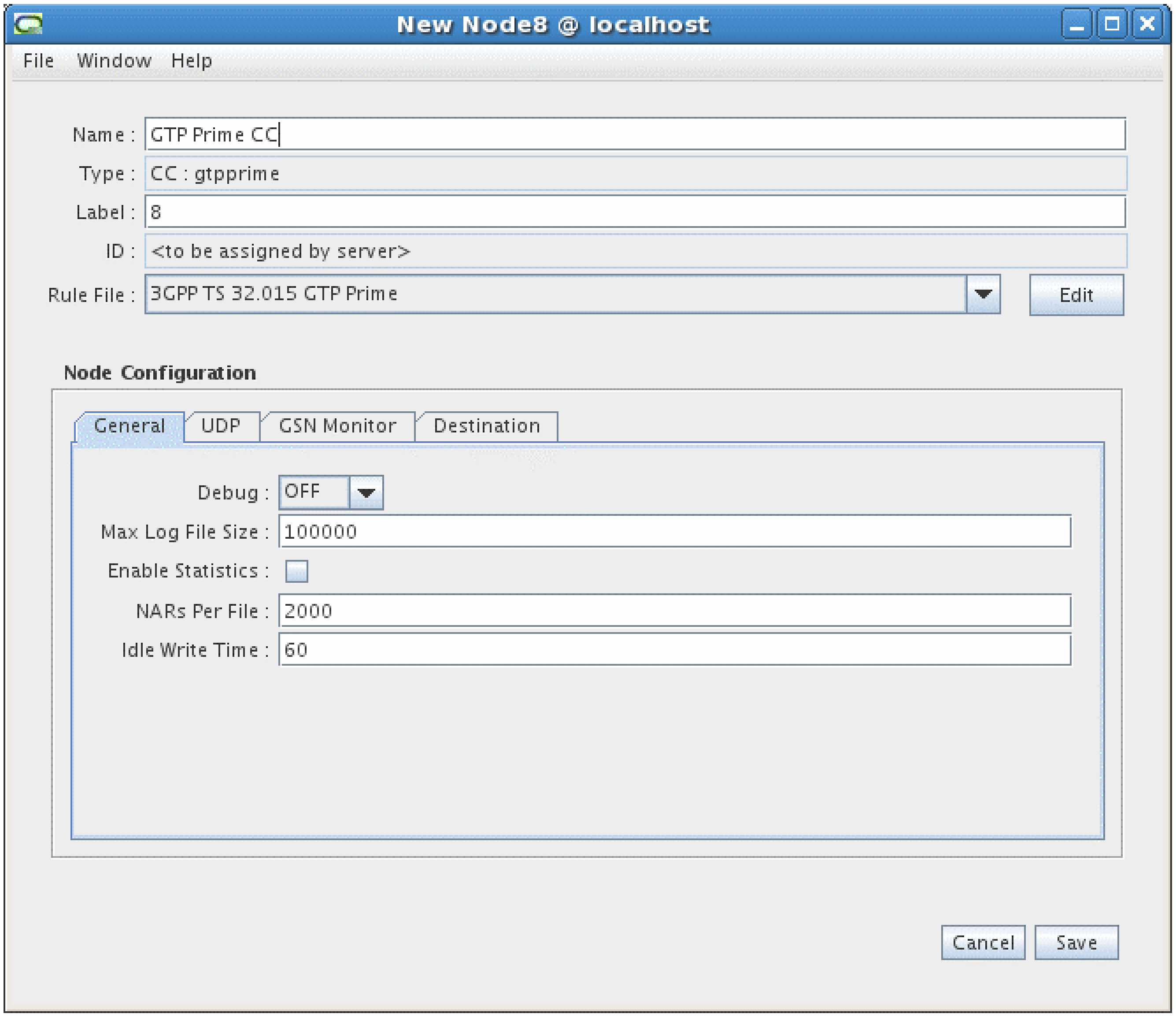
The Administration Client GUI displays configuration panels for all nodes available in the system. Node developers must provide a class that shows configuration information specific to their node.

The base DCNodeConfigGUI classes provide the basic configuration tab panel and the **General** tab components applicable for a particular node type. For CC and Processor nodes, a **Destination** tab allows the user to select the nodes that will receive data from the node.

The images below show the contents of the **General** tab for a Wireless CC, EP, and DC node.

Offline Mediation Controller provides the basic panel layout, which includes the node identification fields at the top and the configuration tabs in the lower half of the screen.

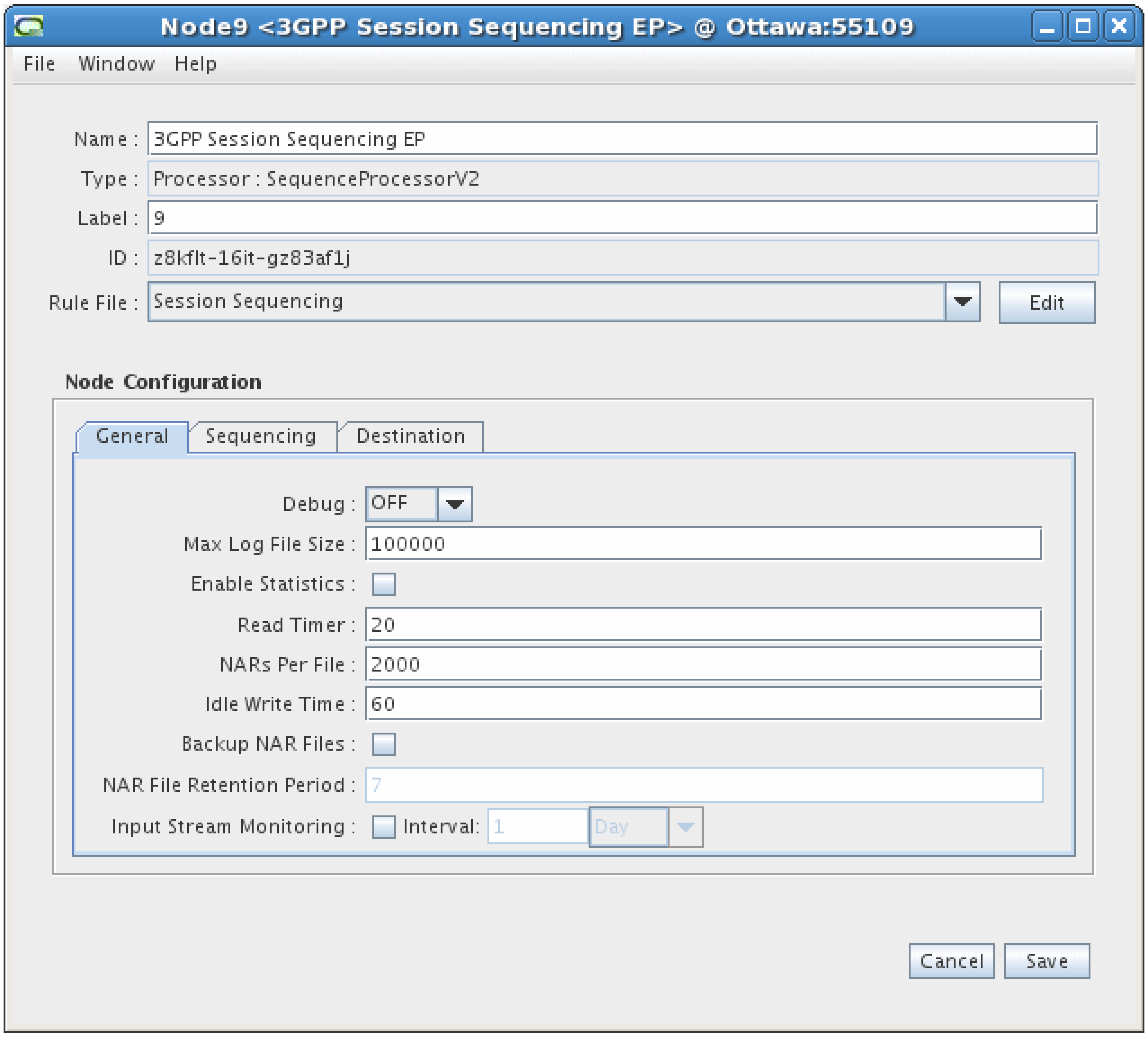
Figure 3-1 CC Node Configuration GUI.



