SeeBeyond ICAN Suite

SNA eWay Intelligent Adapter User's Guide

Release 5.0.0

SeeBeyond Proprietary and Confidential

The information contained in this document is subject to change and is updated periodically to reflect changes to the applicable software. Although every effort has been made to ensure the accuracy of this document, SeeBeyond Technology Corporation (SeeBeyond) assumes no responsibility for any errors that may appear herein. The software described in this document is furnished under a License Agreement and may be used or copied only in accordance with the terms of such License Agreement. Printing, copying, or reproducing this document in any fashion is prohibited except in accordance with the License Agreement. The contents of this document are designated as being confidential and proprietary; are considered to be trade secrets of SeeBeyond; and may be used only in accordance with the License Agreement, as protected and enforceable by law. SeeBeyond assumes no responsibility for the use or reliability of its software on platforms that are not supported by SeeBeyond.

SeeBeyond, eGate, and eWay are the registered trademarks of SeeBeyond Technology Corporation in the United States and select foreign countries; the SeeBeyond logo, e*Insight, and e*Xchange are trademarks of SeeBeyond Technology Corporation. The absence of a trademark from this list does not constitute a waiver of SeeBeyond Technology Corporation's intellectual property rights concerning that trademark. This document may contain references to other company, brand, and product names. These company, brand, and product names are used herein for identification purposes only and may be the trademarks of their respective owners.

© 2005 by SeeBeyond Technology Corporation. All Rights Reserved. This work is protected as an unpublished work under the copyright laws.

This work is confidential and proprietary information of SeeBeyond and must be maintained in strict confidence.

Version 20050121155955.

Contents

Chapter 1

Introduction 7

About SNA 7
Supported Logical Unit Types 10
SNA LU6.2 10
About the SNA eWay 11
What's New in This Release 12
Collaborations 12
Persistence 12
Conversation State 12
Protocol State 13
About This Document 13
What's in This Document 13
Scope 13
Intended Audience 14
Document Conventions 14
Related Documents 14
SeeBeyond Web Site 15
SeeBeyond Documentation Feedback 15

Chapter 2

Installing the eWay 16

Supported Operating Systems 16
System Requirements 16
Supported External Applications 17
Installing the eWay Product Files 17
After Installing the eWay 19

Chapter 3

Configuring the eWay 20

Inbound Connectivity Map Properties 20
Inbound Schedules 21
Listner Schedule 21
Service Schedule 22
Connection Establishment 24
Max Connection Retry 24
Retry Connection Interval 24
Inbound Connection Manager 24
Connection Pool Size 24
Scope of Connection 25
SNA Settings 25
Custom Handshake Class Name 25
Deallocation Type 25
Initialize Conversation 26
Packet Size 26
Synchronization Level 26
Timeout 26
General Settings 26
Scope of State 26

Outbound Connectivity Map Properties 27
Connection Establishment 27
Always Create New Connection 27
Auto Disconnect Connection 28
Auto Reconnect Upon Matching Failure 28
Max Connection Retry 28
Retry Connection Interval 28
SNA Settings 29
Custom Handshake Class Name 29
Deallocation Type 29
Initialize Conversation 29
Packet Size 29
Synchronization Level 30
Timeout 30
General Settings 30
Scope of State 30

Inbound Environment Properties 30
SNA Settings 31
Host Name 31
Local LU Name 31
Local TP Name 31
Symbolic Destination Name 32
General Settings 32
Persistent Storage Location 32

Outbound Environment Properties 32
SNA Settings 33
Host Name 33
Local LU Name 33

Local TP Name 33
Symbolic Destination Name 33
General Settings 33
Persistent Storage Location 33
Object Type Definitions (OTDs) 34

Chapter 4
SNA Java Collaborations 35
Creating Default Java Collaborations 35
Analyzing the Default Java Collaboration 36
Creating Custom Collaborations 39
Inbound SNA Conversations 39
Conversation Sent to a Text File (CPIC) 40
Conversation Sent to a Text File (Helper) 42
Outbound SNA Conversations 45
Conversation Originates from a Text File (CPIC) 45
Conversation Originates from a Text File (Helper) 48
Inbound and Outbound SNA Conversations 50
Best Practices 50
Checking Conversation State 51
Using CPIC Calls 53

Chapter 5
Implementing SNA eWay Projects 54
About the Sample Projects 54
Sample Project Contents 54
Sample Project Zip Files 55
Locating the Sample Projects 55
Importing Projects 55
Running SNA eWay Projects 57
Creating the Environment Profile 57
Configuring the Logical Host 58
SPARC64 logical host deployment 59
Deploying the Project 60
Running the Sample Project 61
Windows 2000/XP/Windows Server 2003 61
IBM AIX 5.1L and 5.2 (32-bit) 62
IBM AIX 5.1L and 5.2 (64-bit) 62
Sparc (32-bit) 62
Sparc (64-bit) 63
Building SNA Business Logic with eGate 63
Building Collaborations 63
Adding Connectivity Maps 66
Building Inbound Connectivity Maps 66

Building Outbound Connectivity Maps 67
SNA Collaborations 68

Chapter 6
SNA eWay Javadocs 69

Index 1

Chapter 1

Introduction

This chapter introduces you to the *SNA eWay Intelligent Adapter User's Guide*, its general purpose and scope, and its organization. It also provides sources of related documentation and information.

What's In This Chapter:

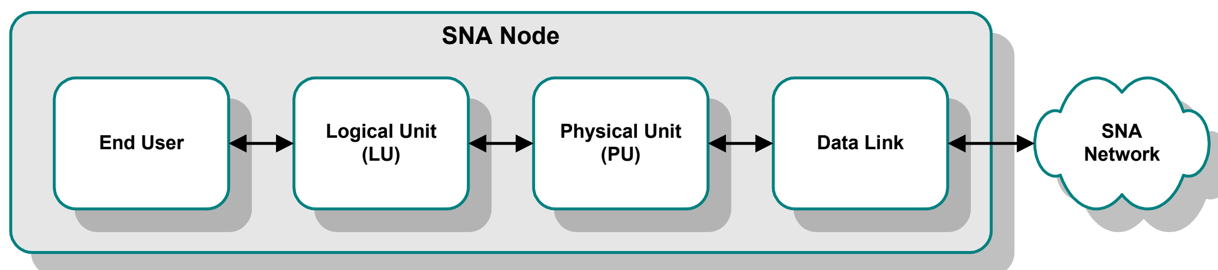
- **"About SNA" on page 7**
- **"About the SNA eWay" on page 11**
- **"What's New in This Release" on page 12**
- **"About This Document" on page 13**
- **"Related Documents" on page 14**
- **"SeeBeyond Web Site" on page 15**
- **SeeBeyond Documentation Feedback** on page 15

1.1 About SNA

SNA (System Network Architecture) is a data communications architecture developed by IBM to specify common conventions for communication between various IBM hardware and software products. It is specifically designed to address issues of reliability and flexibility of sharing data between components and their peripherals. Many vendors other than IBM also support SNA, allowing their products to interact with SNA networks.

An addressable unit on an SNA network is called a node, and is made up of four functional components forming a hierarchy as shown in Figure 1.

Figure 1 SNA Node Architecture

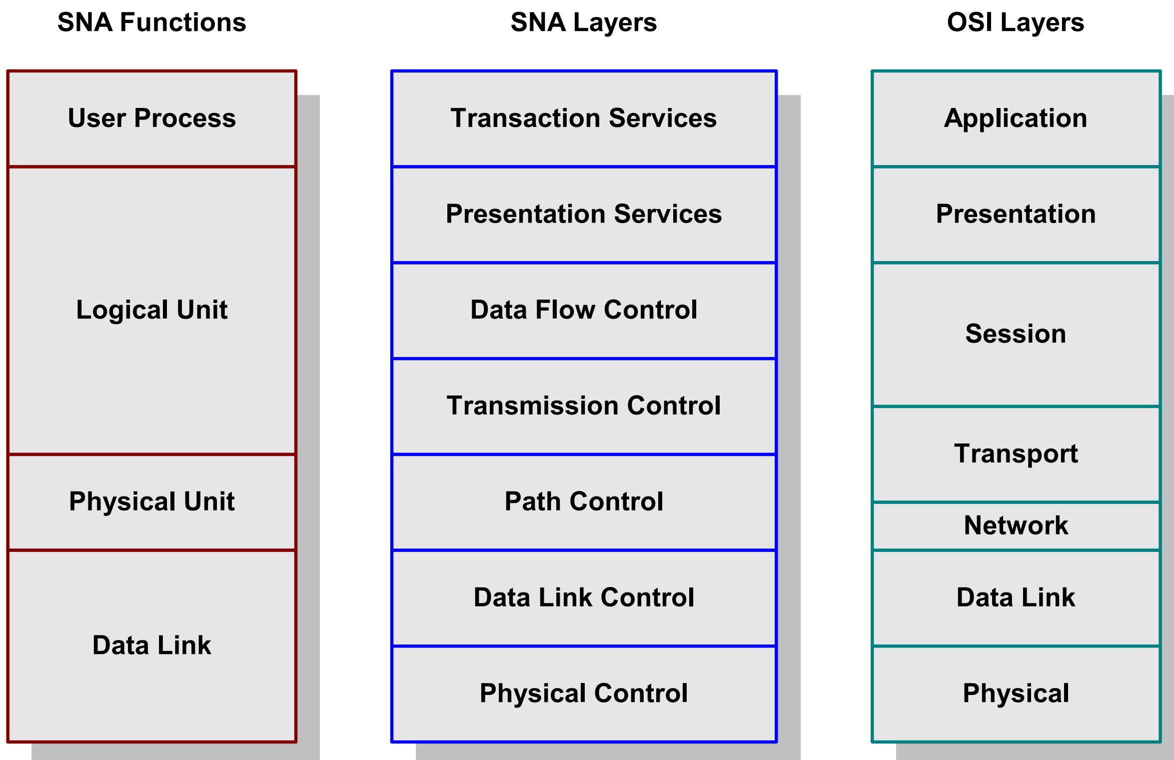


To establish a communications session, SNA uses Logical Units (LUs) as entry points into the network. There are several types of LUs, currently type 0 through type 6.2. Most of the LU types are specific to IBM operating environments, but type 6 is intended for use in a distributed data processing environment.

Generally, an LU can communicate only with another LU of the same type, but specific exceptions to this rule exist with type 6.2. LU6.2 is the least-restrictive of the various LU types, and also supports multiple concurrent sessions. As a result, it is the LU most widely supported by other system vendors.

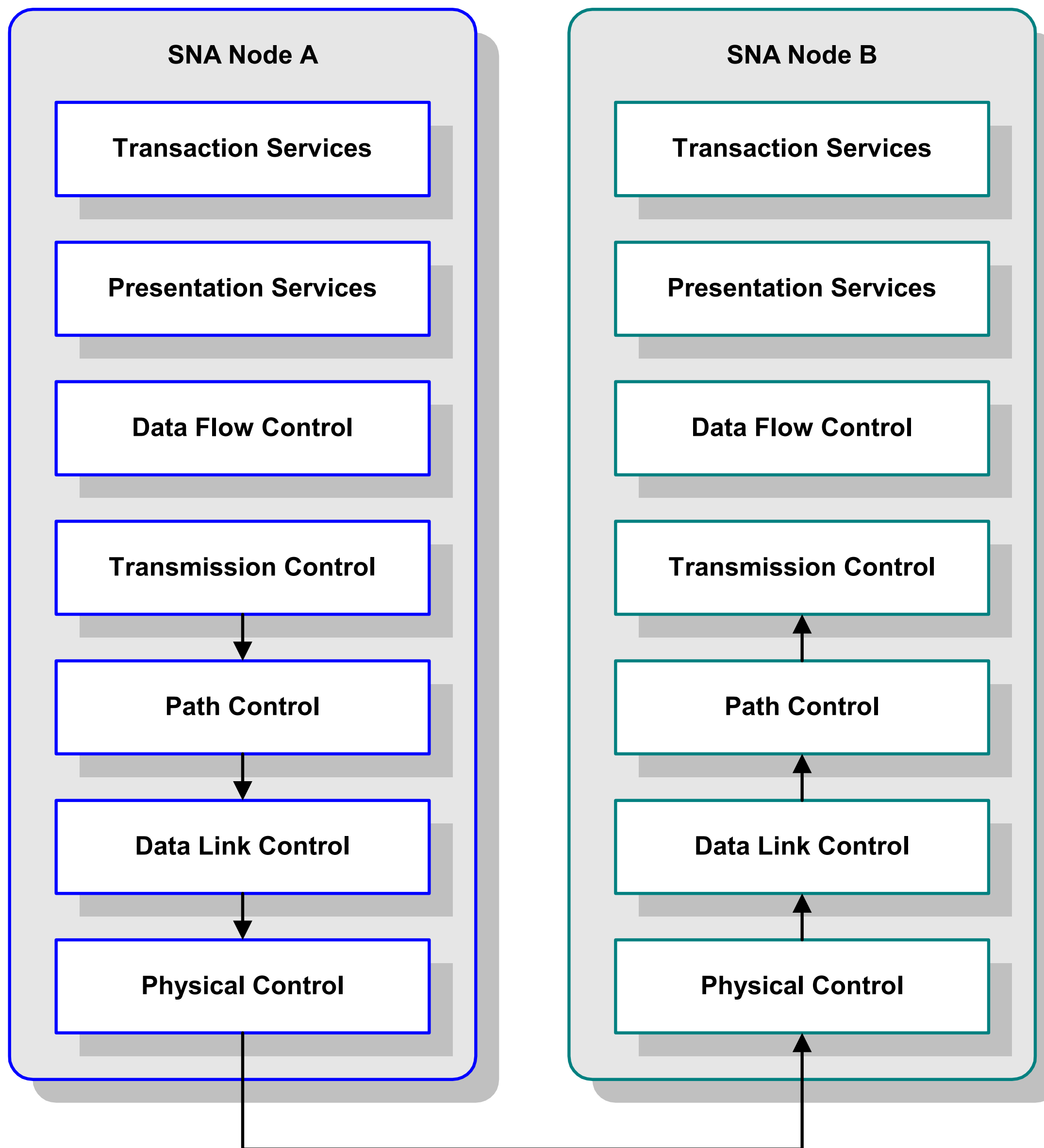
Like the OSI model, SNA functions are divided into seven hierarchical layers, but the layers are not identical. Their relationships to each other, and to the SNA node functionality, are shown in Figure 2. The Transport Network handles the lower three layers, while the Network Accessible Units (NAU) implement the upper four layers by using the services of the Transport Network to establish communication between nodes.

Figure 2 SNA Functional Layers



SNA defines formats and protocols between these layers that allow equivalent layers in different nodes to communicate with each other. Also, each layer provides services to the layer above, and requests services from the layer below. As an example, the communication path between two Transmission Control layers would appear as shown in Figure 3.