Name : GTP Prime CC
Type : CC : gtpprime
Label : 8
ID : <to be assigned by server>
Rule File : 3GPP TS 32.015 GTP Prime

Debug : OFF
Max Log File Size : 100000
Enable Statistics :
NARs Per File : 2000
Idle Write Time : 60

Figure 3-2 EP Node Configuration GUI

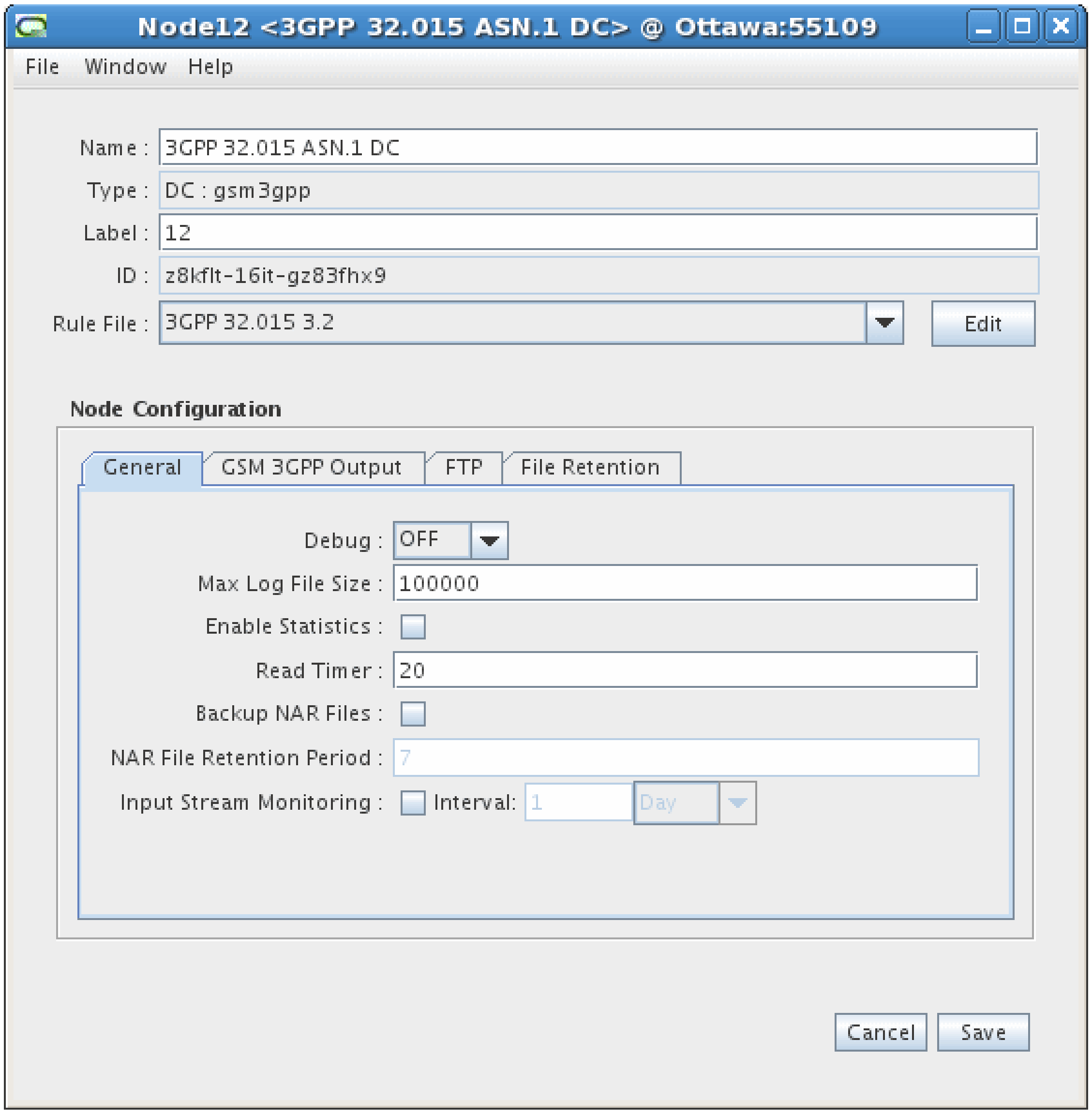


Name :	3GPP Session Sequencing EP
Type :	Processor : SequenceProcessorV2
Label :	9
ID :	z8kflt-16it-gz83af1j
Rule File :	Session Sequencing

Node Configuration

Debug :	OFF
Max Log File Size :	100000
Enable Statistics :	☐
Read Timer :	20
NARs Per File :	2000
Idle Write Time :	60
Backup NAR Files :	☐
NAR File Retention Period :	7
Input Stream Monitoring :	☐ Interval: 1 Day

Figure 3-3 DC Node Configuration GUI


Node12 <3GPP 32.015 ASN.1 DC> @ Ottawa:55109

File Window Help

Name : 3GPP 32.015 ASN.1 DC
Type : DC : gsm3gpp
Label : 12
ID : z8kflt-16it-gz83fhx9
Rule File : 3GPP 32.015 3.2

Node Configuration

General | GSM 3GPP Output | FTP | File Retention

Debug : OFF
Max Log File Size : 100000
Enable Statistics : ☐
Read Timer : 20
Backup NAR Files : ☐
NAR File Retention Period : 7
Input Stream Monitoring : ☐ Interval: 1 Day

New configuration GUI classes should be derived from the appropriate DCNodeConfigGUI class, and at least the following three methods should be overridden:

- **setDefaults()**: Provides appropriate default values for the configuration items being introduced. The defaultValue() method is used to set these individual values.
- **extendConfigGUI()**: Defines any new tab panels.
- **getNodeSpecificConfigData()**: Gets the values from the text fields (or other components) and sets them in the DCNodeConfigData (nodeData) object, which will contain all of the valid configuration information for the node.

The allFieldsValid() method is called when the user selects the **Save** button. The default implementation always returns true. If any input verification is desired, perform the checking here.

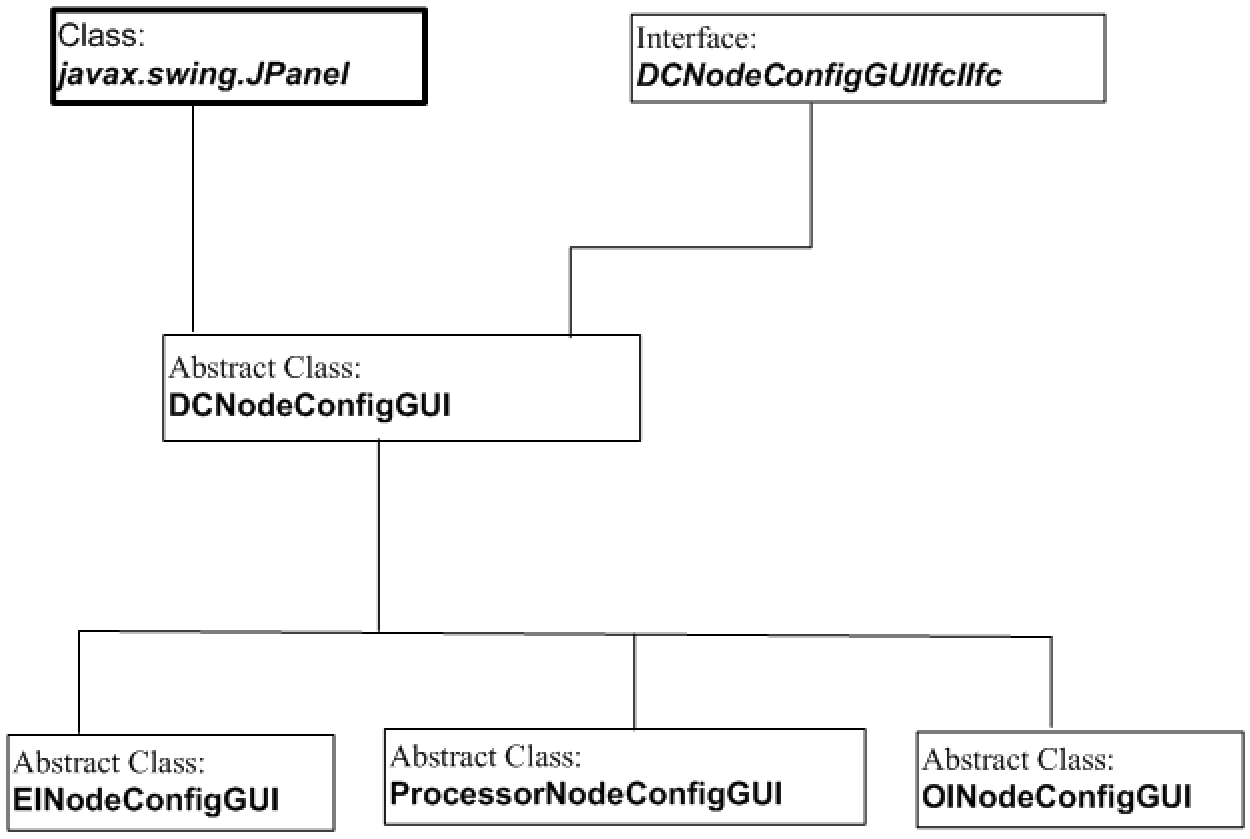
The DCNodeConfigGUI base class provides some methods for creating basic components and panels:

- **makeTextField()** and **makeNumericTextField()**: Create JTextField objects with the supplied parameters.
- **makeNumericTextField()**: Provides simple input verification to ensure the user has entered a numeric value. A minimum and maximum value can also be specified. If a minimum and maximum value are specified, the values will be enforced. If the user enters a value below the minimum, the minimum value will be displayed. Conversely, if the user enters a value above the maximum, the maximum value will be displayed.
- **makePanel()**: Accepts a Vector of labels and a Vector of components, and creates a JPanel that contains those items in the order in which they appear in the Vector. The addPanel() method adds the panel to the tabbed pane in the GUI.

DCNodeConfigGUI Class Hierarchy

Figure 3-4 shows the DCNodeConfigGUI class hierarchy.

Figure 3-4 DCNodeConfigGUI Class Hierarchy



DCNodeConfigData

Use the DCNodeConfigData object to store configuration information for the node. Currently, only String data is supported for configuration information. Node developers should not have to derive any classes from DCNodeConfigData.

Node Template Definitions and Groups

Rules files are the NPL files that will be used by the node (more specifically by the NPLFieldProcessor). These files are stored in the Offline Mediation Controller installation area in the **rules** subdirectory in a directory structure based on the node's major and minor type.

The system uses node template files to determine what nodes are available and how to display those nodes in the GUI tree view and Node Creation Wizard. These files are stored in the **config/template** directory.

The system uses two files to describe what a node is. The first file, the **nodeTemplateDef.xml** file, defines the node's name and other properties, such as the NPL files and the node class. The second file, **nodeGroupDef.xml**, establishes the hierarchy of the nodes in the GUI. You cannot modify the system files, but you can extend them by creating a customized version in the *OMC_home***/customization** directory.

The nodeTemplateDef.xml File

The **nodeTemplateDef.xml** file defines node template elements. Each node template defines a single node in the system. For example, the following defines an Aggregation Processor node named **Sample Node 001** with one NPL file.

```
<nodetemplate id="Access#AP#Testing001">
    <name>Sample Node 001</name>
    <property name="Solution">access</property>
    <rule id="aggr_csg_radius_correlation.npl">
        <name>Sample node correlation</name>
        <nplfile>sample_node_correlation.npl</nplfile>
    </rule>
    <nodeclass>com.nt.udc.aggregator.AggregatorNode</nodeclass>
</nodetemplate>
```

The system *OMC_home***/config/nodeTemplateDefs/nodeTemplateDef.xml** file already defines a large number of nodes. You must not change the existing definitions. You can, however, extend the definitions by creating new NPL files and referring to them in the customized **nodeTemplateDef.xml** file. For example, if **Sample Node 001** was defined in the system node templates and you extended it with a second NPL file named **sample_node_test.npl**, the **nodeTemplateDef.xml** file in the customization directory would appear as follows:

```
<?xml version="1.0"?>
<nodetemplateset version="1.0">
    <nodetemplate id="Access#AP#Testing001">
        <rule id="sample_node_test.npl">
            <name>Sample Node Test</name>
            <nplfile>sample_node_test.npl</nplfile>
        </rule>
    </nodetemplate>
</nodetemplateset>
```

Each rule element defines a single NPL file for the node. The rule ID attribute should be unique and is usually just the file name. The NPL file attribute does not include the path.

Ensure you place the customized **nodeTemplateDef.xml** file in the *OMC_home***/ customization/nodeTemplateDefs** directory. The directory can contain any number of node templates that either add NPL files to existing system templates or define new nodes.

The Offline Mediation Controller system searches for NPL files in the following manner:

1. Construct the abstract location of the NPL file as follows:

   MarketSegment/*BasicNodeType*/*NodeSpec*/*Filename*

   where:

   - *MarketSegment* is defined by the first token in the **nodeTemplate id** attribute (a # separates tokens). In the example given above, the market segment is "Access". If the node template ID is "Wireless#EP#FileEnhancer", the market segment is "Wireless".
   - *BasicNodeType* and *NodeSpec* can be determined by editing a node of that type in the GUI. The **type** field will display both the *BasicNodeType* and *NodeSpec* separated by a colon.

     In the previous example, the type field would display "Processor : Aggregator". In this case, the location of the node's system-defined NPL file is *OMC_home***/rules/Access/Processor/Aggregator/sample_node_test.npl**. If it is customized in **sample_node_customized.npl**, the location is *OMC_home***/customization/Access/Processor/Aggregator/sample_node_test.npl**.
2. Search in *OMC_home***/customization**.
3. If you do not find the file, search in *OMC_home***/rules**.

Node templates have two other properties defined in the XML: a **Solution** property and a **node class**.

- The **Solution** property is the market segment used in licensing. Each node must have a valid **Solution**. If you create a new node template, set the **Solution** to **CartridgeKit**.
- The **node class** is the actual Java type that the system instantiates when creating a node of this type.

The GUI shows only the rule files listed in the node template definition. Therefore, to add a new NPL file, copy it into the appropriate directory and add it to the node template definition.

The nodeGroupDef.xml File

The **nodeGroupDef.xml** file defines the hierarchy of nodes for the GUI to use in its tree view and the Node Creation Wizard. This file has two elements in it, the **nodetemplategroup** and the **nodetemplate**. The **nodetemplategroup** elements define the groupings and the **nodetemplate** elements define the leaf nodes. Each **nodetemplategroup** contains a name and either a list of **nodetemplategroups** or a list of **nodetemplates**. Each **nodetemplate** is an empty element and has an ID that references an existing **nodetemplate** in the **nodeTemplateDef.xml** file. The **nodeGroupDef.xml** file can be extended by adding various **nodetemplategroups** and **nodetemplates** to a **nodeGroupDef.xml** file in the **customization** directory.

For example, if the following was the **system nodeGroupDef.xml** file:

```
<?xml version="1.0" standalone="yes"?>
<nodetemplategroup>
    <name>root</name>
    <nodetemplategroup>
        <name>Wireless</name>
        <nodetemplategroup>
            <name>Collection Cartridge (CC)</name>
            <nodetemplate id="Access#AP#SessionCorrelation"/>
            <nodetemplate id="Access#AP#CsgRadiusCorrelation"/>
        </nodetemplategroup>
    </nodetemplategroup>
</nodetemplategroup>
```

This could be extended by creating the following **nodeGroupDef.xml** file in the **customization** directory.

```
<?xml version="1.0" standalone="yes"?>

<nodetemplategroup>
    <name>root</name>
    <nodetemplategroup>
        <name>Wireless</name>
        <nodetemplategroup>
            <name>Collection Cartridge (CC)</name>
            <nodetemplate id="Access#AP#Test001"/>
        </nodetemplategroup>
    </nodetemplategroup>
</nodetemplategroup>
```

Creating a node and rule file

To create a customized node, you must access the node type you want to use as a template and create a new rule file. To access the node type templates, go to the Cartridge Kit market segment in the Offline Mediation Controller Administration GUI.

Use the following procedure to create a customized node with a new NPL rule file.

Select the node type template to build on

1. In the Administration GUI, go to the **Nodes on Mediation Host** section and click the **New** button.
2. Select the **Cartridge Kit** market segment and click **Next**.
3. Select the **Node Type** and click **Next**.
4. Select a node and click **Finish**.

   The node configuration window appears.

Create and save the new NPL rule file

1. In the node configuration window, ensure “New" is selected in the **Rule File** drop-down list and then click the **Edit** button.
2. Use the template that appears in the NPL Editor window to create your new NPL rule file.
3. Select **File** and **Save** to open the Save As dialog box. Type in the name of the Display Name. Type in the Rule File Name, and ensure the file name ends with “**.npl**".
4. Click **Save**.
5. Exit the NPL Editor by selecting **File** and then **Exit**.
6. Ensure the new rule file is in the list.
7. Configure the information required in the various window tabs.
8. Click **Save**.

SNMP trap generation

There is an SNMPTrapGenerator associated with all nodes. By default, it generates SNMP traps for all of the nodes' logged error conditions (i.e., red alarms). The SNMP Trap Hosts are configured via the SNMP Trap Hosts panel in the Administration Client GUI.

4

Cartridge Creation Example

This chapter describes the creation of Cartridges with the Oracle Communications Offline Mediation Controller CDK.

Creating Airline Flight Node Chain

Follow these guidelines:

1. All nodes should be derived from EINode, ProcessorNode, or OINode.
   - Processor nodes should utilize the NPLProcessorNode or be extensions of it.
   - We have **EINodeTmpl.java** as a template for CC nodes.
   - We have **OINodeTmpl.java** as a template for DC nodes.
2. All "inter-node" communication should be handled by the DCStreamHandler.