Figure 3 Equivalent-Layer Communications Path



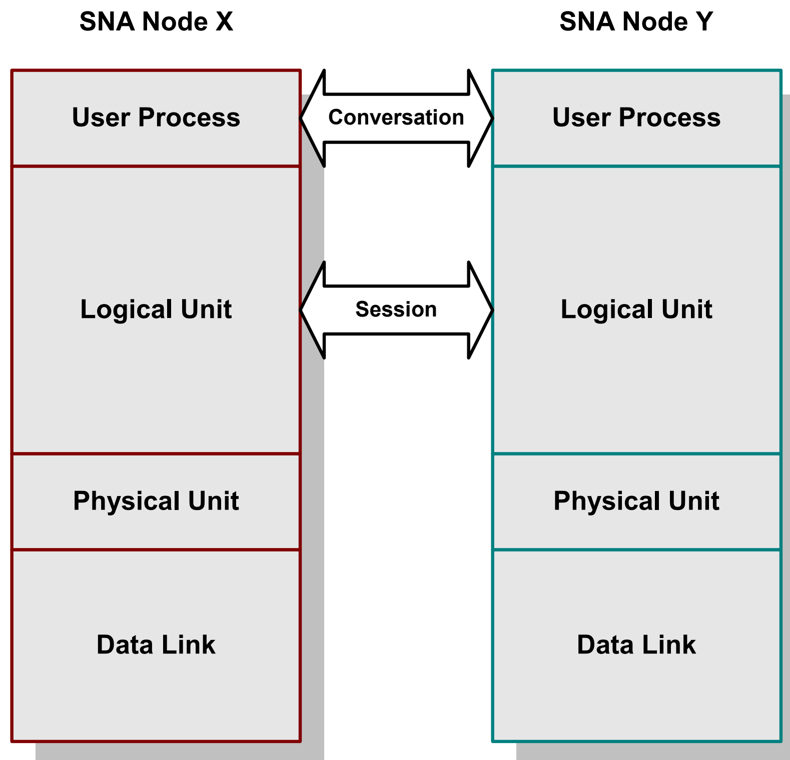
SNA uses a standard method for the exchange of data within a network. This standard method defines how to establish a route between components, how to send and receive data reliably, how to recover from errors, and how to prevent flow problems.

Originally designed for networks in which a mainframe computer controls the communications relationships, SNA has since evolved to incorporate protocols and implementations to allow two user processes to communicate with each other directly. These two different networking models, or roles, are referred to as hierarchical and peer-oriented, respectively. The peer-oriented model is designed to allow distributed control of the communications process independent of the mainframe.

The peer-to-peer connection between two user processes is known as a *conversation*, while the peer-to-peer connection between two LUs is known as a *session*. A session is

generally a long-term connection between two LUs, while a conversation is generally of shorter duration.

Figure 4 Sessions and Conversations



What is shown in Figure 2 and Figure 4 as a *User Process* is also known as a *Transaction Program* (TP). Also, the interface between a User Process and an LU is known as *Presentation Services*.

1.1.1 Supported Logical Unit Types

SNA LU6.2

LU 6.2, also known as APPC (Advanced Program-to-Program Communication), is used for Transaction Programs communicating with each other in a distributed data processing environment. In a CPIC (Common Programming Interface for Communications) implementation, CPIC provides the API that contains the commands, known as verbs, that are used by LU 6.2 to establish communication sessions.

Two types of Presentation Service interfaces are possible with LU6.2: mapped conversations and unmapped, or basic, conversations. Table 1 summarizes the set of LU6.2 commands for basic conversations. Equivalent commands for mapped conversations have the prefix <**MC_**> added to the command name. Note that “control operator verbs” are not listed.

Table 1 LU6.2 Commands

Name	Description
ALLOCATE	Allocates a conversation with another program.
CONFIRM	Sends a confirmation request to the remote process and waits for a reply.
CONFIRMED	Sends a confirmation reply to the remote process.
DEALLOCATE	De-allocates a conversation.
FLUSH	Forces the transmission of the local SEND buffer to the other LU.
GET_ATTRIBUTES	Obtains information about a conversation.
PREPARE_TO_RECEIVE	Changes the conversation state from SEND to RECEIVE.
RECEIVE_AND_WAIT	Waits for information (either data or confirmation request) to be received from the partner process.
RECEIVE_IMMEDIATE	Receives any information that is available in the local LU's buffer, but does not wait for information to arrive.
REQUEST_TO_SEND	Notifies the partner process that the local process wants to send data. When a "send" indication is received from the partner process, the conversation state changes.
SEND_DATA	Sends one data record to the partner process.
SEND_ERROR	Informs the partner process that the local process has detected an application error.

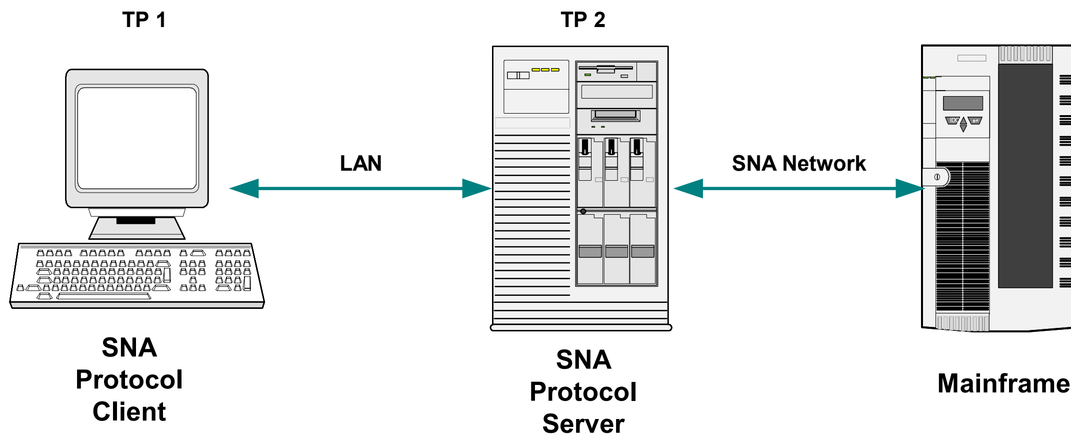
1.2 About the SNA eWay

The SNA eWay enables the SeeBeyond eGate Integrator system to access an SNA network environment to drive entire transactions, including conversational transactions.

The SNA eWay is an interface that makes calls to an SNA Server. The SNA Server acts as a high-speed gateway between distributed SNA Clients and the SNA network having a mainframe host system (see Figure 5).

In a typical data exchange using the SNA eWay, the eWay invokes the LU6.2 protocol--through the invocation of CPI-C calls--to enable the SNA client to send requests to the SNA server. For outbound eWays, the eWay can be triggered by any incoming message. For inbound eWays, the eWay is triggered by established conversation activity.

Figure 5 SNA Data Exchange



1.3 What's New in This Release

- "Collaborations" on page 12
- "Persistence" on page 12
- "Protocol State" on page 13

1.3.1 Collaborations

The SNA eWay exposes a common set of java methods equivalent to CPI-C v1.1 calls for all supported platforms. The eWay also provides a set of helper methods that group various CPI-C calls into one Java method so that a higher level function is achieved. These helper methods map directly to the methods that are present in version 4.5.x of the SNA eWay.

1.3.2 Persistence

Similar in nature to other transaction protocol eWays, the SNA eWay provides a persistence service that allows users to safely write any state information as a string to disk. The file names used are unique so as to enable multiple eWays to be run concurrently.

1.3.3 Conversation State

This attribute is set by the RA to the last known conversation state of the active conversation. Users must retrieve conversation state with a get method. There is no set method.

1.3.4 Protocol State

This attribute is set by the user and the RA upon conversation initiation or reset. It is a string that can be set or retrieved with the appropriate get or set method. The value of this variable persists between collaboration invocations; however, if a conversation is reset, lost or initiated, the protocol state is set to the initial value of the empty string.

1.4 About This Document

- **What's in This Document** on page 13
- **Scope** on page 13
- **Intended Audience** on page 14
- **Document Conventions** on page 14

1.4.1 What's in This Document

This document is organized topically as follows:

- **Chapter 1 "Introduction"** gives a general preview of this document, its purpose, scope, and organization.
- **Chapter 2 "Installing the eWay"** gives you an overview of the installation process.
- **Chapter 3 "Configuring the eWay"** describes the eWay properties and configurations settings available in the Connectivity Map and the Environment Properties.
- **Chapter 4 "SNA Java Collaborations"** decribes how to create basic and custom Java collaborations.
- **Chapter 5 "Implementing SNA eWay Projects"** explains how to import and run the supplied sample projects.

1.4.2 Scope

This user's guide describes the procedures necessary to install the SeeBeyond® Technology Corporation (SeeBeyond) SNA Intelligent Adapter eWay. In addition to the eWay's installation, the eWay's properties, collaborations, and samples are described in detail so that you may reference this book while creating your ICAN projects. Several ICAN projects, which use the SNALU62 eWay, are also provided and explained to enable you to quickly deploy your customized ICAN projects that use the SNA eWay Intelligent Adapter.

1.4.3 Intended Audience

The reader of this guide is presumed to be a developer or system administrator with responsibility for maintaining the SeeBeyond™ eGate™ Integrator system, and have a working knowledge of:

- Windows NT/2000 and/or UNIX operations and administration
- Windows-style GUI operations
- SNA Server, LU6.2, and CPIC APIs

Developers that choose to create projects with the exposed CPI-C Java methods provided by this eWay should be expert CPI-C programmers that possess extensive knowledge and understanding of CPI-C. To use either the exposed CPI-C Java methods or the helper Java methods to create your eWay collaborations, you should have a working knowledge and understanding of SNA LU6.2.

1.4.4 Document Conventions

The following conventions are observed throughout this document.

Table 2 Document Conventions

Text	Convention	Example
Names of buttons, files, icons, parameters, variables, methods, menus, and objects	**Bold** text	▪ Click **OK** to save and close. ▪ From the **File** menu, select **Exit**. ▪ Select the **logicalhost.exe** file. ▪ Enter the **timeout** value. ▪ Use the **getClassName()** method. ▪ Configure the **Inbound** File eWay.
Command line arguments, code samples	`Fixed` font. Variables are shown in ***bold italic***.	`bootstrap -p` ***`password`***
Hypertext links	**Blue** text	See **“Document Conventions” on page 14**
Hypertext links for Web addresses (URLs) or email addresses	**Blue underlined** text	**http://www.seebeyond.com** **docfeedback@seebeyond.com**

1.5 Related Documents

Many of the procedures included in this User’s Guide are described in greater detail in the *eGate Integrator User’s Guide*.

1.6 SeeBeyond Web Site

The SeeBeyond Web site is your best source for up-to-the-minute product news and technical support information. The site's URL is:

http://www.seebeyond.com

1.7 SeeBeyond Documentation Feedback

We appreciate your feedback. Please send any comments or suggestions regarding this document to:

docfeedback@seebeyond.com

Chapter 2

Installing the eWay

This chapter describes the requirements and procedures for installing the SNA eWay.

What's In This Chapter:

- **"Supported Operating Systems" on page 16**
- **"System Requirements" on page 16**
- **"Supported External Applications" on page 17**
- **"Installing the eWay Product Files" on page 17**
- **"After Installing the eWay" on page 19**

2.1 Supported Operating Systems

The SNA eWay is available on the following operating systems:

- Windows 2000, Windows XP, Windows Server 2003
- IBM AIX 5.1L and 5.2
- Sun Solaris 8 and 9
- Korean Windows 2000, Windows XP, Windows Server 2003
- Korean IBM AIX 5.1L and 5.2
- Korean Sun Solaris 8 and 9

2.2 System Requirements

The system requirements for the SNA eWay are the same as for eGate Integrator. For information, refer to the *SeeBeyond ICAN Suite Installation Guide*. Additional requirements that must be fullfilled before deploying an SNA eWay project are as follows:

- eGate Integrator, version 5.0.4 or higher
- File eWay (to support sample projects)
- Logical Host

- SNA network connection

2.3 Supported External Applications

The SNA eWay supports the following external systems:

- Win32 MS Host Integration Server
- AIX 32bit IBM eNetwork Communications Server v6.x or newer
- AIX 64bit IBM eNetwork Communications Server
- Solaris 64 bit SNAP/IX
- Alebra Brixton R4.1.3.6 or R5

Note: *Sunlink 9.x is not supported. Solaris platforms will need to be upgraded with the appropriate Alebra Brixton or SNAP/IX drivers.*

2.4 Installing the eWay Product Files

During the eGate Integrator installation process, the Enterprise Manager, a web-based application, is used to select and upload products as .sar files from the eGate installation CD-ROM to the Repository.

The installation process includes installing the following components:

- Uploading products to the Repository
- Downloading components
- Viewing product information home pages

Follow the instructions for installing the eGate Integrator in the *SeeBeyond ICAN Suite Installation Guide*, and include the following steps:

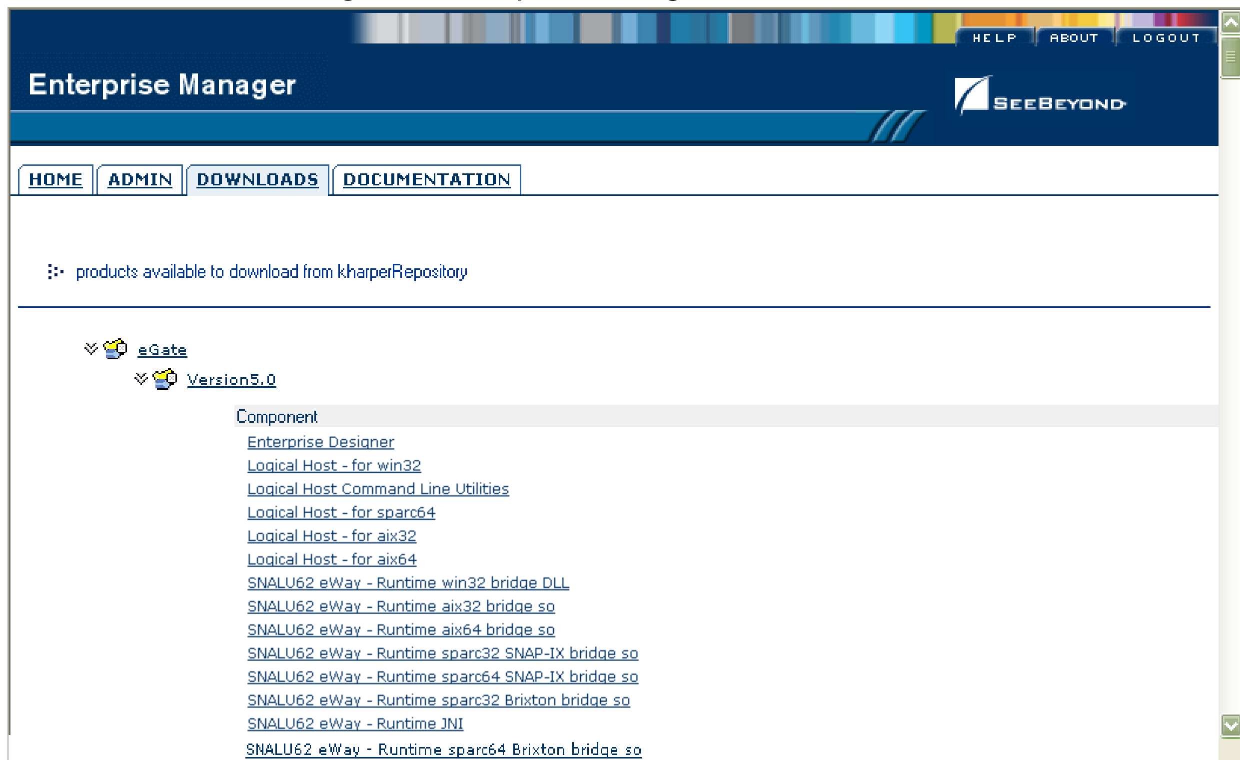
1. During the procedures for uploading files to the eGate Repository using the Enterprise Manager, after uploading the **eGate.sar** file, select and upload the following below as described in the *SeeBeyond ICAN Suite Installation Guide*:
    - **SNAeWay.sar** (to install the SNA eWay)
    - **FileeWay.sar** (to install the File eWay, used in the sample Projects)
    - **SNAeWayDocs.sar** (to install the user guide and the sample Projects)
2. From the Enterprise Manager, click the **DOCUMENTATION** tab.
3. Click **SNA eWay**.
4. In the right-hand pane, click **Download Sample**, and select a location for the .zip file to be saved.