   The DCStreamHandler at this time is the NARFileManager, which is responsible for writing NAR files to disk.
3. All nodes should have NPL rules file, which maps ALL fields available in the case of a CC node, and only the fields necessary for enhancement in the case of a Processor node. Obviously, a DC node will only include the fields that meet that specific output requirement.
4. All nodes should have the necessary NodeConfigGUI class, rules/NPL file, and should be defined in the appropriate properties file.
   - *OMC_home***/customization/nodeTemplateDef.xml**
   - *OMC_home***/customization/nodeGroupDef.xml**

   where *OMC_home* is the directory in which you installed Offline Mediation Controller
5. All nodes should have implementations of DataProviders and DataReceivers with getData() and processData() methods for which the issues of synchronization have been taken into consideration.
6. All nodes should have a main() method to be used for testing purpose only.

   Allows the node to run from the command line before the NodeConfigGUI component is complete.

Existing Node Types

The following list displays the nodes existing in Offline Mediation Controller CDK when you install the product. You can use these existing nodes as a base and customize the NPL file to achieve the desired functionality.

Collection Cartridge (CC) Nodes:

- ASCII CC node
- IPDR File CC node
- Network Accounting Record CC node

Enhancement Processor (EP) Nodes

- Record Processing EP node
- Record Enhancement (Local File) EP node
- Service Resource EP node
- LDAP Enhancer EP node
- Record Enhancement (LDAP) EP node

Aggregation Processors (AP) Nodes

- Aggregation Processor
- Programmable Aggregation Processor

Distribution Cartridges (DC) Nodes

- ASCII DC node
- IPDR DC node
- XML DC node
- Network Accounting Record DC node
- JDBC DC node

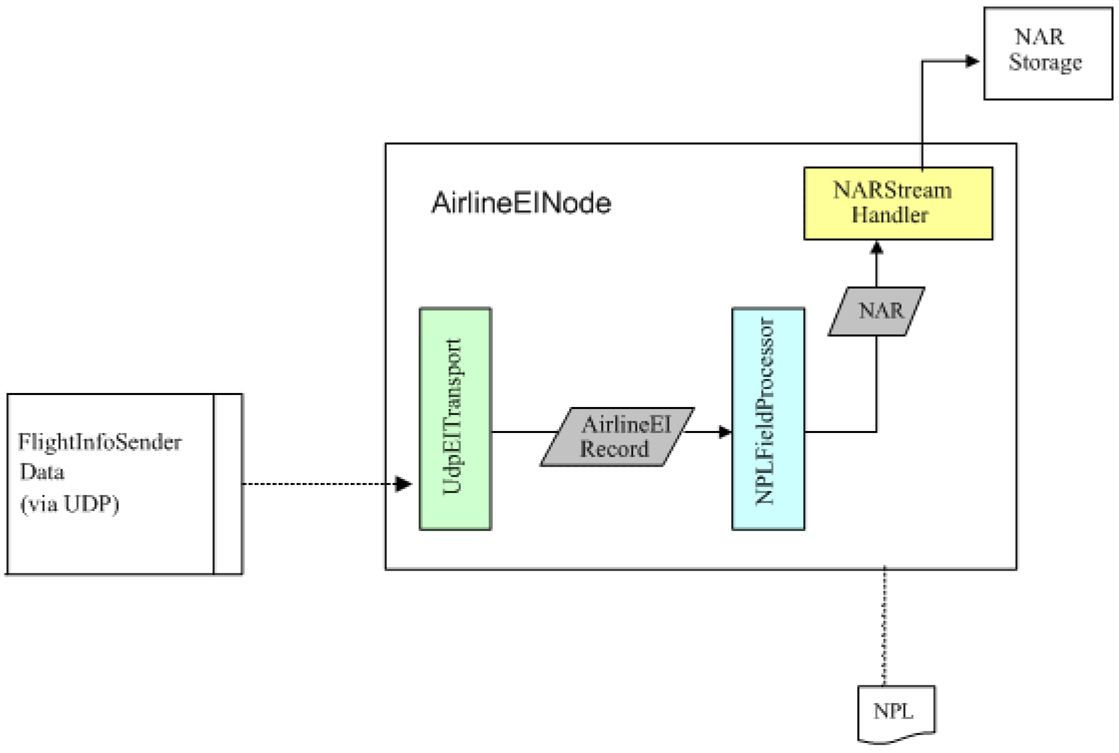
1. Identify your data source or destination (CC node vs. DC node)
   - Know the format
   - Know how it is transferred
2. Identify your transport
   - Does one already exist?
   - Do you need to write your own?
   - Will your transport know about your DCFieldContainer? -OR-
   - Will you need to write a factory?
3. (Optional) Write a factory, which is the mass production of record objects
   - A DCRecordFactory is generally used to take raw data and produce EIRecord objects.
   - A factory would typically be used to take in an OIRecord and produce raw data for an OITransport.
   - The definition of this object is only necessary when dealing with an object (generally a transport) that needs this to generate record objects for it.
4. Write a DCFieldContainer
   - In an EIRecord. This means getting raw data into the object, and implementing the interface methods for getting the data out as individual attributes. Refer to **EIRecordTmpl.java**.
   - In an OIRecord. This means implementing methods to get the data in as fields and a mechanism to get raw data out. Refer to **OIRecordTmpl.java**.
5. Implement the Node, where you really put it all together

- Write the constructor if warm restart capability is not implemented.
  - Get all configuration information for this node.
  - Construct your transport with knowledge of your factory or record.
  - Construct an NPLFieldProcessor with knowledge of its output type (NAR or your Record object) and knowledge of your rules file (which is obtained from config).
  - Set receivers and providers appropriately.
  - Decide what is running in a thread.
  - Start the appropriate components.
- If implementing warm restart capability, construct the node without any operations, override Boolean warmRestartImplemented() to return true, and implement these two methods
  - startup(): Instantiate transport and field processor and start all threads.
  - reconfigure(): Reconfigure the node without stopping itself.
- Implement remaining abstract methods
  - getMinorType()
  - getConfigGUIClass()
- Implement an “orderly" shutdown method
  - Make sure components are shutdown in the proper order.
  - Make sure all threads are REALLY stopped.
- Implement a main() method for testing purposes
- Write your NPL file
  - In a CC node and a DC node, this should be primarily mapping to/from a NAR attribute.
- Implement your NodeConfigGUI class
- Modify/create the appropriate property files

Create AirlineEINode (CC Node)

Now, we will go through step by step as described in the previous section to create an AirLineEINode for the Airline Flight node chain.

Figure 4-1 AirLineEINode



1. Identify the data source or destination (CC node vs. DC node).

   Write a CC node for the Airline Flight node chain. This CC node will collect the Airline passenger data via UDP. We have the Airline Flight Data Simulator, FlightInfoSender, which will be running on a machine somewhere and sending the datagram packet through UDP.

   FlightInfoSender Data Format: There is one passenger record per datagram packet. A single flight record is a byte array of 7 integer attributes. These attributes are (in their respective order):

   - Passenger ID
   - Flight Number
   - Departing Airport ID
   - Arriving Airport ID
   - Departure Time in Seconds
   - Arrival Time in Seconds
   - Number of Bags of Luggage for Passenger

2. Identify the transport.

   Airline passenger data will be transferred to this CC node via UDP. In CDK, we already have a transport class for UDP, which is UdpEITransport. Therefore, we don't need to write our own transport class for this CC node. Here are the constructors of UdpEITransport:

   public UdpEITransport (EINode einode, DCRecordFactoryIfc factory, int prt)

   throws NodeStartException

   public UdpEITransport(EINode einode, DCRecordFactoryIfc factory, String host, int prt) throws NodeStartException

public UdpEITransport(EINode einode, DCRecordFactoryIfc factory, int prt, int pktSize, int qSize) throws NodeStartException

public UdpEITransport(EINode einode, DCRecordFactoryIfc factory, String host, int prt, int pktSize, int qSize) throws NodeStartException

Obviously, UdpEITransport needs a DCRecordFactory in order to know about the EIRecord of the Airline passenger data.

3. Write AirlineRecordFactory.

   AirlineRecordFactory is implementing DCRecordFactoryIfc. It will produce our AirlineEIRecord for Airline passenger data, which will be created in the next step. We have a template class for creating RecordFactory, please refer to RecordFactoryTmpl.java. This template is written particularly for producing EIRecord from UDP packet. For other type of raw data, you need to use the proper type of raw data instead of QueuedUdpPacket in getRecord() and getRecord() two methods.

4. Write DCFieldContainer: AirlineEIRecord.

   Now we will create an EIRecord class, which will be responsible for collecting the data produced by the FlightInfoSender class and provide “get" methods for extracting the individual fields. Please refer to EIRecordTmpl.java.

   The AirlineEIRecord is derived from DCEIRecord. This DCFieldContainer should map all fields available in the input data. We know that from FlightInfoSender we have one passenger record per datagram packet. A single flight record is a byte array of 7 integer attributes. These attributes are (in their respective order):

   - Passenger ID
   - Flight Number
   - Departing Airport ID
   - Arriving Airport ID
   - Departure Time in Seconds
   - Arrival Time in Seconds
   - Number of Bags of Luggage for Passenger

   AirlineEIRecord should have the knowledge to extract each of the 7 integer attributes and map it into a DCField. Therefore, an AirlineEIRecord should contain totally 7 DCFields. Each DCField should have its ID, type, and value. For example, the raw data “Passenger ID" will map to an IntField, whose ID will be “Passenger_ID", type will be Integer; and value will be the integer value of the raw data. The IDs of these 7 DCFields will be listed as the InputRec in NPL of AirlineEINode.

   - AirlineEIRecord should at least implement the following methods defined in DCEIRecord:
   - DCField getField(String ID) - Giving the ID of a DCField, we should be able to retrieve the DCField itself.
   - byte[] getFieldValue(String ID) - Giving the ID of a DCField, we should be able to retrieve the value of this DCField as a byte array.
   - int getType(String ID) - Giving the ID of a DCField, we should be able to retrieve the type of this DCField.
   - String toString() - We should be able to get the string representation of this EIRecord.

- byte[] toByteArray() - We should be able to get the binary representation of this EIRecord.

5. Implement the Node.

   Now we will implement the AirlineEINode itself. We have a template, EINodeTmpl.java, which can be used to start creating the AirlineEINode.

   > **Note:**
   >
   > AirlineEINode will implement a warm restart capability, which means that this node must implement startup() and reconfigure() these two methods.

   a. Construct AirlineEINode with warm restart implemented.

   b. Construct the node with no operation.

   ```
   public AirlineEINode( String[] args ) throws NodeStartException {
               super(args);
   }
   ```

   c. Override method warmRestartImplemented() to return true.

   d. Implement startup() method.

   - Get all the config information of this node.
   - Construct the UdpEITransport as described in step 2, which will utilize the record factory created in Step 3 in "Create AirlineEINode (CC Node)".

   ```
   // Create record factory
    String nid = getNodeId();
    recordFactory = new AirlineRecordFactory(nid, this);

    // Create the Transport
    UdpEITransport eiTransport = new UdpEITransport(this,
                                                        recordFactory,
                                                        udpListenPort,
                                                        maxPacketSize,
                                                        queueSize);
   setTransport(eiTransport);
   ```

   e. Construct an NPLFieldProcessor.

   ```
   // Set the FieldProcessor
   Class NARClass = (new NAR()).getClass();
   try {
            setFieldProcessor(new NPLFieldProcessor(NARClass, this,
                                   scratchDir.getAbsolutePath(), NPLFile)
                                  );
   }
   catch (NodeProcessingException npex)
   {
            throw new NodeStartException(npex.getMessage());
   }
   ```

   f. Set up appropriate receivers and providers relationship.

> **Note:**
>
> UdpEITransport requires a "pull" data transfer model.

```
getFieldProcessor().setDataProvider(getTransport());
getFieldProcessor().setDataReceiver(getDCStreamHandler());
```

g. Start the appropriate components.

```
// Start the threads
setTransportThread(getTransport());
getTransportThread().start();
setFieldProcessorThread(getFieldProcessor());
getFieldProcessorThread().start();
```

6. Implement reconfigure() method so that node can be reconfigured without stopping it. Here are the steps that should be performed during reconfiguration:
    a. Set the state of transport to begin reconfiguration.
    b. Shut down Field Processor and DCStreamHandler.
    c. Start reconfiguration:
        - Call super.reconfigure().
        - Get the new config information.
        - Reconfigure the transport.
        - Re-instantiate Field Processor.
        - Re-establish the receivers and providers relationship.
        - Start Field Processor.
    d. Finally set the state of transport to end reconfiguration and set the flag of this node to be not in reconfiguration state.
7. Implement remaining abstract methods:
    a. GetMinorType(): return the minor type of this node, which will be “Airline" in this case.
    b. getConfigGUIClass(): return the package name of the config GUI class of this node. We will implement the config GUI class for this node later.
8. Implement an “orderly" shutdown method.

    Shutdown method is very important for a node. When you are writing the shutdown() method:

    - Make sure components are shutdown in the proper order.
    - Make sure all threads are really stopped.

    Usually the shutdown order will be like this:

    a. Call super.shutdown().
    b. Shut down the transport.
    c. Shut down the Field Processor.
    d. Shut down the DCStreamHandler.
9. Implement a main() method for testing purpose.

10. Write the NPL file for this node. In a CC node this should be primarily mapping to NAR attribute. Following is the NPL file, **airline.npl**, for AirlineEINode:

    ```
    InputRec {
      Integer Passenger_ID;
      Integer Flight_Number;
      Integer Depart_Airport_ID;
      Integer Arrive_Airport_ID;
      TimeInSecs Depart_TIS;
      TimeInSecs Arrive_TIS;
      Integer Number_Bags;
    } in;

    OutputRec {
      Integer 0;
      Integer 1;
      Integer 2;
      Integer 3;
      TimeInSecs 4;
      TimeInSecs 5;
      Integer 6;
    } out;

    out.0 = in.Passenger_ID;
    out.1 = in.Flight_Number;
    out.2 = in.Depart_Airport_ID;
    out.3 = in.Arrive_Airport_ID;
    out.4 = in.Depart_TIS;
    out.5 = in.Arrive_TIS;
    out.6 = in.Number_Bags;

    write(out);
    ```

11. Implement the NodeConfigGUI class.

    We now implement the config GUI class, AirlineEINodeCOnfigGUI, for AirlineEINode. We have a template config GUI class, **EINodeConfigGUITmpl.java**, for reference.

    When we write the config GUI class, we need to identify what kind of config information we need for the node and what kind of node (CC, DC, or Processor). For CC node, its config GUI class should be derived from EINodeConfigGUI; for DC node, it should be derived from OINodeConfigGUI; for Processor node, it should be derived from ProcessorNodeConfigGUI.

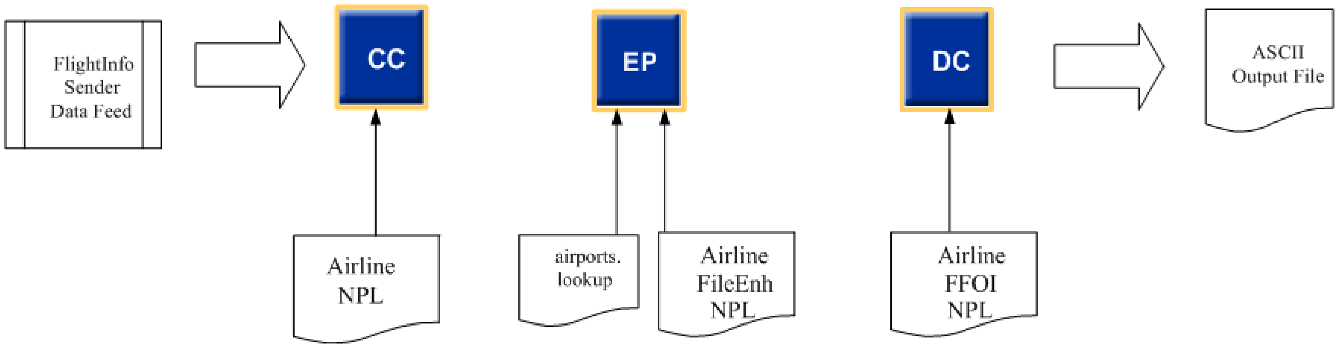
    Next, we need to identify the necessary config information for this node. AirlineEINode is collecting the airline passenger data via UDP. We need to know the UDP port number, as well as the size of the queue used in the transport. So we need to add two TextFields in the GUI to allow users to specify the UDP port number and queue size. Now follow the template, we should be able to implement AirlineEINodeConfigGUI class.

Now AirlineEINode is finished. We need to work on EP nodes and DC nodes. When all the nodes are created, we will modify/create the appropriate property files. So we can create those nodes with Node Creation Wizard.

Create Node Chain

You have finished creating the AirlineEINode. You still need to create the EP node and DC nodes to set up a node chain to process the airline passenger data.

Figure 4-2 Creating Node Chain



The EP node will read information from a file to add two additional attributes to each NAR:

- Departure Airport Name
- Arrival Airport Name

These values will be read from an **airports.lookup** file which maps the (integer) Airport ID to the Airport Name. The EP node will be the FileEnhancer node, which is included in the CDK.

The **airports.lookup** file contents:

```
1000=Logan International Airport
1001=O'Hare International Airport
1002=John F. Kennedy International Airport
1003=Orlando International Airport
1004=Baltimore/Washington International Airport
1005=Bangor International Airport
1006=Portland International Airport
1007=St. Louis Regional Airport
1008=Honolulu International Airport
1009=Austin-Bergstrom International Airport
1010=Dallas/Fort Worth International Airport
1011=LaGuardia International Airport
1012=Los Angeles International Airport
1013=San Francisco International Airport
1014=Manchester Airport
1015=Seattle-Tacoma International Airport
1016=Juneau International Airport
```

The DC node will output all records to an ASCII file. The DC node will be FFOINode, which is also included in the CDK.