For information about importing and using the sample, refer to **“Locating, Importing, and Using the Sample Projects” on page 46**.

5 Click on the Enterprise Manager’s **DOWNLOADS** tab. The Component list, as displayed in **Figure 6 on page 18**, includes 7 SNALU eWay components:

- **SNALU62 eWay - Runtime win32 bridge DLL** (stc_jnisna.dll): bridge shared libraries for Windows platforms.
- **SNALU62 eWay - Runtime aix32 bridge so**, **SNALU62 eWay - Runtime aix64 bridge so**, **SNALU62 eWay - Runtime sparc32 SNAP-IX bridge so**, **SNALU62 eWay - Runtime sparc64 SNAP-IX bridge so**, **SNALU62 eWay - Runtime sparc32 Brixton bridge so**(libstc_jnisna.so): bridge shared libraries for Unix platforms. Must be copied to the Integration Server Library path
- **SNALU62 eWay - Runtime JNI** (snalu62jni.jar): must be copied to the Integration Server classpath.

Figure 6 Enterprise Manager - DOWNLOADS



6 Download the JNI bridge file for your operating system.

For win32 JNI bridge files:

A Save the file to a directory that is declared in the system PATH statement (eg. C:\WINNT\System32).

For Unix bridge files:

B Save the appropriate bridge file to the local system.

C FTP the bridge file to your Unix directory.

- The target directory for AIX systems must be declared in LIBPATH.
- The target directory for Solaris must be declared in LD_LIBRARY_PATH.

7. Continue installing eGate Integrator according to the *eGate Integrator Installation Guide*.

2.5 After Installing the eWay

Once you have installed the SNA eWay, you must then incorporate it into an eGate Project and Environment in Enterprise Designer. The next chapters description how you add the eWay to an eGate Project and an eGate Environment.

Chapter 3

Configuring the eWay

This chapter explains how to configure the SNA eWay and environment properties.

What's In This Chapter:

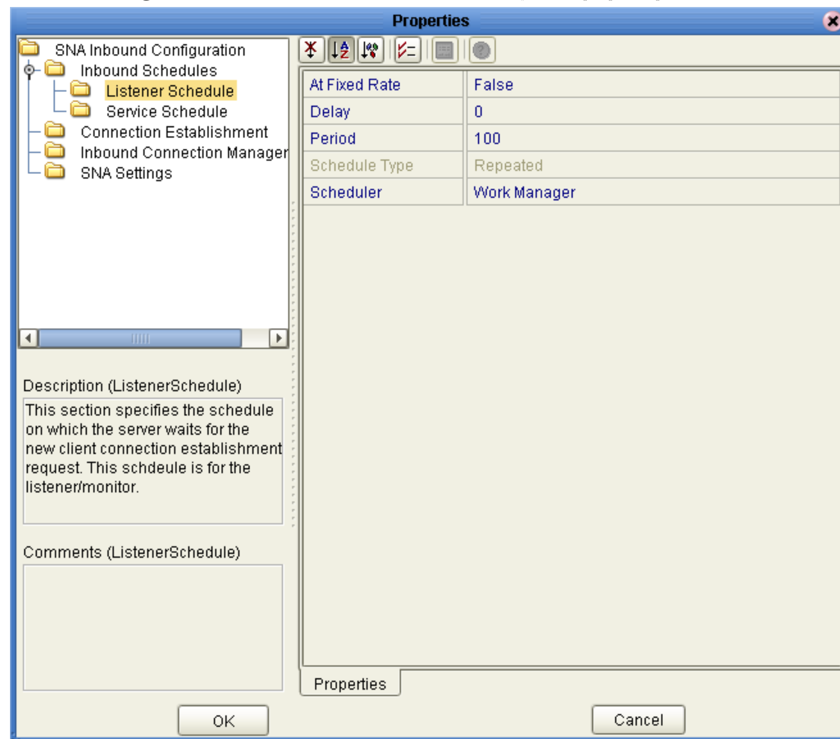
- **"Inbound Connectivity Map Properties" on page 20**
- **"Outbound Connectivity Map Properties" on page 27**
- **"Inbound Environment Properties" on page 30**
- **"Outbound Environment Properties" on page 32**
- **"Object Type Definitions (OTDs)" on page 34**

3.1 Inbound Connectivity Map Properties

This section describes in detail the inbound SNA eWay properties that are configured via the Connectivity Map:

- **"Inbound Schedules" on page 21**
- **"Connection Establishment" on page 24**
- **"Inbound Connection Manager" on page 24**
- **"SNA Settings" on page 25**
- **"General Settings" on page 26**

Figure 7 Inbound connectivity map properties



3.1.1 Inbound Schedules

This section describes the properties required to configure server inbound communication:

- **"Listner Schedule" on page 21**
- **"Service Schedule" on page 22**

Listner Schedule

Listner Schedule properties specifies the schedule upon which the server waits for the new client connection establishment request. This schdeule is for the listener/monitor. The listner schedule contains the following configuration properties:

- **"At Fixed Rate" on page 21**
- **"Delay" on page 22**
- **"Period" on page 22**
- **"Scheduler" on page 22**

At Fixed Rate

It is used for "Repeated" schedule type by the "Timer Service" scheduler. A true value means "Fixed-Rate"; a false value means "Fixed-Delay". In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch

up.". In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate). In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

- **True**
- **False** (Default)

Delay

Delay in milliseconds before task is to be executed. For details, please refer to the javadoc for java.util.Timer.

- Integer Value
- Default: 0

Period

It is used for "Repeated" schedule type. It specifies the regular interval in milliseconds between successive task executions. It should be a positive integer.

- Integer Value
- Default: 100

Scheduler

Specifies the scheduler type for this inbound communication. "Timer Service" - the task will be scheduled through the J2EE Timer Service; "Work Manager" - the work will be scheduled through the J2EE (JCA) Work Manager. If your container doesn't support JCA Work Management (prior to JCA1.5), you should choose "Timer Service".

- **Timer Service** - the task will be scheduled through the J2EE Timer Service
- **Work Manager** (Default) - the work will be scheduled through the J2EE Work Manager

Service Schedule

This section specifies the schedule upon which the server executes the business task (collaboration or business process) over the existing connection. This schedule is for the actual business rule that is defined by user. The service schedule contains the following configuration properties:

- **"At Fixed Rate" on page 23**
- **"Delay" on page 23**
- **"Period" on page 23**
- **"Schedule Type" on page 23**
- **"Scheduler" on page 23**

At Fixed Rate

It is used for "Repeated" schedule type by the "Timer Service" scheduler. A true value means "Fixed-Rate"; a false value means "Fixed-Delay". In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up.". In the long run, the frequency of execution will be exactly the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate). In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period (assuming the system clock underlying Object.wait(long) is accurate).

- **True**
- **False** (Default)

Delay

Delay in milliseconds before task is to be executed. For details, please refer to the javadoc for java.util.Timer.

- Integer Value
- Default: 0

Period

It is used for "Repeated" schedule type. It specifies the regular interval in milliseconds between successive task executions. It should be a positive integer.

- Integer Value
- Default: 100

Schedule Type

Schedule Type. It is used to define the type of schedule. "OneTime" - The task will be scheduled for one-time execution; "Repeated" - The task will be scheduled for repeated execution at regular intervals (which is defined by parameter "Period" in this section).

- **One Time** - The task will be scheduled for one-time execution
- **Repeated** (Default) - The task will be scheduled for repeated execution at regular intervals (which is defined by parameter "Period" in this section)

Scheduler

Specifies the scheduler type for this inbound communication. "Timer Service" - the task will be scheduled through the J2EE Timer Service; "Work Manager" - the work will be scheduled through the J2EE (JCA) Work Manager. If your container doesn't support JCA Work Management (prior to JCA1.5), you should choose "Timer Service".

- **Timer Service** - the task will be scheduled through the J2EE Timer Service
- **Work Manager** (Default) - the work will be scheduled through the J2EE Work Manager

3.1.2 Connection Establishment

Defines some configuration parameters used for controlling the connection establishment. This section describes the properties required to establish connections to SNA end-points:

- **"Max Connection Retry" on page 24**
- **"Retry Connection Interval" on page 24**

Max Connection Retry

The maximum number of times the eWay will attempt to connect to the specified external SNA LU62 destination before giving up.

- Integer Value
- Default: 3

Retry Connection Interval

The number of milliseconds to wait between attempts to connect to the specified external SNA LU62 destination.

- Integer Value
- Default: 30000

3.1.3 Inbound Connection Manager

Defines some configuration parameters used for inbound connection management, for example, the connection pool and the life cycle of the inbound connection. This section describes the properties required to accept inbound connections:

- **"Connection Pool Size" on page 24**
- **"Scope of Connection" on page 25**

Connection Pool Size

It defines the maximum number of the concurrent connections for the particular listener/monitor that is listening/monitoring over the specified SNA LU62 destination. It represents the capability/availability of this server service. Each connect-request from a client gains one concurrent connection.This parameter also represents the maximum number of clients that can concurrently connect to this server service and get served by the particular listener/monitor at the same time. 0 means no limit.

- Integer Value
- Default: 6

Scope of Connection

Scope Of Connection. It is used to define the scope of the accepted connection used by the eWay. "Resource Adapter Level" - Resource adapter will close the connection upon termination request, so that the connection may keep alive across multiple times's executions of the collaboration; "Collaboration Level" - The connection will be closed once the execution of the collaboration is done, so that the connection has the same life cycle as the collaboration.

- **Collaboration Level** - The connection will be closed once the execution of the collaboration is done, so the connection has the same life cycle as the collaboration
- **Resource Adaptor Level** (Default) - Resource adapter will close the connection upon closure request (via ClosureCommandMessage), so the connection may keep alive across multiple times's executions of the collaboration

3.1.4 SNA Settings

General SNA Settings. It represents general SNA configuration information. The following general configuration settings are described in this sections:

- **"Custom Handshake Class Name" on page 25**
- **"Deallocation Type" on page 25**
- **"Initialize Conversation" on page 26**
- **"Packet Size" on page 26**
- **"Synchronization Level" on page 26**
- **"Timeout" on page 26**

Custom Handshake Class Name

Custom Handshake Class Name. It will be used only when the users want to define their own SNA conversation handshake logic; otherwise, leave it blank. Once it is specified, it should be a fully qualified class name like 'com.abc.MyClass'. This class must implement interface com.stc.connector.snalu62.api.SNACustomerHandshake. See documentation for details.

Deallocation Type

Set this value to the type of deallocation required at the end of the conversation, when a shutdown is issued. Please refer to your SNA documentation for more information.

- **0 - CM_DEALLOCATE_SYNC_LEVEL** (Default)
- **1 - CM_DEALLOCATE_FLUSH**
- **2 - CM_DEALLOCATE_CONFIRM**
- **3 - CM_DEALLOCATE_ABEND**

Initialize Conversation

Specifies how the eWay will establish the SNA conversation. "true" - The eWay will initialize SNA conversations as an invoking TP; "false" - The eWay will accept SNA conversations as an invokable TP.Unless your use case requires this parameter to be modified, it is recommended that you keep the default setting, which is "false".

- **True**
- **False** (Default)

Packet Size

The number of bytes per packet of data. This number also determines the size of the buffers. It should be a positive integer.

- Integer Value
- Default: 1024

Synchronization Level

Set the synchronization level parameter (CM_SYNC_LEVEL). Please refer to your SNA manual for more information.

- **0 - None** (Default)
- **1 - Confirm**

Timeout

Used when making requests to the server, this is the number of milliseconds to wait for a response.

- Integer Value
- Default: 1000

3.1.5 General Settings

General Settings represent general control information for the eWay.

Scope of State

Scope Of State. It is used to define the scope of the State object, which is an OTD sub-node. The valid options for this parameter are:

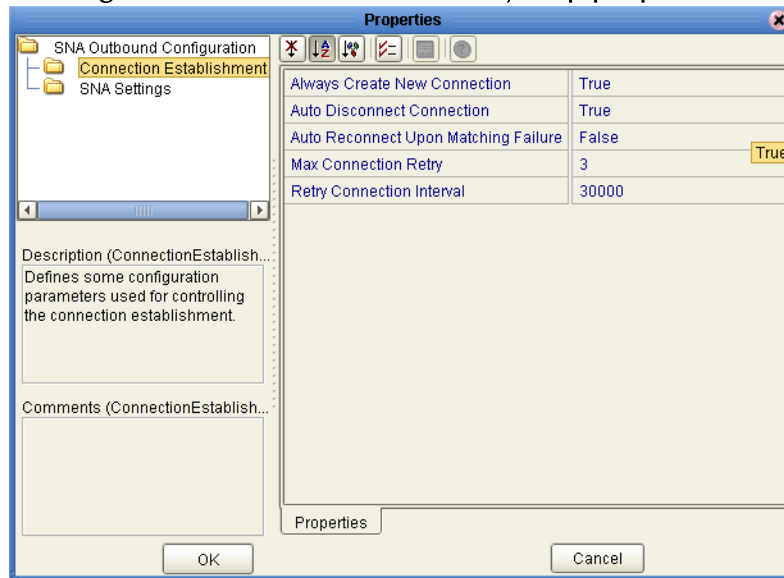
- **Resource Adapter Level** - the State has the same life cycle as the resource adapter;
- **Connection Level** (Default) - the State has the same life cycle as the connection;
- **OTD Level** - the State has the same life cycle as the OTD object. This scope represents the life cycle of the State.

3.2 Outbound Connectivity Map Properties

This section describes in detail the outbound SNA eWay properties that are configured via the Connectivity Map:

- "Connection Establishment" on page 27
- "SNA Settings" on page 29
- "General Settings" on page 30

Figure 8 Outbound connectivity map properties



3.2.1 Connection Establishment

Defines some configuration parameters used for controlling the connection establishment. This section describes in detail the configuration parameters that control connections:

- "Always Create New Connection" on page 27
- "Auto Disconnect Connection" on page 28
- "Auto Reconnect Upon Matching Failure" on page 28
- "Max Connection Retry" on page 28
- "Retry Connection Interval" on page 28

Always Create New Connection

This flag indicates whether to ALWAYS try to create a new connection for a connection establishment request. If it is false, an attempt to match an existing connection

(managed by container) is made; if it is true, a new connection is always created without trying to match an existing connection.

- **True** - always try to create a new connection without trying to match connection.
- **False** (Default)- try to match an existing connection (managed by container).