In this case, we only need to write NPL rules files for both EP and DC nodes.

1. Write an NPL file for the EP node that:
   - Imports FileEnhMethodHandlerIfc.
   - Uses a corresponding “tablename" value as one that will be entered in the client GUI. Your choice, just be consistent. (i.e., the “tablename" value that the method handler used must be the same value as that entered in the GUI).
   - Uses the method Java.lookup(StringField tablename, StringField attribute) to look up the Departure Airport Name and Arrival Airport Name.

     Following is the NPL file, **AirPortEnh.npl**, for the EP node:

```
import com.nt.udc.processor.FileEnhancer.FileEnhMethodHandlerIfc;

String table_name = "airports";

InputRec {
  Integer 0;     // Passenger_ID
  Integer 1;     // Flight_Number
  Integer 2;     // Depart_Airport_ID
  Integer 3;     // Arrive_Airport_ID
  TimeInSecs 4;  // Depart_TIS  (time in seconds)
  TimeInSecs 5;  // Arrive_TIS  (time in seconds)
  Integer 6;     // Number_bags
} in;
//  The output record only needs to have the two new attributes defined.
OutputRec {
  String 7;      // Departing Airport Name
  String 8;      // Arriving Airport Name
} out;

//  Record to record assignment
out = in;

//  Use the "javahook" lookup method to fetch the airport names using their IDs
out.7 = Java.lookup(table_name,int2str(in.2));
out.8 = Java.lookup(table_name,int2str(in.3));

write(out);
```

2. Write the NPL file, **ffoi_airline.npl**, for the DC node that maps all of the NAR attributes from the input record (total of 9) to the output record.

Now, it is time to modify/create the node template definitions.

1. First, we add the new Airline nodes to the **nodeTemplateDef.xml** file in the customization directory (*OMC_home***/customization/nodeTemplateDef.xml**), where *OMC_home* is the directory in which you installed Offline Mediation Controller.

```
<nodetemplate id="Airline#EI#AirlineEINode">
    <name>Airline CC</name>
    <property name="Solution">cartridgekit</property>
    <rule id="airline.npl">
<name>Airline Passenger</name>
        <nplfile>airline.npl</nplfile>
    </rule>
    <nodeclass>com.nt.udc.ei.node.airline.AirlineEINode</nodeclass>
</nodetemplate>

<nodetemplate id="Airline#EP#AirlineEPNode">
    <name>Airport Name EP</name>
    <property id="Solution">cartridgekit</property>
    <rule id="AirportEnh.npl">
        <name>Airport Name</name>
        <nplfile>AirportEnh.npl</nplfile>
    </rule>
    <nodeclass>com.nt.udc.processor.FileEnhancer.FileEnhancerNode</nodeclass>
</nodetemplate>

<nodetemplate id="Airline#OI#AirlineOINode">
    <name>ASCII DC</name>
    <property name="Solution">cartridgekit</property>
    <rule id="ffoi_airline.npl">
        <name>Airline</name>
        <nplfile>ffoi_airline.npl</nplfile>
    </rule>
    <nodeclass>com.nt.udc.oi.node.flatfile.FFOINode</nodeclass>
</nodetemplate>
```

2. Next, we will add the node templates to the GUI using the **nodeGroupDef.xml** file in the customization directory (*OMC_home***/customization/nodeGroupDef.xml**).

   ```
   <nodetemplategroup>
       <name>Airline Example</name>
       <nodetemplategroup>
           <name>Collection Cartridge (CC)</name>
           <nodetemplate id="Airline#EI#AirlineEINode"/>
       </nodetemplategroup>
       <nodetemplategroup>
           <name>Enhancement Processor (EP)</name>
           <nodetemplate id="Airline#EP#AirlineEPNode"/>
       </nodetemplategroup>
       <nodetemplategroup>
           <name>Distribution Cartridge (DC)</name>
           <nodetemplate id="Airline#OI#AirlineOINode"/>
       </nodetemplategroup>
   </nodetemplategroup>
   ```

3. Finally, add the NPL rule files:

   a. Create directory *OMC_home***/rules/Airline/EI/Airline** and copy **airline.npl** to this directory.

   b. Create directory *OMC_home***/rules/Airline/Processor/FileEnhancer** and copy **AirPortEnh.npl** to this directory.

   c. Create directory *OMC_home***/rules/Airline/OI/FlatFile** and copy **ffoi_airline.npl** to this directory.

At this point, we have finished creating/modifying the appropriate node template files. We are able to start the client GUI and use Node Creation Wizard to create our Airline Flight node chain.

Starting FlightInfoSender Simulator

Before we can start our Airline Flight node chain, we need to start the FlightInfoSender simulator, which will send the simulated airline passenger data via UDP.

This simulator is intended strictly for generating data for the CDK Course Workshop. It generates fictional “flight" data.

- The FlightInfoSender program sends data over TCP port number 2112
- This program can be run from the command line as follows:

  ```
  $JAVA_HOME/bin/java -classpath OMC_home/web/htdocs FlightInfoSender OMC_home/web/
  htdocs/DestIps.dat 10
  ```

  where:

  – *OMC_home* is the directory in which you installed Offline Mediation Controller.

  – **DestIps.dat** is the file that lists the IP addresses of the machines to send the flight data to (One (1) IP Address per line). The default version of this file lists only “localhost". If this does not work for you, or you want to send data to one or more different machines, replace localhost with the IP addresses of the desired machines.

  – **10** is the delay in milliseconds between records being sent.

The simulator runs in an endless loop, generating distinct passenger records up to a point:

- Each record has a unique passenger ID, generated sequentially, starting at 1.
- Flight numbers start at 1 and are incremented every 100 passengers.
- A departing and arriving airport are generated randomly when a new flight number is created. The Airport IDs are pulled from a short list of “valid" airport IDs. A lookup file maps these airport IDs from integers to names. Use the lookup file for the FileEnhancer. However, the IDs used by the flight sender are hard-coded and would require a recompile of the source code to add or change values.

This is the end of the CDK workshop exercise. You can now start the Airline Flight node chain and start collecting the airline passenger data.

5

Transferring Custom Node Chains

This chapter describes how to transfer custom node chains from one Oracle Communications Offline Mediation Controller system to another. You can transfer the configuration of a system with a custom node or nodes to another system that may or may not have custom nodes.

There are two scenarios described in this chapter:

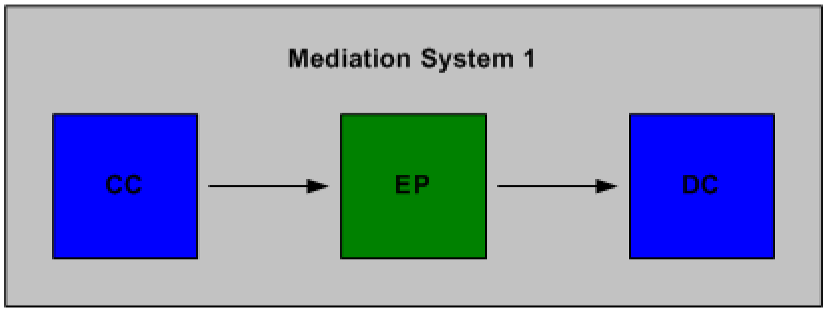
- Scenario 1 describes how to transfer a configuration with custom nodes to an Offline Mediation Controller system that has an identical base node chain
- Scenario 2 describes how to transfer a configuration with custom nodes and a secondary custom node chain to an Offline Mediation Controller system that has an identical base node chain

In both scenarios, the goal is to update System 1 with the System 2 node chain.

Scenario 1

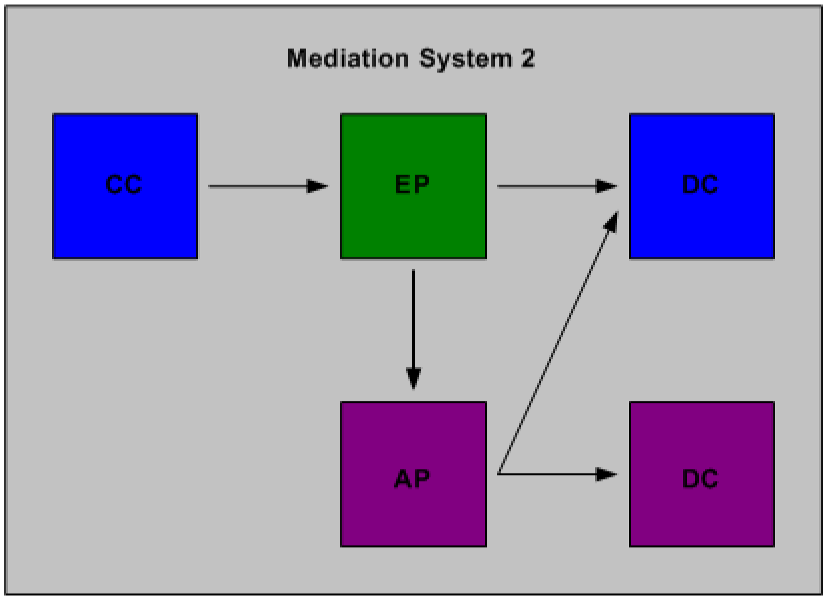
In Scenario 1, System 1 is the original Offline Mediation Controller system in the customer environment, as shown in Figure 5-1. The core nodes are shown in blue, and a custom node is shown in green.

Figure 5-1 Original Offline Mediation System 1



On System 2, a node chain has been verified in the lab and is ready to be deployed in the customer environment (System 1). System 2 has the identical base node chain as System 1, with two new custom nodes, shown in purple in Figure 5-2.

Figure 5-2 Offline Mediation System 2



Updating the node chain

Use the following procedure to update System 1 with the System 2 node chain.

To update System 1 with the System 2 node chain

On System 2

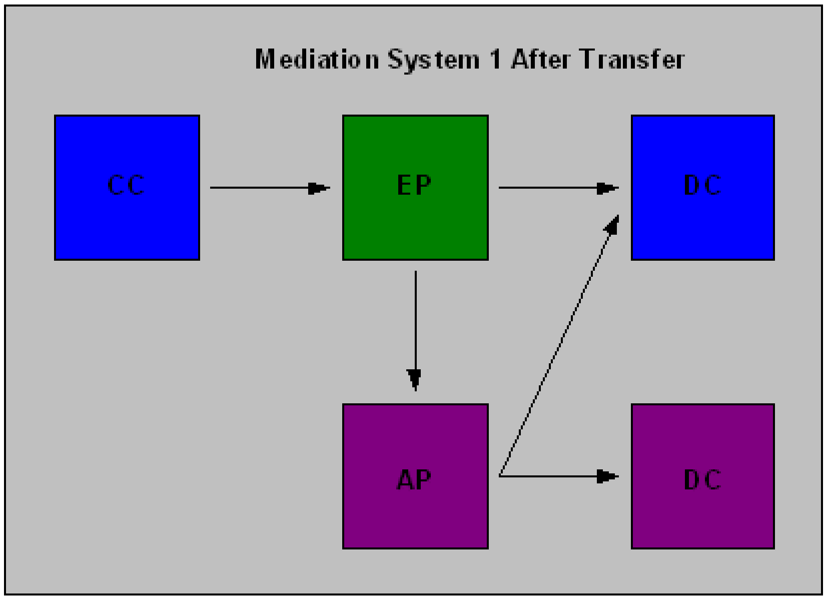
To export the node configuration, tar the following directories:

- *OMC_home_1*/**customization**
- *OMC_home_2*/**cartridges**

On System 1

1. Ensure all data in the node chain has been processed, then delete the node chain.
2. Stop the Node Manager, Administration Server, and GUI.
3. On System 1, untar the *OMC_home_2*/**cartridges** directory into the *OMC_home_1*/**cartridges** directory.
4. On System 1, untar the *OMC_home_2*/**customization** directory into the *OMC_home_1*/**customization** directory.
5. Start the Node Manager, Administration Server, and GUI.
6. Import the configuration from System 2.

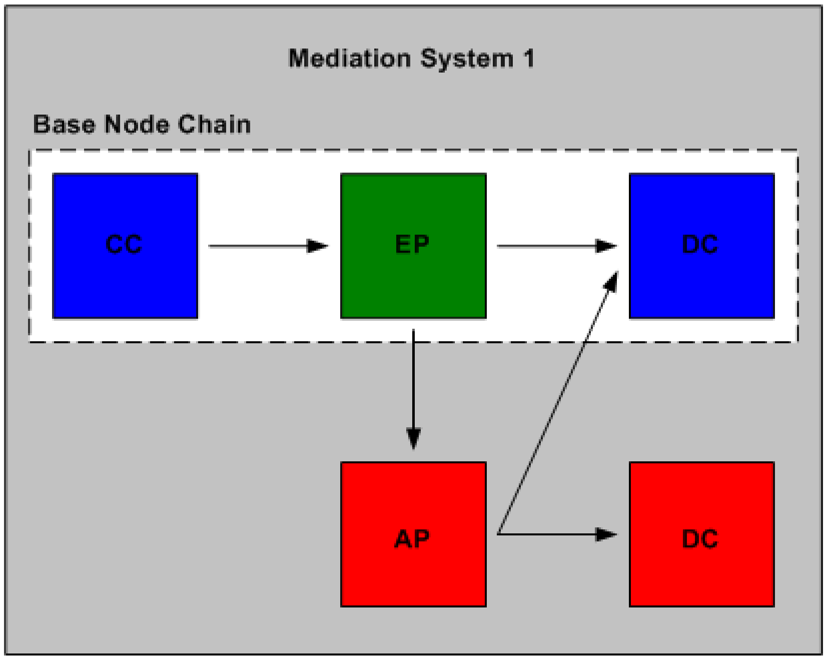
   System 1 now appears as in the following diagram.



Scenario 2

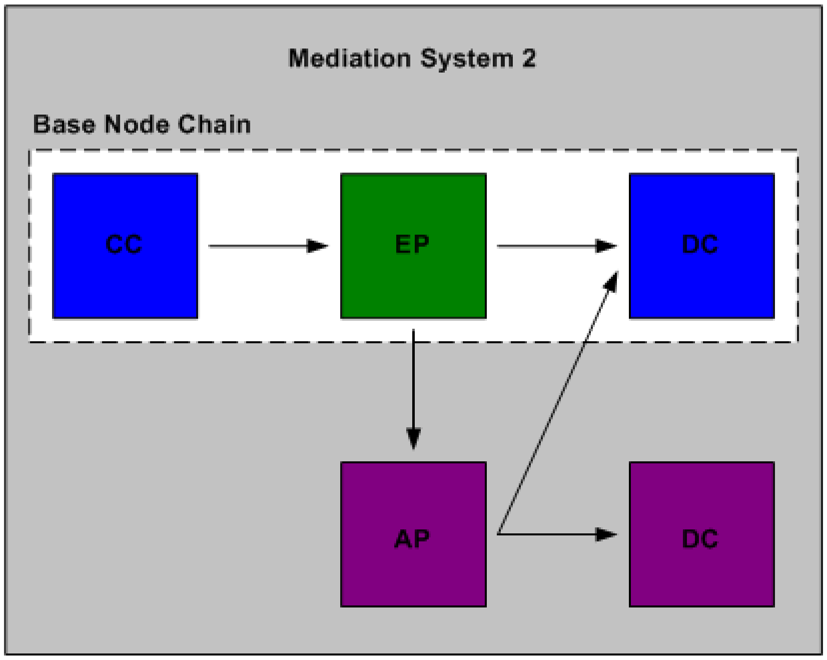
In Scenario 2, System 1 has a base node chain and a secondary node chain that contains custom nodes, shown in red in Figure 5-3.

Figure 5-3 Offline Mediation System 1



System 2 has the same base node chain as System 1 but with a secondary node chain that contains custom code, shown in purple in Figure 5-4.

Figure 5-4 Offline Mediation System 2



Updating the node chain

Use the following procedure to update System 1 with the System 2 node chain while preserving the secondary node chain on System 1.