Auto Disconnect Connection

This flag indicates whether the eWay disconnects the connection automatically after the work upon the connection finishes. If it is false, the connection will be left for reuse; if it is true, the connection will be disconnected and it won't be re-used.

- **True** - the connection will be disconnected and it won't be re-used.
- **False** (Default)- the connection will be left for reuse.

Auto Reconnect Upon Matching Failure

Specifies whether or not to make an attempt to re-connect automatically--after getting a matched connection from a container. This this connection is not valid due to different reasons. For example, the external side of the connection was closed/reset upon the external application's logic. A value of 'true' means that we will discard the invalid matched connection and try to re-establish a new connection automatically. A value of 'false' means that we won't try to re-establish a new connection automatically. Instead, we will simply leave/defer the control to users' business rules, which should detect this kind of failure and perform the desired operations.

- **True** (Default)- discards the invalid matched connection and tries to re-establish a new connection automatically.
- **False** - the eWay won't try to re-establish a new connection automatically; instead, connection control is deferred to users' business rules. User must detect this kind of failure and develop the business logic to perform accordingly.

Max Connection Retry

The maximum number of times the eWay will attempt to connect to the specified external SNA LU62 destination before giving up.

- Integer Value
- Default: 3

Retry Connection Interval

The number of milli-seconds to wait between attempts to connect to the specified external SNA LU62 destination.

- Integer Value
- Default: 30000

3.2.2 SNA Settings

General SNA Settings. It represents general SNA configuration information. For more details, you may need to refer to your SNA documentation, the eWay documentation and the javadocs. This section describes in detail the general SNA configuration parameters:

- **"Custom Handshake Class Name" on page 29**
- **"Deallocation Type" on page 29**
- **"Initialize Conversation" on page 29**
- **"Packet Size" on page 29**
- **"Synchronization Level" on page 30**
- **"Timeout" on page 30**

Custom Handshake Class Name

Custom Handshake Class Name. It will be used only when the users want to define their own SNA conversation handshake logic; otherwise, leave it blank. Once it is specified, it should be a fully qualified class name like 'com.abc.MyClass'. This class must implement interface com.stc.connector.snalu62.api.SNACustomerHandshake. See documentation for details.

Deallocation Type

Set this value to the type of deallocation required at the end of the conversation, when a shutdown is issued. Please refer to your SNA documentation for more information.

- 0 - CM_DEALLOCATE_SYNC_LEVEL (Default)
- 1 - CM_DEALLOCATE_FLUSH
- 2 - CM_DEALLOCATE_CONFIRM
- 3 - CM_DEALLOCATE_ABEND

Initialize Conversation

Specifies how the eWay will establish the SNA conversation. "true" - The eWay will initialize SNA conversations as an invoking TP; "false" - The eWay will accept SNA conversations as an invokable TP.Unless your use case requires this parameter to be modified, it is recommended that you keep the default setting, which is "true".

- **True** (Default)
- **False**

Packet Size

The number of bytes per packet of data. This number also determines the size of the buffers. It should be a positive integer.

- Integer Value
- Default: 1024

Synchronization Level

Set the synchronization level parameter (CM_SYNC_LEVEL). Please refer to your SNA manual for more information.

- **0 - None** (Default)
- **1 - Confirm**

Timeout

Used when making requests to the server, this is the number of milliseconds to wait for a response.

- Integer Value
- Default: 1000

3.2.3 General Settings

General Settings represent general control information for the eWay.

Scope of State

Scope Of State. It is used to define the scope of the State object, which is an OTD sub-node. The valid options for this parameter are:

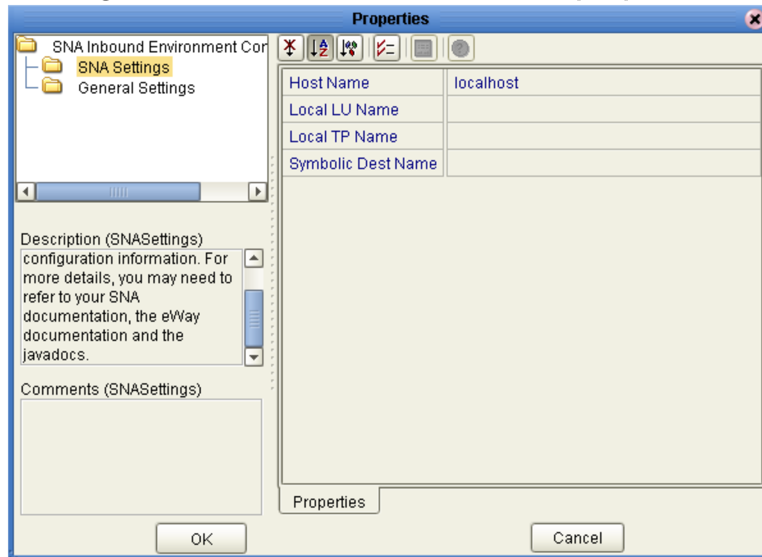
- **Resource Adapter Level** - the State has the same life cycle as the resource adapter;
- **Connection Level** (Default) - the State has the same life cycle as the connection;
- **OTD Level** - the State has the same life cycle as the OTD object. This scope represents the life cycle of the State.

3.3 Inbound Environment Properties

This section describes in detail the inbound SNA environment properties that are configured via the Environment Explorer:

- **"SNA Settings" on page 31**
- **"General Settings" on page 32**

Figure 9 Inbound SNA environment properties



3.3.1 SNA Settings

General SNA Settings. It represents general SNA configuration information. For more details, you may need to refer to your SNA documentation, the eWay documentation and the javadocs. The following general SNA configuration parameters are configured from within the SNA environment properties:

- **"Host Name" on page 31**
- **"Local LU Name" on page 31**
- **"Local TP Name" on page 31**
- **"Symbolic Destination Name" on page 32**

Host Name

The Host Name where the lu68 Server runs, only needed for Brixton LU62 server.

- Default: localhost

Local LU Name

The local LU name defined to the SunLink LU6.2 server. This parameter is required for Sunlink P2P LU6.2 9.1 and is ignored on other platforms. This name is case-sensitive. Please refer to your SNA documentation for more information.

Local TP Name

The local Transaction Program, TP, name that is running on the local Logical Unit (LU). It is case-sensitive. Please refer to your SNA documentation for more information.

Symbolic Destination Name

The symbolic destination name associated with a side information entry loaded from the configuration file. This name is case-sensitive. Please refer to your SNA documentation for more information.

3.3.2 General Settings

General Settings represent general control information for the logical host environment.

Persistent Storage Location

Specifies the Persistent Storage Location (a local folder name). The folder name that contains the file used to store the persistent data. The base file name will be generated according to project/deployment/collaboration information.

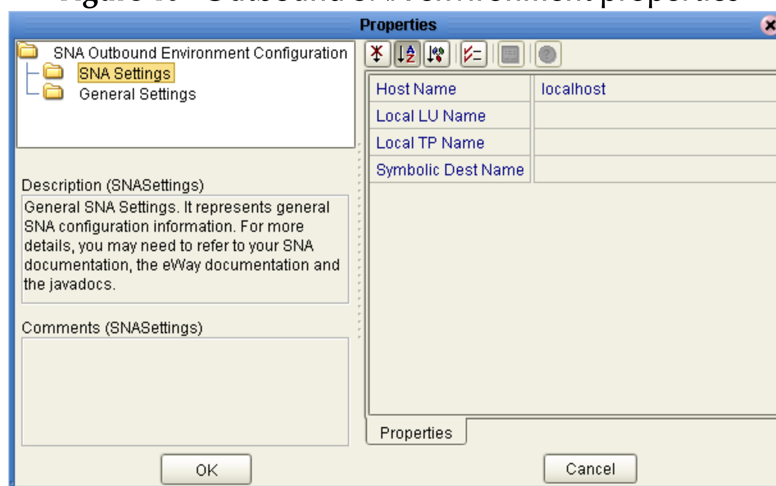
- Default: *C:/temp/snalu62inbound/persist*

3.4 Outbound Environment Properties

This section describes in detail the inbound SNA environment properties that are configured via the Environment Explorer:

- **"SNA Settings" on page 33**
- **"General Settings" on page 33**

Figure 10 Outbound SNA environment properties



3.4.1 SNA Settings

General SNA Settings represent SNA configuration parameters required to communicate with entities via SNA. The following general SNA configuration parameters are configured from within the SNA environment properties:

- **"Host Name" on page 33**
- **"Local LU Name" on page 33**
- **"Local TP Name" on page 33**
- **"Symbolic Destination Name" on page 33**

Host Name

The Host Name where the lu68 Server runs, only needed for Brixton LU62 server.

- Default: localhost

Local LU Name

The local LU name defined to the SunLink LU6.2 server. This parameter is required for Sunlink P2P LU6.2 9.1 and is ignored on other platforms. This name is case-sensitive. Please refer to your SNA documentation for more information.

Local TP Name

The local Transaction Program, TP, name that is running on the local Logical Unit (LU). It is case-sensitive. Please refer to your SNA documentation for more information.

Symbolic Destination Name

The symbolic destination name associated with a side information entry loaded from the configuration file. This name is case-sensitive. Please refer to your SNA documentation for more information.

3.4.2 General Settings

General Settings represent general control information for the logical host environment.

Persistent Storage Location

Specifies the Persistent Storage Location (a local folder name). The folder name that contains the file used to store the persistent data. The base file name will be generated according to project/deployment/collaboration information.

Default: *C:/temp/snalu62outbound/persist*

3.5 Object Type Definitions (OTDs)

Unlike most other eWays, the SNA eWay does not consist of an OTD wizard. OTD wizards typically facilitate the creation of a collaborations that are used with eWay projects. When an OTD wizard is available, a skeleton collaboration is created to provide minimal funtionality that you must modify to suit your application's needs. Without the OTD wizard, as in the case of the SNA eWay, you must create your collaborations completely from scratch.

To associate the standard SNA eWay OTD to a new Java collaboration:

1. From the Project Explorer, right-click the targeted project.
2. Select **New > Collaboration Definition (Java)...**
3. Complete steps 1 and 2 of the **Collaboration Definition Wizard (Java)**.
4. Select the OTD to use in the new collaboration by traversing the **Look In** drop-down box: SeeBeyond.eWays.SNALU62.
5. Highlight the desired OTD name and click the **Add** button.
6. Optionally, modify the instance name of the OTD that will be used in the collaboration.
7. Click the **Finish** button.

The new collaboration that impliments the SNA eWay OTD is created. For details about the SNA eWay methods that may be used with collaborations for the, refer to **"SNA eWay Javadocs" on page 69**.

Chapter 4

SNA Java Collaborations

This chapter provides an overview of the default SNA collaboration that is created with new Java collaborations that are associated with the standard SNA OTD. This chapter also discusses how to create custom collaborations that can be used with the project samples and recommended practices for developing new collaborations.

What's in This Chapter

- **Creating Default Java Collaborations** on page 35
- **Creating Custom Collaborations** on page 39
- **Best Practices** on page 50

4.1 Creating Default Java Collaborations

In order for your project to use the functionality available in the SNA eWay, you must have a collaboration that implements the SNA OTD. This section describes how to create a default SNA Java collaboration using the **Collaboration Definition Wizard (Java)**.

To create a default Java collaboration

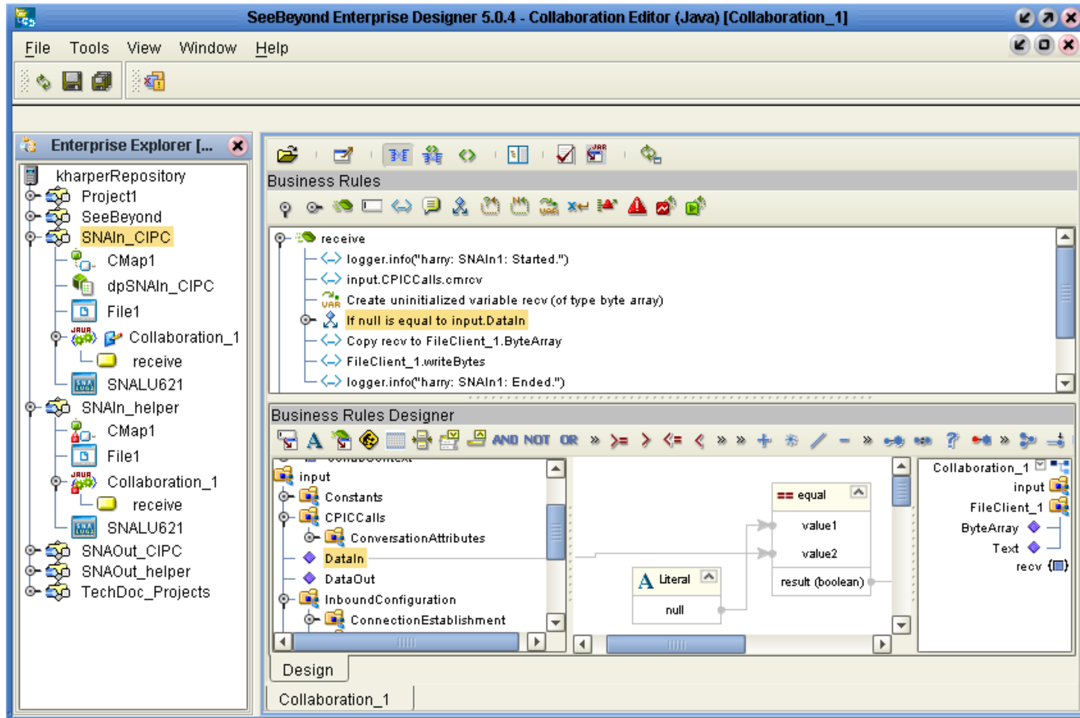
1. In the **Project Explorer** tab of the Enterprise Designer, right-click the project, select **New** > **Collaboration Definition (Java)**.
2. Complete steps 1 and 2 of the **Collaboration Definition Wizard (Java)**. For details about this wizard, refer to the *eGate Integrator User's Guide*.
3. Select the SNA OTD to use in the new collaboration by traversing the **Look In** drop-down box: SeeBeyond.eWays.SNA.
4. Highlight the desired OTD name and click the **Add** button.
5. Optionally, modify the instance name of the OTD that will be used in the collaboration.
6. Click the **Finish** button.

A new collaboration associated with the SNA OTD is placed into your targeted project.

7. In the **Collaboration Editor** window, create the source code and the data mappings for the collaboration. For details about the **Collaboration Editor**, refer to the *eGate Integrator User's Guide*. For information about SNA methods, refer to **"SNA eWay Javadocs" on page 69**.

The figure below shows an example of data mapping for an inbound SNA collaboration. To explore the business logic design for an actual project, import the SNA sample projects as described in **“Importing Projects” on page 55**.

Figure 11 Inbound collaboration (SNAIn_CPIC)



4.1.1 Analyzing the Default Java Collaboration

Newly created Java collaborations that implement the SNA OTD consist of skeleton SNA functionality required to use the collaboration with your project. The collaboration code generated upon SNA Java collaboration creation is provided in Figure 12.

Figure 12 Default collaboration code listing (complete)

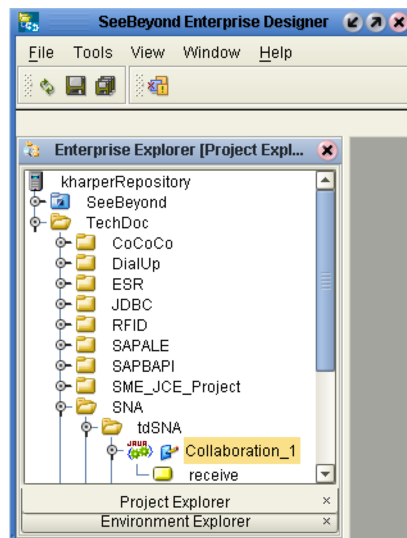
```
package TechDocSNAtdSNA;
public class Collaboration_1
{
   public void receive(
      com.stc.connector.snalu62.inbound.SNAInboundApplication input,
      com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1
   )
      throws Throwable
         {
               //code goes here.
         }

   public com.stc.codegen.logger.Logger logger;
   public com.stc.codegen.alerter.Alerter alerter;
   public com.stc.codegen.util.CollaborationContext collabContext;

}
```

The first line of code declares the package to which the collaboration belongs. The default package name is always the concatenation of the Project Explorer tree structure for a particular repository. For this example the tree structure is shown in figure 22. The collaboration belongs to the TechDoc.SNA.tdSNA project. Therefore, when the collaboration wizard creates the new collaboration, the package name is: `TechDocSNAtdSNA`.

Figure 13 Example project tree structure.

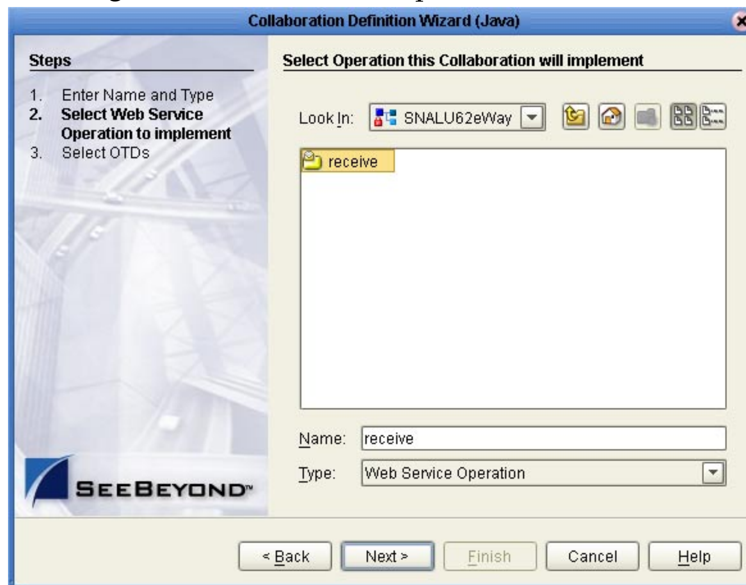


After the package name is declared, the collaboration object itself is defined. In order for the class object to be available to other collaborations and more complex projects, the collaboration class is defined as a `public class`. The name of the collaboration object class was specified on the first page of the **Collaboration Definition Wizard (Java)**. This example used the default name provided by the wizard. If you find it necessary to alter the name of the collaboration object defined in the code, you must perform two steps. Firstly, you must rename the object specified by the `public class`

declaration. Secondly, you must rename the collaboration file listed in the project tree structure to match the name of the defined object. Since this example uses Collaboration_1 as the name of the collaboration in the Project Explorer, the defined collaboration object becomes: `public class Collaboration_1`

After the collaboration object and package name declarations, the web service operation--specified in the Collaboration Definition Wizard (see Figure 14)--that the collaboration will perform is specified.

Figure 14 Web Service Operation declaration



Since this example only uses the receive web service operation, only the receive constructor is generated. In order for this collaboration to be available to other collaborations and projects, it too must be declared as a `public` method.

```
public void receive(
   com.stc.connector.snalu62.inbound.SNAInboundApplication input,
   com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1
)
   throws Throwable {}
```

The `receive` method does not return any data to the collaboration and, as such, is assigned a `void` data type. The receive method provides two interfaces that may be used in your collaboration. The `SNAInboundApplication` interface is used to handle all incoming conversations. The `SNAOutboundApplication` interface is used to handle outbound conversations. If this collaboration were configured to communicate with other eWays, you would declare those interfaces here also. For example, if you also needed the collaboration to send data to a file, you could use the File eWay and declare it as follows:

```
public void receive(
   com.stc.connector.snalu62.inbound.SNAInboundApplication input,
   com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1
   com.stc.connector.appconn.file.FileApplication FileClient_1
)
   throws Throwable {}
```

Minimal error handling for the collaboration is provided by the `Throwable {}` class. It is recommended that you implement a more robust exception handling mechanism for your project to assist with any potential issues that may arise

Additional eWay development utilities are also provided in your generated collaboration. You should consult the eGate Integrator User's Guide for details about these declared methods.

```
public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
```

4.2 Creating Custom Collaborations

This section describes how to create the collaborations provided in the sample projects. Each of the 4 collaborations provide different techniques for using the eWay to perform varying ranges of SNA conversation tasks that you may need to execute. The collaborations are not meant to demonstrate the only way to perform desired operations; rather, they should provide insight into how you may use the SNA eWay to develop your applications.

The Java collaborations you create can be grouped into one of three categories, each of which is discussed in this chapter.

About the Collaborations

- **Inbound SNA Conversations** on page 39
- **Outbound SNA Conversations** on page 45
- **Inbound and Outbound SNA Conversations** on page 50

4.2.1 Inbound SNA Conversations

Inbound SNA conversations accept incoming conversation requests from remote transaction programs that initialize conversations. This section describes how to create

collaborations that accept conversations and output the conversation to a file to be stored on the local system. An explanation of how to create the collaboration with both the CPIC methods and the Helper methods is provided.

- **Conversation Sent to a Text File (CPIC)** on page 40
- **Conversation Sent to a Text File (Helper)** on page 42

Conversation Sent to a Text File (CPIC)

This collaboration demonstrates how to use the eWay, in conjunction with the CPIC Java methods, to accept an incoming SNA conversation and output the conversation to a file on the local system.

Figure 15 SNAIn_CPIC collaboration code

```
package SNAIn1;
public class Collaboration_1
{
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public void receive(
    com.stc.connector.snalu62.inbound.SNAInboundApplication input,
    com.stc.connector.appconn.file.FileApplication FileClient_1
    )
        throws Throwable
    {
        // @map:logger.info("SNAIn1: Started.")
        logger.info( "SNAIn1: Started." );
        // @map:CPICCalls.cmrcv
        input.getCPICCalls().cmrcv();
        // @map:byte[] recv;
        byte[] recv;
        // @map:
        if (null == input.getDataIn()) {
            // @map:Copy Bytes to recv
            recv = "No data is received.".getBytes();
        } else {
            // @map:Copy DataIn to recv
            recv = input.getDataIn();
        }
        // @map:Copy recv to ByteArray
        FileClient_1.setByteArray( recv );
        // @map:FileClient_1.writeBytes
        FileClient_1.writeBytes();
        // @map:logger.info("SNAIn1: Ended.")
        logger.info( "SNAIn1: Ended." );
    }

}
```

The first six lines of code in this collaboration are created by the default collaboration wizard. As such, these line of code will not discuss these here. However, keep in mind that if you are creating a new Java collaboration, your package name and class name may differ. The next three lines of code tell the collaboration from where the data should be retrieved and to where the data should be sent.

```
    public void receive(
      com.stc.connector.snalu62.inbound.SNAInboundApplication input,
      com.stc.connector.appconn.file.FileApplication FileClient_1
     )
      throws Throwable
         {
```

Since the goal of this collaboration is to handle incoming conversations, you need to create an instance of the `SNAInboundApplication` interface in order to read incoming conversation traffic. In order for the collaboration to know to where the data should be sent, you must create an instance of the `FileApplication` interface that belongs to the File eWay. This will allow you to output incoming conversation traffic to a file on the local system. Error handling is handled by the `Throwable` class.

```
        // @map:logger.info("SNAIn1: Started.")
        logger.info( "SNAIn1: Started." );
        // @map:CPICCalls.cmrcv
        input.getCPICCalls().cmrcv();
        // @map:byte[] recv;
        byte[] recv;
```

One of the first things you should do before processing any conversation traffic is turn on the logging feature. Here, the logger is instantiated with a specific phrase to include in the log file. Now that limited debugging for the collaboration is available, the next step is to tell the collaboration to listen to the incoming traffic. Listening to the conversation traffic is performed by using the CPIC `cmrcv()` method of the exposed Java CPIC calls. Since the SNA CPIC calls belong to `getCPICCalls()`, listening to the input from the looks like this: `input.getCPICCalls().cmrcv()`. In order to process the incoming traffic, a byte array, `recv`, is created.

```
        // @map:
        if (null == input.getDataIn()) {
            // @map:Copy Bytes to recv
            recv = "No data is received.".getBytes();
        } else {
            // @map:Copy DataIn to recv
            recv = input.getDataIn();
        }
```

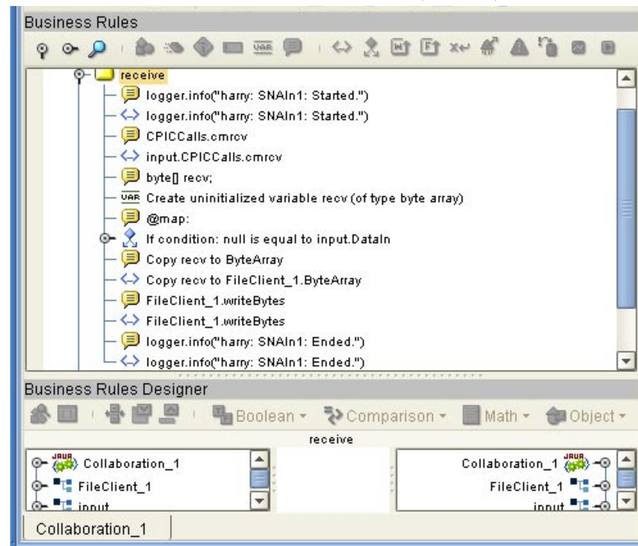
Depending on the incoming conversation traffic, a logic loop is setup to tell the collaboration what to do with the incoming byte data. If no data is received (`if (null == input.getDataIn())`), an output message is displayed, `recv = "No data is received.".getBytes()`, before continuing to listen for any change in the data stream. When an incoming data array is finally detected, the loop captures the data, `recv = input.getDataIn()`, and sends it to the `FileApplication` interface for further processing.

```
// @map:Copy recv to ByteArray
FileClient_1.setByteArray( recv );
// @map:FileClient_1.writeBytes
FileClient_1.writeBytes();
// @map:logger.info("SNAIn1: Ended.")
logger.info( "SNAIn1: Ended." );
```

Since the goal is to output the collected data array to a file, the File eWay takes the input data, as a byte array (`FileClient_1.setByteArray( recv )`), and writes the data (`FileClient_1.writeBytes()`) to the file specified by the eWay properties. After the data array from the incoming conversation is collected and output to a file, a final log message is displayed, `logger.info( "SNAIn1: Ended." )`

If you are using the business rule designer to create your collaboration, see Figure 16 for the business rules associated with this collaboration.

Figure 16 Inbound collaboration (CPIC) business rules



Conversation Sent to a Text File (Helper)

This collaboration demonstrates how to use the eWay to accept an incoming SNA conversation and send the conversation to a text file located on the local system. To begin with, you should create a new Java collaboration with the **Collaboration Definition Wizard**. Use the wizard to specify the SNA `receive` web service operation and 2 external FileClient applications.

Figure 17 `SNAIn_helper collaboration code`

```
package SNAIn1;
public class Collaboration_1
{
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public void receive(
    com.stc.connector.snalu62.inbound.SNAInboundApplication input,
    com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        // @map:logger.info("SNAIn1: Started.")
        logger.info( "SNAIn1: Started." );
        input.recv();
        byte[] recv;
        // @map:
        if (null == input.getDataIn()) {
            // @map:Copy Bytes to recv
            recv = "No data is received.".getBytes();
        } else {
            // @map:Copy DataIn to recv
            recv = input.getDataIn();
        }
        // @map:Copy recv to ByteArray
        FileClient_1.setByteArray( recv );
        // @map:FileClient_1.writeBytes
        FileClient_1.writeBytes();
        // @map:logger.info("SNAIn1: Ended.")
        logger.info( "SNAIn1: Ended." );
    }
}
```

The first six lines of code in this collaboration are created by the default collaboration wizard. As such, these lines of code will not discuss these here. However, keep in mind that if you are creating a new Java collaboration, your package name and class name may differ. The next three lines of code tell the collaboration from where to get the data and to where the data should be sent.

```
    public void receive(
     com.stc.connector.snalu62.inbound.SNAInboundApplication input,
     com.stc.connector.appconn.file.FileApplication FileClient_1
    )
     throws Throwable
        {
```

Since the goal of this collaboration is to handle incoming conversations, you need to create an instance of the `SNAInboundApplication` interface in order to read incoming conversation traffic. In order for the collaboration to know to where the data should be sent, you must create an instance of the `FileApplication` interface that belongs to the File eWay. This will allow you to output incoming conversation traffic to a file on the local system. Error handling is handled by the `Throwable` class.

```
// @map:logger.info("SNAIn1: Started.")
logger.info( "SNAIn1: Started." );
input.recv();
// @map:byte[] recv;
byte[] recv;
```

One of the first things you should do before processing any conversation traffic is turn on the logging feature. Here, the logger is instantiated with a specific phrase to include in the log file. Now that limited debugging for the collaboration is available, the next step is to tell the collaboration to listen to the incoming traffic. Listening to the conversation traffic is performed by using the Helper `recv()` method (refer to **"SNA eWay Javadocs" on page 69**). In order to process the incoming traffic, a byte array, `recv`, is created.

```
// @map:
if (null == input.getDataIn()) {
    // @map:Copy Bytes to recv
    recv = "No data is received.".getBytes();
} else {
    // @map:Copy DataIn to recv
    recv = input.getDataIn();
}
```

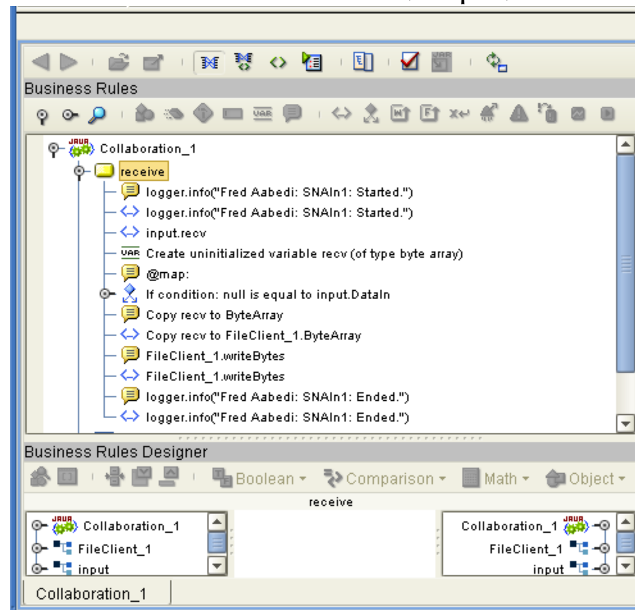
Depending on the incoming conversation traffic, a logic loop is setup to tell the collaboration what to do with the incoming byte data. If no data is received (`if (null == input.getDataIn())`), an output message is displayed, `recv = "No data is received.".getBytes()`, before continuing to listen for any change in the data stream. When an incoming data array is finally detected, the loop captures the data, `recv = input.getDataIn()`, and sends it to the `FileApplication` interface for further processing.

```
// @map:Copy recv to ByteArray
FileClient_1.setByteArray( recv );
// @map:FileClient_1.writeBytes
FileClient_1.writeBytes();
// @map:logger.info("SNAIn1: Ended.")
logger.info( "SNAIn1: Ended." );
```

Since the goal is to output the collected data array to a file, the File eWay takes the input data, as a byte array (`FileClient_1.setByteArray( recv )`), and writes the data (`FileClient_1.writeBytes()`) to the file specified by the eWay properties. After the data array from the incoming conversation is collected and output to a file, a final log message is displayed, `logger.info( "SNAIn1: Ended." )`

If you are using the business rule designer to create your collaboration, see Figure 18 for the business rules associated with this collaboration.