To update System 1 with the System 2 node chain

On System 2

1. Open the following file: *OMC_home_2*/**customization/nodeTemplateDefs/nodeTemplateDef.xml**.
2. Locate and open the same file on System 1.
3. Examine the contents of the two files and ensure the contents of both files are captured in **nodeTemplateDef.xml** on System 2. Remove any duplication.
4. Open the following file: *OMC_home_2*/**customization/groupTemplateDefs/groupTemplateDef.xml**.
5. Locate and open the same file on System 1.
6. Examine the contents of the two files and ensure the contents of both files are captured in **groupTemplateDef.xml** on System 2. Remove any duplication.
7. To export the configuration, tar the following directories:
   - *OMC_home_2*/**customization**
   - *OMC_home_2*/**cartridges**

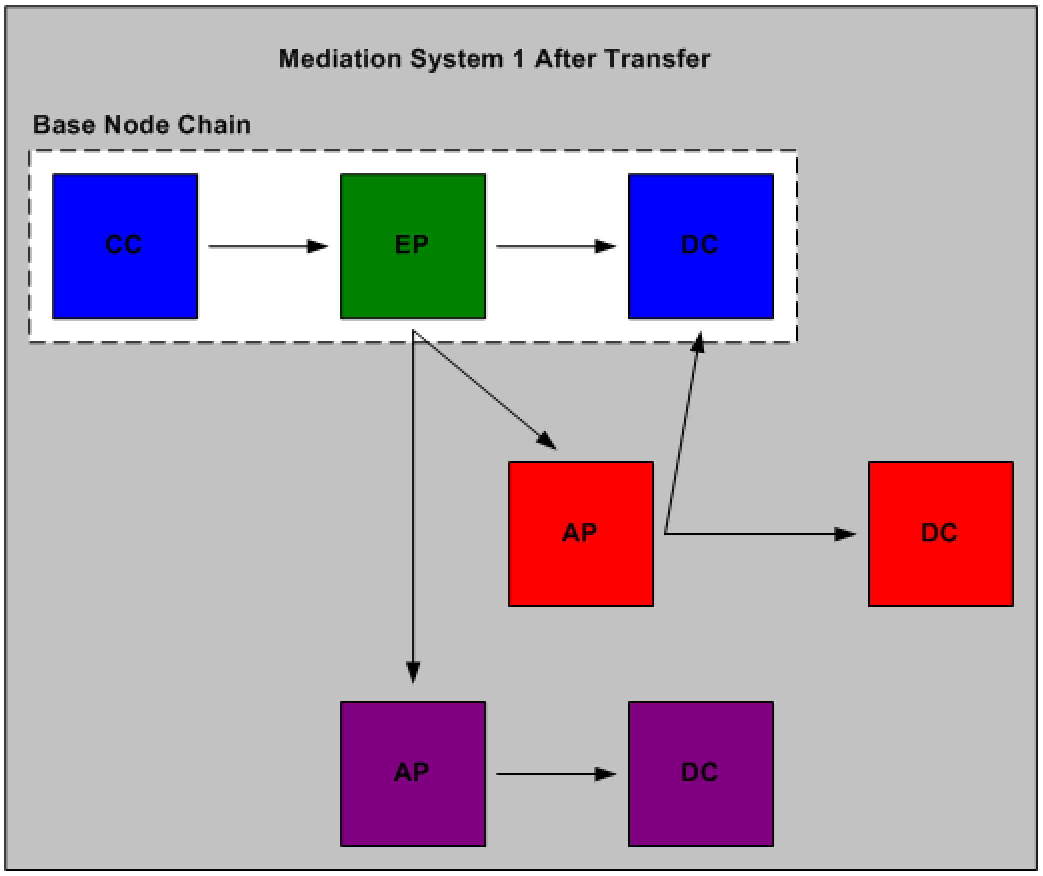
On System 1

1. Ensure all data in the node chain has been processed. Then, delete the routing link between the EP node on the base node chain and the AP node on the secondary node chain.

2. Delete the base node chain.
3. Stop the Node Manager, Administration Server, and GUI.
4. On System 1, untar the following directories into the following locations:
   - *OMC_home_2*/**cartridges** into *OMC_home_1*/**cartridges**
   - *OMC_home_2*/**customization** into *OMC_home_1*/**customization**
5. Start the Node Manager, Administration Server, and GUI.
6. Import the configuration from System 2.

   After the update, there is no routing link between the EP node of the base node chain and the AP node of the secondary node chain. You must manually restore this link via the Administration GUI.

   System 1 now appears as in the following diagram.



6

Debugging Tools and Tips

This chapter describes the utilities and tips that may be used for debugging purposes when developing new nodes in Oracle Communications Offline Mediation Controller. The source code for **NARMaker.java** and **NARViewer.java** is available under the “Utilities" section of the CDK Development LiveLink site.

Running Nodes from the Command Line

You can run a node from the command line as a Java application, assuming the main() method has been implemented. The basic syntax for running a node from the command line is:

```
java fullyQualifiedClassName nodeID OMC_home configFile
```

where:

- *nodeID*: The unique string identifier for this node.
- *OMC_home*: The location of Offline Mediation Controller installation, such as ***/ocomc***.
- *configFile*: The file name and location of this node's configuration file. Configuration files are generated by the Offline Mediation Controller Administration Client GUI and are generally located in *OMC_home***/config/***nodeID***/general.cfg**.

The basic contents of the **general.cfg** file are:

```
monitorinputtimeunit 'Day'
numchannels '0'
osarsperfile '2000'
rulesfile 'file:@TEST_OCOMC_HOME@/npl/?
class=com.nt.udc.rules.Processor.NPL.PRCDemo_npl'
debuglevel 'OFF'
maxlogfilesize '100000'
backup 'false'
checkforosarstimer '1'
idleosarwritetime '1'
recordStatistics 'false'
monitorinput 'false'
asCodebase 'file:@TEST_OCOMC_HOME@'
secondaryConfigFile 'secondary.cfg'
narbackup 'false'
narbackupdays '7'
monitorinputtime '1'
```

When running the node from the command line, the fully qualified class name should be specified by the rulesfile parameter. (That is the name of the class that was generated by the NPLCompiler).

NAR Viewer

Use the NARViewer to view the contents of the binary NAR files produced by a node. NAR files are found in the node's output directory (*OMC_home***/output/***node_id***/**, where *OMC_home* is the directory in which you installed Offline Mediation Controller). This utility allows developers to verify the contents of the data that the node is producing.

The NARViewer is in the **com.nt.udc.general** package in the **nodes.jar** file.

```
java com.nt.udc.general.NARViewer NAR_file Output_file
```

where:

- *NAR_file*: The file name of the NAR file to view.
- *Output_file*: The name of file to print contents to (in ASCII).

Note:

The above usage statement assumes that the *OMC_home***/web/htdocs/nodes.jar** is included in the CLASSPATH environment variable.

NAR Generator

There is a template NAR Generator class available. To provide meaningful data, developers need to modify the template or create a new class, according to their own requirements.

The sample NARMaker class, as described below, will generate the specified number of NARs in a file called **nars.dat**. The NARMaker will create NARs that include the following fields:

Table 6-1 NARs Created by NARMaker

ID	Type	Value
2	MillisField	<timestamp>
3	IntField	<Counter starting at 0>
4	StringField	"Test field"

The sample NARMaker class is in the **com.nt.udc.general** package in the **nodes.jar** file.

```
java com.nt.udc.general.NARMaker #NARs
```

where *#NARs* is the number of NARs to generate.

Note:

The above usage statement assumes that the *OMC_home***/web/htdocs/nodes.jar** file is included in the CLASSPATH environment variable.

The source code for the NARMaker class is included here for illustration purposes.

Example 6-1   NARMaker.java

```
·package com.nt.udc.general;
·
·import java.io.*;
·import java.util.*;
·import com.nt.udc.ndk.node.*;
·import com.nt.udc.nar.*;
·
·/**
· * This is a _very_ basic NAR file generator. It is intended to be
· * used as a template to generate specific NAR's that match the appropriate
· * node's rules. Modify this code as needed.
· *
· * In this example, the generated NAR's will hold the following fields
· * and values:
· *
· *   Field 2 - Time interval
· *   Field 3- Integer count
· *   Field 4- String
· */
·public class NARMaker
·{
·    /** Main method which takes 1 argument:
·     *    args[0]    number of NARs to generate   */
·    public static void main( String [] args )
·    {
·        int numNars = 0;
·
·        if(args.length != 1)
·        {
·            String usage = "Usage: \n" +
·                           "  java com.nt.udc.nar.NARMaker <num_nars>";
·            System.out.println( usage );
·            return;
·        }
·
·        try
·        {
·            numNars = Integer.parseInt( args[0] );
·        }
·        catch( Exception e )
·        {
·            System.out.println( "Exception: " + e );
·        }
·
·        // Create the NAR, and set up the initial attribute values
·        NAR nar = new NAR();
·
·        // Create 3 DCField objects to put into the NAR.
·        DCField[] fields = new DCField[3];
·
·        // Create a dummy time field.
·        long time = System.currentTimeMillis();
·        fields[0] = new MillisField( 2, time );
·        // Create a integer count
·        fields[1] = new IntField( 3, 0 );
·        // Create a sample String field
·        fields[2] = new StringField( 4, "Test field" );
·
·        // Put the fields in the NAR
·        for( int i = 0; i < fields.length; i++ )
·        {
·            nar.setField( fields[i] );
·        }
·
·        try
·        {
·            // Use the file called "nars.dat"
·            FileOutputStream fos = new FileOutputStream( "nars.dat" );
·
·            // Loop for "numNars", and create a new NAR.
·            for( int i = 0; i < numNars; i++ )
·            {
·                // Update the time field.
·                fields[0] = new MillisField( 2, time + ( i * 100 ) );
·
·                // Update the integer count
·                fields[1] = new IntField( 3, i );
·
·                nar.setField( fields[0] );
·                nar.setField( fields[1] );
·
·                nar.toStream( fos );
·            }
·        }
·        catch( Exception ex )
·        {
·            System.out.println( "Exception caught : " + ex );
·        }
·    }
·}
```