Figure 18 Inbound collaboration (Helper) business rules



4.2.2 Outbound SNA Conversations

Outbound SNA conversations initialize conversations and relay (send) data to transaction programs that accept the initialize conversation request. This section describes how to create collaborations that route conversation data to an SNA point. The conversation data (from this collaboration) sent originates in a text file located on the local system. An explanation of how to create the collaboration with both the CPIC methods and the Helper methods is provided.

- **Conversation Originates from a Text File (CPIC)** on page 45
- **Conversation Originates from a Text File (Helper)** on page 48

Conversation Originates from a Text File (CPIC)

This collaboration demonstrates how to use the eWay, in conjunction with the CPIC Java methods, to read a text message stored in a text file and send that message as a conversation.

Figure 19 SNAOut_CPIC collaboration code

```
package SNAOut1;
public class Collaboration_1
{
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public void receive(
    com.stc.connector.appconn.file.FileTextMessage input,
    com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1,
    com.stc.connector.appconn.file.FileApplication FileClient_1
    )
        throws Throwable
    {
        // @map:logger.info("SNAOut1: Started.")
        logger.info( "SNAOut1: Started." );
        // @map:Copy ByteArray to DataOut
        SNALU62eWay_1.setDataOut( input.getByteArray() );
        // @map:CPICCalls.cmsend
        SNALU62eWay_1.getCPICCalls().cmsend();
        // @map:CPICCalls.cmflus
        SNALU62eWay_1.getCPICCalls().cmflus();
        // @map:logger.info("SNAOut1: Ended.")
        logger.info( "SNAOut1: Ended." );
    }
}
```

The first six lines of code in this collaboration are created by the default collaboration wizard. As such, these lines of code will not discuss these here. However, keep in mind that if you are creating a new Java collaboration, your package name and class name may differ. The next three lines of code tell the collaboration from where the data should be retrieved and to where the data should be sent.

```
   public void receive(
   com.stc.connector.appconn.file.FileTextMessage input,
   com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1,
   com.stc.connector.appconn.file.FileApplication FileClient_1
)
     throws Throwable
        {
```

Since the goal of this collaboration is to retrieve data from a file and send the data, you need to create an instance of the `SNAOutboundApplication` interface in order to deliver conversation traffic. In order for the collaboration to know from where to retrieve the data that is to be sent via the `SNAOutboundApplication` interface, you must create an instance of the `FileApplication` interface that belongs to the File eWay. This will allow you to read the data file on the local system. Error handling is handled by the `Throwable` class.

```
// @map:logger.info("SNAOut1: Started.")
logger.info( "SNAOut1: Started." );
// @map:Copy ByteArray to DataOut
SNALU62eWay_1.setDataOut( input.getByteArray() );
```

One of the first things you should do before processing any conversation traffic is turn on the logging feature. Here, the logger is instantiated with a specific phrase to include in the log file. Now that limited debugging for the collaboration is available, the next step is to tell the collaboration from where to retrieve the data that will be transmitted. Using the getByteArray method available for the File eWay, the data can be read with this: `input.getByteArray()`.

A moderately simple construct of the `SNAOutboundApplication` interface can use this getByteArray method to load the data into the send buffer before it is transmitted. In order to load the data to the outbound buffer, you can use something similar to this: `SNALU62eWay_1.setDataOut( input.getByteArray())`. This sets the content of the outgoing payload buffer to the data found in the file.

```
// @map:CPICCalls.cmsend
SNALU62eWay_1.getCPICCalls().cmsend();
// @map:CPICCalls.cmflus
SNALU62eWay_1.getCPICCalls().cmflus();
```

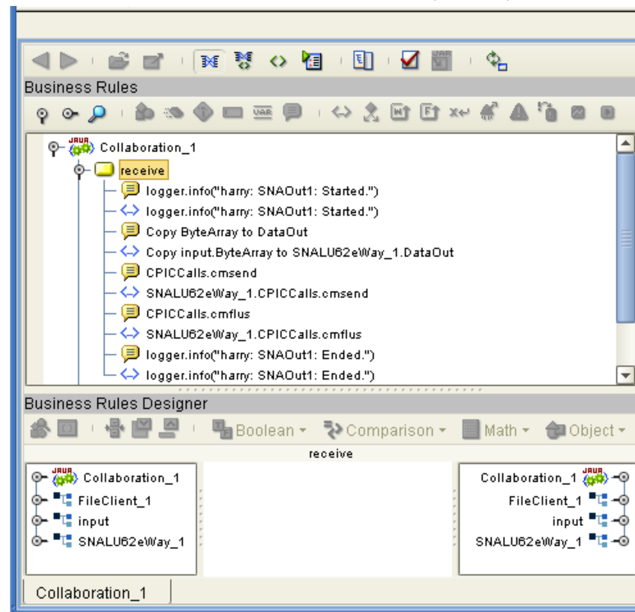
After the payload buffer is filled with the contents of the conversation message to be delivered, you can then send the message. Sending a message to can be performed by using the CPIC `cmsend()` method of the exposed Java CPIC calls. Since the SNA CPIC calls belong to `getCPICCalls()`, listening to the input looks like this: `input.getCPICCalls().cmsend()`. After you initiate the send message activity, you must to flush the contents of the payload buffer to ensure that all the data was sent and to clean the buffer so that it can be used again, if the need arises. To flush the payload buffer, you can use: `SNALU62eWay_1.getCPICCalls().cmflus()`.

```
// @map:logger.info("SNAIn1: Ended.")
logger.info( "SNAIn1: Ended." );
```

After the conversation traffic has been sent, a final log message is displayed, `logger.info( "SNAIn1: Ended." )`.

If you are using the business rule designer to create your collaboration, see Figure 20 for the business rules associated with this collaboration.

Figure 20 Outbound collaboration (CPIC) business rules



Conversation Originates from a Text File (Helper)

This collaboration demonstrates how to use the eWay to read a message from a local file and send that message data. To begin with, you should create a new Java collaboration with the **Collaboration Definition Wizard**. Use the wizard to specify the SNA `receive` web service operation and 2 external FileClient applications.

Figure 21 `SNAOut_helper collaboration code`

```
package SNAOut1;
public class Collaboration_1
{
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public void receive(
    com.stc.connector.appconn.file.FileTextMessage input,
   com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1,
    com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        // @map:logger.info("SNAOut1: Started.")
        logger.info( "SNAOut1: Started." );
        // @map:Copy ByteArray to DataOut
        SNALU62eWay_1.setDataOut( input.getByteArray() );
        SNALU62eWay_1.send();
        logger.info( "SNAOut1: Ended." );
    }
}
```

The first six lines of code in this collaboration are created by the default collaboration wizard. As such, these lines of code will not discuss these here. However, keep in mind

that if you are creating a new Java collaboration, your package name and class name may differ. The next three lines of code tell the collaboration from where the data should be retrieved and to where the data should be sent.

```
   public void receive(
   com.stc.connector.appconn.file.FileTextMessage input,
   com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1,
   com.stc.connector.appconn.file.FileApplication FileClient_1
)
      throws Throwable
         {
```

Since the goal of this collaboration is to retrieve data from a file and send the data, you need to create an instance of the `SNAOutboundApplication` interface in order to deliver conversation traffic. In order for the collaboration to know from where to retrieve the data that is to be sent via the `SNAOutboundApplication` interface, you must create an instance of the `FileApplication` interface that belongs to the File eWay. This will allow you to read the data file on the local system. Error handling is handled by the `Throwable` class.

```
         // @map:logger.info("SNAOut1: Started.")
         logger.info( "SNAOut1: Started." );
         // @map:Copy ByteArray to DataOut
         SNALU62eWay_1.setDataOut( input.getByteArray() );
```

One of the first things you should do before processing any conversation traffic is turn on the logging feature. Here, the logger is instantiated with a specific phrase to include in the log file. Now that limited debugging for the collaboration is available, the next step is to tell the collaboration to where the data that will be transmitted should be sent. Using the getByteArray method available for the File eWay, the data can be read with this: `input.getByteArray()`.

A moderately simple construct of the `SNAOutboundApplication` interface can use this `getByteArray` method to load the data into the send buffer before being transmitted. In order to load the data to the outbound buffer, you can use something similar to this: `SNALU62eWay_1.setDataOut( input.getByteArray())`. This sets the content of the outgoing payload buffer to the data found in the file.

```
         SNALU62eWay_1.send();
```

After the payload buffer is filled with the contents of the conversation message to be delivered, you can then send the message. Sending a message can be performed by using the Helper `send()` method of the exposed Java methods (refer to **"SNA eWay Javadocs" on page 69**). The `send()` method sends the outgoing payload (buffer) represented in the OTD as node DataOut into the local LU's send buffer for transmission to the partner TP with confirmation flag. When the flag is true, a CPIC `cmcfm` will be called after the data is sent. For the general logic flow of this helper

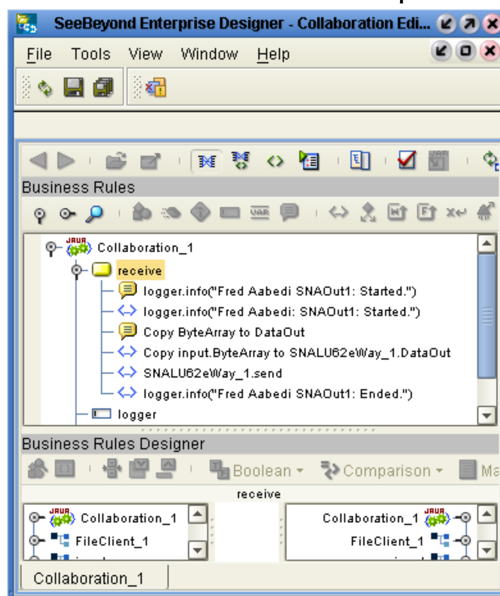
function, you may need to refer to the native interface
`com.stc.connector.snalu62.jni.SNAInterface#send(boolean).`

```
// @map:logger.info("SNAIn1: Ended.")
logger.info( "SNAIn1: Ended." );
```

After the conversation traffic has been sent, a final log message is displayed,
`logger.info( "SNAIn1: Ended." ).`

If you are using the business rule designer to create your collaboration, see Figure 22 for the business rules associated with this collaboration.

Figure 22 Outbound collaboration (Helper) business rules



4.2.3 Inbound and Outbound SNA Conversations

A collaboration that both accepts and transmits conversation initialization requests is achieved by creating a default SNA collaboration--see **Creating Default Java Collaborations** on page 35 for details. After the default collaboration is generated, you can then modify the collaboration to suit your application's needs.

4.3 Best Practices

Best practices are not intended to teach how to program SNA; it is assumed that you already know how to program SNA. This section attempts to provide guidelines that

you should follow when using the SNA eWay to create SNA Java collaborations and interfaces.

An SNA conversation is the connection between the two transaction programs (TP). When a TP want to communicate with another TP, it must first contact each of the other potential TPs and determine their state. Each transaction program in the conversation should be aware of the other TP and work together. If one side is expecting the other side to do perform certain operations, and vise versa, you should ensure that the behaviors are present, accurate, and accounted for; as, any change on one side of the conversation may affect the other side.

4.3.1 Checking Conversation State

When writing collaborations, it is a good programming procedure to check the pre-condition and post-condition for any outstanding function call or major logic check-point. Depending on the returned logic or function state, you can then determine the next course of action. For example, after you call an SNA receive function, you should check the `lastReturnCode,` exposed as an OTD node, as demonstrated in Figure 23.

Figure 23 Inbound conversation collaboration

```
public void receive(
  com.stc.connector.snalu62.inbound.SNAInboundApplication
  input, com.stc.connector.appconn.file.FileApplication FileClient_1 )
     throws Throwable
{
// it works along with project SNAOut1_Confirm which sends data and
//  requests confirmation.
// main logic: receive data over SNA, write it into file, then confirm it.
  logger.info( "SNAIn1_Confirm: Started." );
  // you can do whatever other logic before/after this CPIC call according
  // to your own actual case
  input.getCPICCalls().cmrcv();
  if (input.getLastReturnCode() !=
      input.getConstants().getReturnCodes().getCM_OK()) {
          logger.error( "SNAIn1_Confirm: cmrcv: Failed." );
          throw new Exception( "SNAIn1_Confirm: cmrcv: Failed." );
          // or do your own error handling
  }
  //Conversation processing goes here
```

In the above example, a CPIC call, `input.getCPICCalls().cmrcv()`, is used to accept an initiate conversation request. A `getLastReturnCode` is immediately called on the inbound conversation to obtain the return code from the last SNA conversation-related function call. The logic, as implemented here, checks to see if the returned code from the `cmrcv()` call matches the return code value of `getCM_OK`. If the returned codes match, the data from the conversation is captured and output to the target file as specified in the remainder of the collaboration code. If the return code obtained by the `input.getLastReturnCode()` call does not match, an exception is thrown and logged.

Similar in construct to the previous example, this next inbound conversation code segment, Figure 24, uses the Confirmed (`cmcfmd`) CPIC call to send a confirmation reply to the remote program confirmation request. The local and remote programs can use the Confirmed and Confirm calls to synchronize their processing.

Figure 24 Inbound conversation confirmation

```
// you can do whatever other logic before/after this CPIC call according
to your design
input.getCPICCalls().cmcfmd();
if (input.getLastReturnCode() !=
  input.getConstants().getReturnCodes().getCM_OK()) {
  logger.error( "SNAIn1_Confirm: cmcfmd: Failed." );
  throw new Exception( "SNAIn1_Confirm: cmcfmd: Failed." );
  // or do your own error handling
}
//Conversation processing goes here
```

To maintain the cohesion between the inbound conversation collaboration and the outbound conversation collaboration, the outbound conversation code, Figure 25, will request a confirmation of the conversation state before proceeding. Without the outbound conversation collaboration requesting the confirmation from the inbound conversation collaboration, it is possible that unexpected results could occur since the confirmation codes are being sent to the requestor, even though the requestor didn't ask for confirmation.

Figure 25 Outbound conversation confirmation

```
SNALU62eWay_1.getCPICCalls().cmcfm();
if (SNALU62eWay_1.getLastReturnCode() !=
  SNALU62eWay_1.getConstants().getReturnCodes().getCM_OK()) {
  logger.error( "SNAOut1_Confirm: cmcfm: Failed." );
  throw new Exception( "SNAOut1_Confirm: cmcfm: Failed." );
  // or do your own error handling
}
//Conversation processing goes here
```

As you can see, the Confirm (`cmcfm`) call is used by the outbound conversation collaboration to send a confirmation request to the inbound conversation collaboration and then wait for a reply. The inbound conversation collaboration replies with a Confirmed (`CMCFMD`) call (see Figure 24). The inbound and outbound conversation collaborations use the Confirm and Confirmed calls to synchronize their processing of data.

If your design requires further conversation synchronization, you can check the `lastStatus` (exposed as an OTD node) and/or check the `lastConversationState` (exposed as an OTD node); in addition to, other confirmation status check calls (see **"SNA eWay Javadocs" on page 69** for additional information). Once the state is determined, your program flow can be modified based on these expected or unexpected conversation states. If you are not receiving expected returned values, you will know that something is wrong and can process the conversation accordingly.

Depending on which type of Java calls you use in your collaboration code, you will need to select the proper calls that are related. A (non-exclusive) list of commly used related confirmation methods is provided below:

- cpic `cmcfm()`
- cpic `cmcfmd()`

- helper `confirm()`
- helper `confirmed()`
- helper `send()` or `send(true)`
- helper `recv()` or `recv(true)`

4.3.2 Using CPIC Calls

For the users that want to use CPIC calls (`cmxxxx`) directly, the eWay and integration server manage conversation initiation and termination. Normally, it is not necessary to explicitly call the CPIC calls (e.g. `cmaccp`, `cminit`, `cmdeal`, etc.) that manage conversation initiation and termination. Unless your design requires you to manage the conversation on your own logic, of course risk also, you need not implement CPIC calls for conversation handshakes.

The collaborations discussed in this chapter and the collaborations provided with the sample projects (**"Implementing SNA eWay Projects" on page 54**) demonstrate several examples of common practices as applied to the SNA eWay.

Chapter 5

Implementing SNA eWay Projects

This chapter provides an introduction to how the SNA eWay components are created and implemented in an eGate project. It is assumed that the reader understands the basics of creating a project using the SeeBeyond Enterprise Designer. For more information on creating an eGate project see the *eGate Integrator Tutorial* and the *eGate Integrator User's Guide*.

What's in This Chapter

- **About the Sample Projects** on page 54
- **Locating the Sample Projects** on page 55
- **Importing Projects** on page 55
- **Running SNA eWay Projects** on page 57
- **Building SNA Business Logic with eGate** on page 63

5.1 About the Sample Projects

The SNA eWay includes the following sample projects that you can import. This enables you to see how ICAN projects can work with SNA applications.

- SNAIn_CPIC for use with eGate
- SNAOut_CPIC for use with eGate
- SNAIn_helper for use with eGate
- SNAOut_helper for use with eGate

SNA1In and SNAOut1 projects demonstrate the use of the SNA eWay using CPIC functions. SNAIn1_helper and SNAOut1_helper demonstrate how to use the SNA eWay with the windows helper functions.

Learn more about the sample projects:

- **"Sample Project Contents" on page 54**
- **"Sample Project Zip Files" on page 55**

5.1.1 Sample Project Contents

Each project contains the following:

- Input data
- Connectivity Maps
- Collaborations Definitions
- Inbound or outbound SNA eWays

The sample projects provide a project that allows you to browse its configurations to learn how inbound and outbound SNA projects are designed. The projects do not include ICAN Environments and deployment profiles necessary to deploy the sample projects. To learn how to complete the projects for deployment, refer to **"Running SNA eWay Projects" on page 57**.

5.1.2 Sample Project Zip Files

The SNA eWay sample projects are provided as a zip file, **SNA_eWay_Sample.zip**, which contains the following files:

- **SNAIn_CPIC.zip** for the SNA project (eGate)
- **SNAOut_CPIC.zip** for the SNA project (eGate)
- **SNAIn_helper.zip** for the SNA project (eGate)
- **SNAOut_helper.zip** for the SNA project (eGate)
- **SNAOut1_input1.txt** (input file)

5.2 Locating the Sample Projects

The SNA eWay sample projects are included in the **SNAeWayDocs.sar** file. This file is uploaded separately from the SNA eWay sar file during installation. For information, refer to **"Installing the eWay" on page 16**.

Once you have uploaded the **SNAeWayDocs.sar** to the repository and you have downloaded the sample projects (**SNA_eWay_Sample.zip**) via the **DOCUMENTATION** tab in the Enterprise Manager, the samples resides in the folder you specified during the download.

5.3 Importing Projects

You can import the SNA projects as described below. To find out where the sample projects reside, refer to **"Locating the Sample Projects" on page 55**.

To import the sample projects

1. Unzip the **SNA_eWay_Sample.zip** file. This creates the following zip files:
   - **SNAIn_CPIC.zip** (for the eGate project)
   - **SNAOut_CPIC.zip**(for the eGate project)

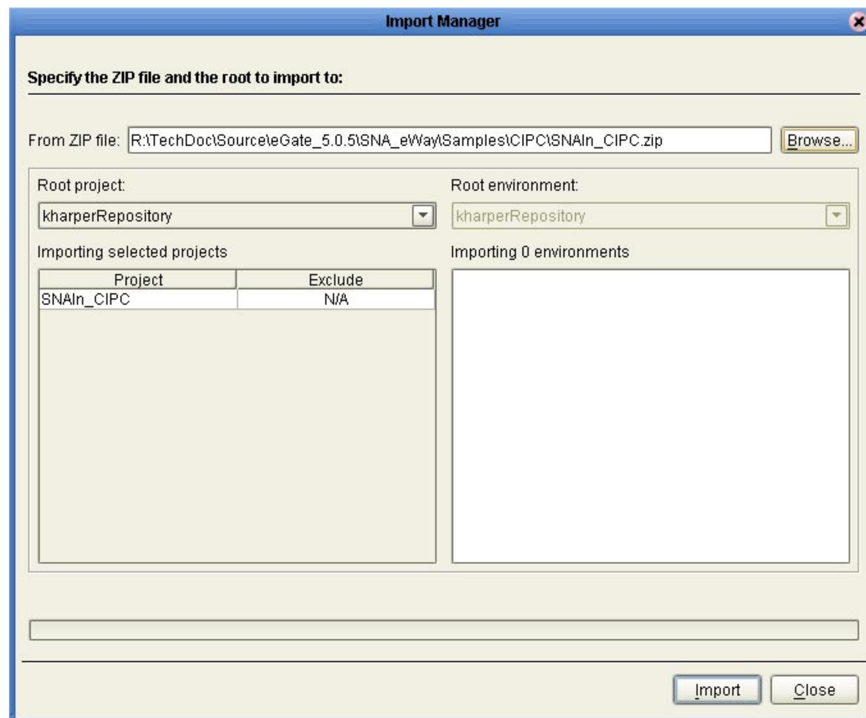
- **SNAIn_helper.zip** (for the eGate project)
- **SNAOut_helper.zip** (for the eGate project)
- **SNAOut1_input1.txt** (input file)

2 In the **Project Explorer** tab of the Enterprise Designer, right-click the repository and click **Import**. The **Import Manager** dialog box appears.

3 Click **Browse** and navigate to the folder where you unzipped the sample zip file.

4 Click the desired sample file. The **Import Manager** dialog box appears similar to the following:

Figure 26 Import Manager Dialog Box



5 Click **Import**. A dialog box confirms that the project import was successful.

6 Click **OK** and click **Close**.

You can now explore the connectivity maps, the OTDs, and the business logic for the collaborations or business processes.

5.4 Running SNA eWay Projects

The sample projects do not include the eGate environments, deployment profiles, and the physical configurations for the eWays needed to deploy the projects. To deploy SNA projects, perform the following after importing or creating new projects:

1. Configure the eWay properties– see **“Building SNA Business Logic with eGate” on page 63**.
2. Create the environment profile - see **“Creating the Environment Profile” on page 57**.
3. Configure the eWay environment properties - see **“Inbound Environment Properties” on page 30** or **“Outbound Environment Properties” on page 32**.
4. Configure the logical host - see **“Configuring the Logical Host” on page 58**.
5. Deploy the project - see **“Deploying the Project” on page 60**.
6. Run the project - see **“Running the Sample Project” on page 61**

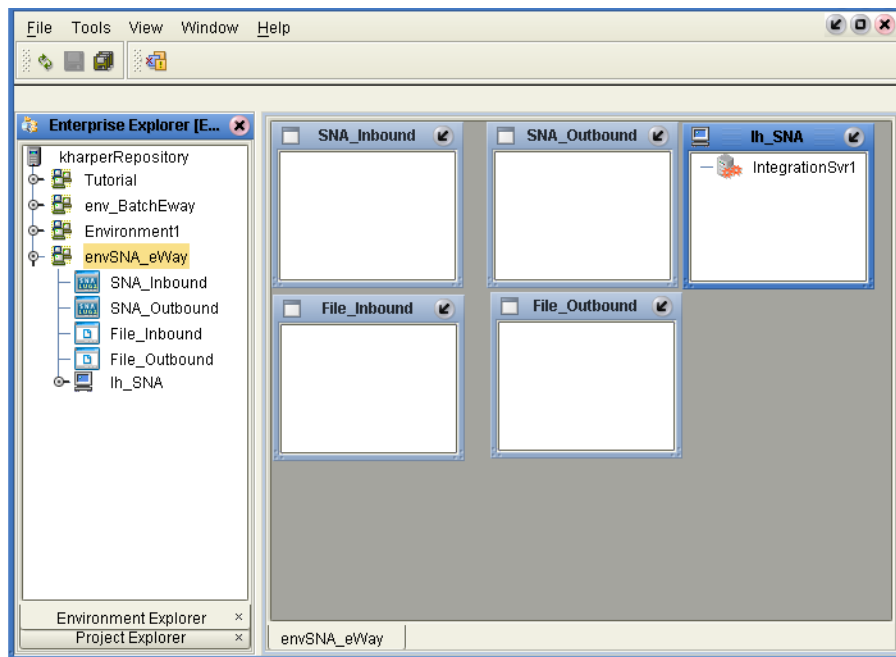
5.4.1 Creating the Environment Profile

The procedure below describes how you create an eGate Environment for the SNA sample projects. For detailed information about creating Environments, refer to the *eGate Integrator User’s Guide*.

1. In the Environment Explorer tab of the Enterprise Designer, right-click the repository and click **New Environment**.
2. Right-click the environment and click **New File External System** to add a File eWay. The list below shows which external systems to add for which project collaboration:
   - SNAIn_CPIC: one outbound File eWay
   - SNAOut_CPIC: one inbound File eWay and one outbound File eWay
   - SNAIn_helper: one outbound File eWay
   - SNAOut_helper: one inbound File eWay and one outbound File eWay
3. Right-click the environment and click **New SNALU62 External System** to add an SNA eWay. The list below shows which systems to add for which project collaboration:
   - SNAIn_CPIC: one inbound SNA eWay
   - SNAOut_CPIC: one outbound SNA eWay
   - SNAIn_helper: one inbound SNA eWay
   - SNAOut_helper: one outbound SNA eWay

Figure 27 shows the completed environment profile--should you choose to host all the projects in the same environment profile. If you choose to deploy each project to a separate environment profile, your environment profile may not have all of the external systems as displayed in Figure 27. Only the external systems, as specified in steps 2 and 3 for a particular project, will be present.

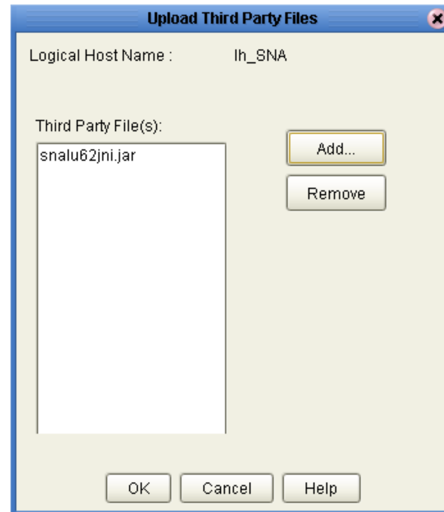
Figure 27 eGate environment for sample projects



5.4.2 Configuring the Logical Host

Before you can execute any projects created with the SNA eWay, you must add the SNA eWay Runtime JNI to the logical host.

1. If the logical host is not already installed, download and install [but do not bootstrap) the logical host as described in the *ICAN Suite Installation Guide*. Do not
2. From the Environment Explorer, right-click the targeted environment and click **New Logical Host**.
3. Launch the Enterprise Manager.
4. From the Enterprise Manager, click the **DOWNLOADS** tab.
5. Download and save the eWay Runtime JNI file to the local system.
6. Return to the Enterprise Designer's Environment Explorer.
7. Right-click the targeted logical host name for your SNA eWay project and select: **Upload File...**
8. Select the path and file name for the Runtime JNI file. The Upload Third Party Files window appears.



9 Confirm that the file to be uploaded is the Runtime JNI file for SNA and click **OK** to continue.

10 Right-click the logical host and click **New SeeBeyond Integration Server.**

SPARC64 logical host deployment

The SNA eWay is shipped with the Sparc64 logical host. The JVM for the Sparc64 logical host can be started in either 32-bit or 64-bit mode. By default, the JVM is started in 32-bit mode. To properly deploy the logical host for Sparc 64, you must ensure that the JVM bit mode matches the bit size of the JNI bridge shared library for the appropriate Brixton or SNAP-IX library that you installed in **Chapter 2**, **"Installing the eWay Product Files" on page 17**.

If planning to host large EJB applications, the big EJB applications run more efficiently on Solaris 9 with the JVM configured for 64-bit execution. To modify the logical host so that the JVM starts in 64-bit mode, instead of the default 32-bit mode, you must first perform all the procedures as specified in **"Configuring the Logical Host" on page 58**, before continuing with the steps described below.

To set the JVM to start in 64-bit mode:

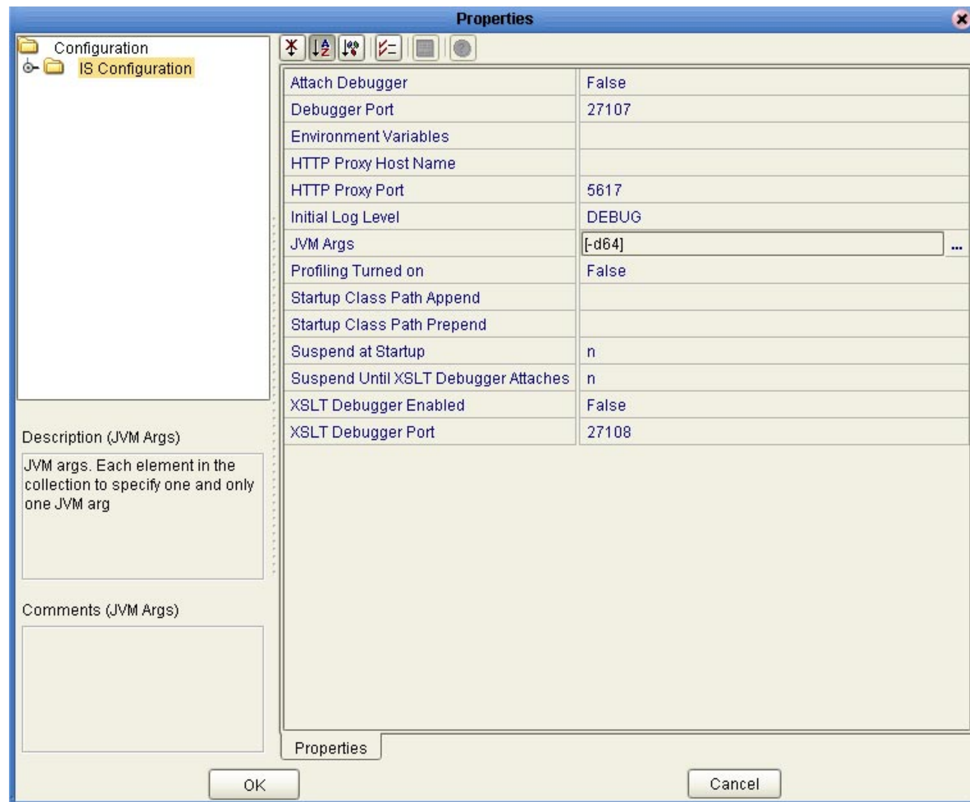
1 Open the Enterprise Designer and navigate to the Environment Explorer.

2 Locate the logical host for the integration server whose JVM you want to start in 64-bit mode.

3 Right-click the integration server and click **Properties**.

4 Set the JVM Args attribute to ***-d64***. See Figure 28 for details.

Figure 28 Configuring JVM for 64-bit mode



5 Click **OK** to save the integration server settings.

5.4.3 Deploying the Project

Once you have created the environment and added its components, you can create the deployment profiles. The procedure below describes how to create deployment profiles for the inbound and outbound collaborations.

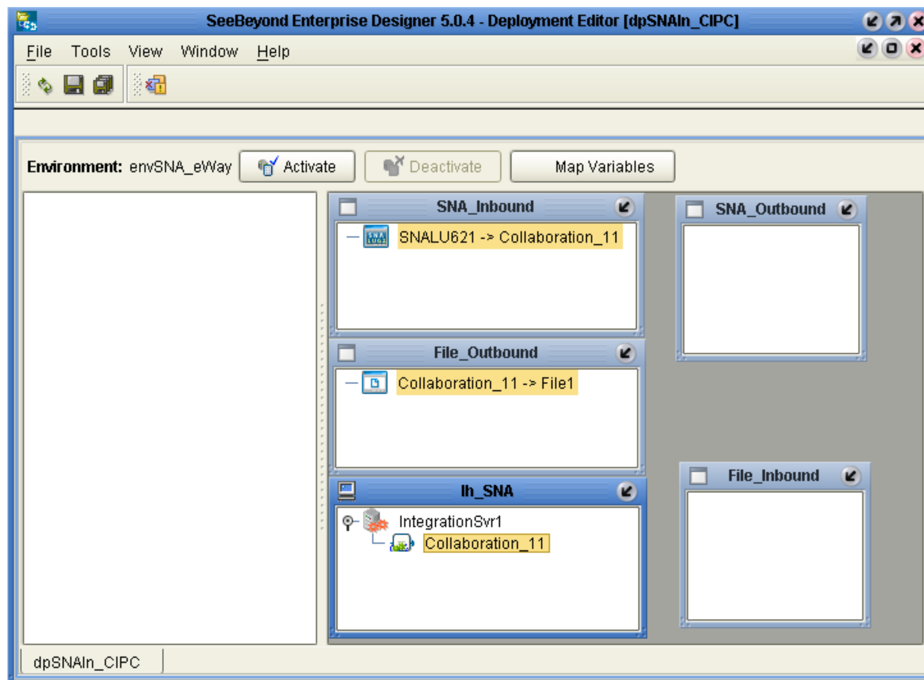
To create Deployment Profiles for sample projects

1 In the Project Explorer tab of the Enterprise Designer, right-click the project and click **New Deployment Profile**.

2 Enter a name for the inbound deployment profile, and select the environment you created for the sample.

3 Double-click the inbound deployment profile. Drag the project components to the environment component as shown in Figure 29.

Figure 29 Deployment profile



4 Click **Activate** to enable the deployment profile.

5.4.4 Running the Sample Project

Before you run SNA eWay projects, you should ensure all other project preparation procedures have been completed. Refer to **"Running SNA eWay Projects" on page 57** for a complete list of tasks required to run SNA eWay projects. Additionally, specific logical host bootstrap procedures must be adhered to for this eWay. Modifications to the bootstrap procedures are described below and should be implemented according to the logical host platform:

- **Windows 2000/XP/Windows Server 2003** on page 61
- **IBM AIX 5.1L and 5.2 (32-bit)** on page 62
- **IBM AIX 5.1L and 5.2 (64-bit)** on page 62
- **Sparc (32-bit)** on page 62
- **Sparc (64-bit)** on page 63

Refer to the *ICAN Suite Installation Guide* for general instructions about bootstrapping the logical host.

Windows 2000/XP/Windows Server 2003

1 Ensure the stc_jnisna.dll resides in a directory specified in the system PATH statement. Refer to **"Installing the eWay Product Files" on page 17** for instructions on how to download and configure the stc_jnisna.dll file.

2 Run the **bootstrap.bat** file.

IBM AIX 5.1L and 5.2 (32-bit)

1. Ensure the libstc_jnisna.so resides in a directory specified in the system LIBPATH statement. Refer to **"Installing the eWay Product Files" on page 17** for instructions on how to download and configure the libstc_jnisna.so file.
2. Add the IBM Communication Server directory path (e.g., /usr/lib/sna) to the system LIBPATH.
3. From a command prompt, execute the following:

```
EXPORT OBJECT_MODE=32

EXPORT LIBPATH
```

4. Execute the **bootstrap.sh** file with the **-32bit** parameter (plus other required parameters).

IBM AIX 5.1L and 5.2 (64-bit)

1. Ensure the libstc_jnisna.so resides in a directory specified in the system LIBPATH statement. Refer to **"Installing the eWay Product Files" on page 17** for instructions on how to download and configure the libstc_jnisna.so file.
2. Add the */usr/sna/lib* directory to the system LIBPATH.
3. From a command prompt, execute the following:

```
EXPORT OBJECT_MODE=64

EXPORT LIBPATH
```

4. Execute the **bootstrap.sh** file.

Sparc (32-bit)

For Sparc 32-bit platforms, the SNA eWay supports two third-party SNA servers: SNAP-IX, and Brixton. Both the SNAP-IX and Brixton libraries can be executed in either 32-bit or 64-bit modes. Refer to **"Installing the eWay Product Files" on page 17** for instructions on how to download and configure the libstc_jnisna.so file.

1. Ensure the runtime bridge library, libstc_jnisna.so, for either the **SNA eWay - Runtime sparc32 SNAP-IX bridge so** or the **SNA eWay - Runtime sparc32 Brixton bridge so** reside in a directory that is specified in the **LD_LIBRARY_PATH** environment variable.
2. Ensure that the Brixton or SNAP-IX 32-bit library directory is specified in the **LD_LIBRARY_PATH** environment variable.
3. Ensure the JVM for the integration server specified in your logical host is set to 32-bits. Refer to **"Configuring the Logical Host" on page 58** to configure the logical host.
4. Execute the **bootstrap.sh** file.

Sparc (64-bit)

For bigger EJB applications hosted on Solaris 9, the 64-bit JVM has better performance. If you run the SeeBeyond Integration Server JVM in 64-bit mode, you must also use the 64-bit JNI bridge shared library and the corresponding SNAP-IX or Brixton 64-bit library.

1. Ensure the runtime bridge library, libstc_jnisna.so, for either the **SNA eWay - Runtime sparc64 SNAP-IX bridge so** or the **SNA eWay - Runtime sparc64 Brixton bridge so** reside in a directory that is specified in the **LD_LIBRARY_PATH** environment variable.
2. Ensure that the Brixton or SNAP-IX 64-bit library directory is specified in the **LD_LIBRARY_PATH** environment variable.
3. Ensure the JVM for the integration server specified in your logical host is set to 64-bits. Refer to **“Configuring the Logical Host” on page 58** to configure the logical host.
4. Execute the **bootstrap.sh** file.

5.5 Building SNA Business Logic with eGate

This section describes how to build the SNA collaborations:

- **Building Collaborations** on page 63
- **Adding Connectivity Maps** on page 66
- **Building Inbound Connectivity Maps** on page 66
- **Building Outbound Connectivity Maps** on page 67

To see an example of SNA collaborations and connectivity maps, import the SNA sample projects as described in **“Implementing SNA eWay Projects” on page 54**.

5.5.1 Building Collaborations

After you have built the OTDs as described in **“Configuring the eWay” on page 20**, you are ready to build Collaboration Definitions.

To build Collaborations

1. In the **Project Explorer** tab of the Enterprise Designer, right-click the project, select **New** > **Collaboration Definition (Java)**.
2. Complete steps 1 and 2 of the **Collaboration Definition Wizard (Java)**. For details about this wizard, refer to the *eGate Integrator User’s Guide*.
3. Select the SNA OTD to use in the new collaboration by traversing the **Look In** drop-down box: SeeBeyond.eWays.SNA.
4. Highlight the desired OTD name and click the **Add** button.

5 Optionally, modify the instance name of the OTD that will be used in the collaboration.

6 Click the **Finish** button.

A new collaboration associated with the SNA OTD is placed into your targeted project.

7 In the **Collaboration Editor** window, create the source code and the data mappings for the collaboration. For details about the **Collaboration Editor**, refer to the *eGate Integrator User's Guide*. For information about SNA methods, refer to **"SNA eWay Javadocs" on page 69**.

The figure below shows an example of data mapping for an inbound SNA collaboration. To explore the business logic design for an actual project, import the SNA sample projects as described in **"Importing Projects" on page 55**.

Figure 30 Inbound collaboration (SNAIn_CPIC)

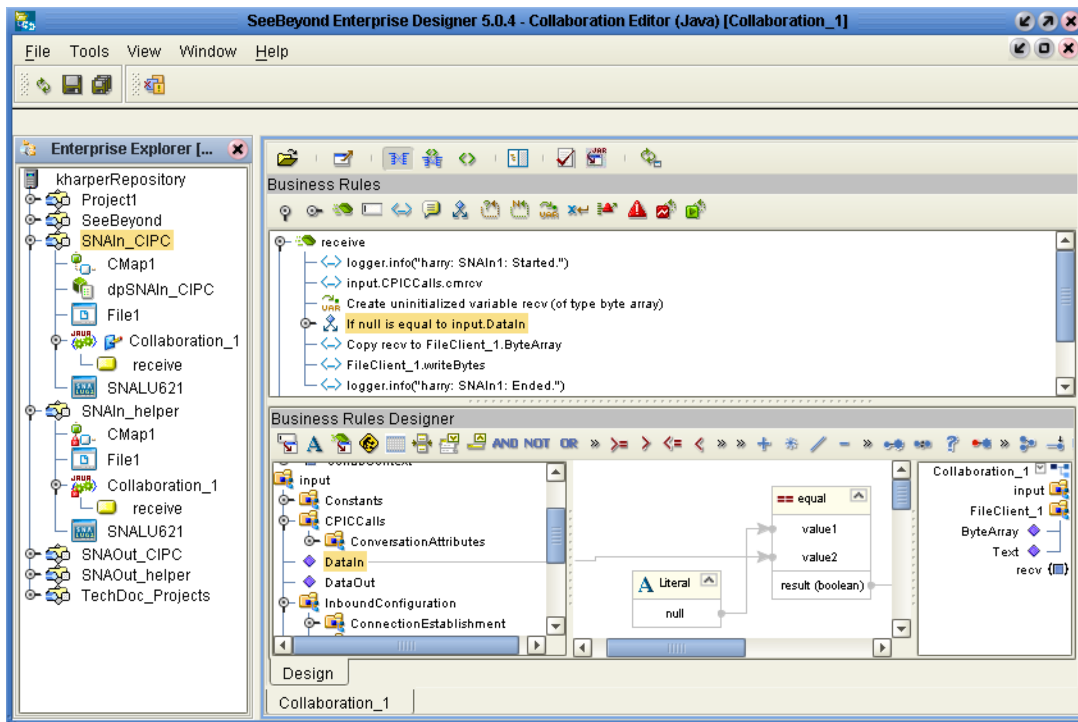


Figure 31 Inbound collaboration (SNAIn_helper)

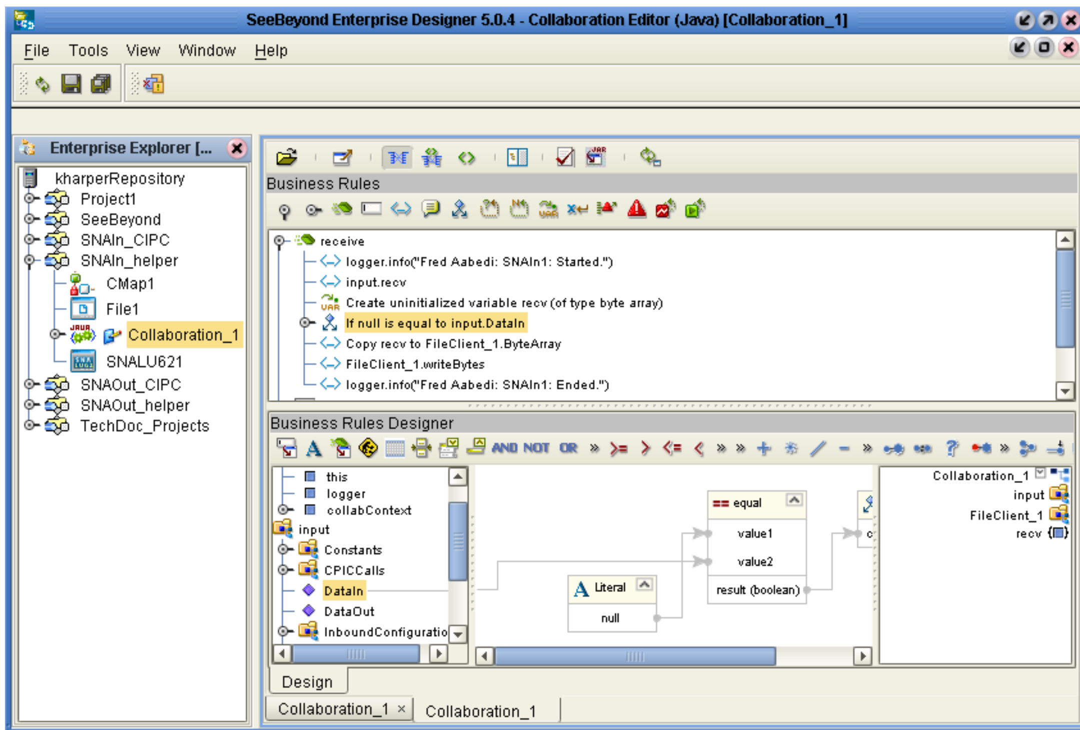


Figure 32 Outbound collaboration (SNAOut_CPIC)

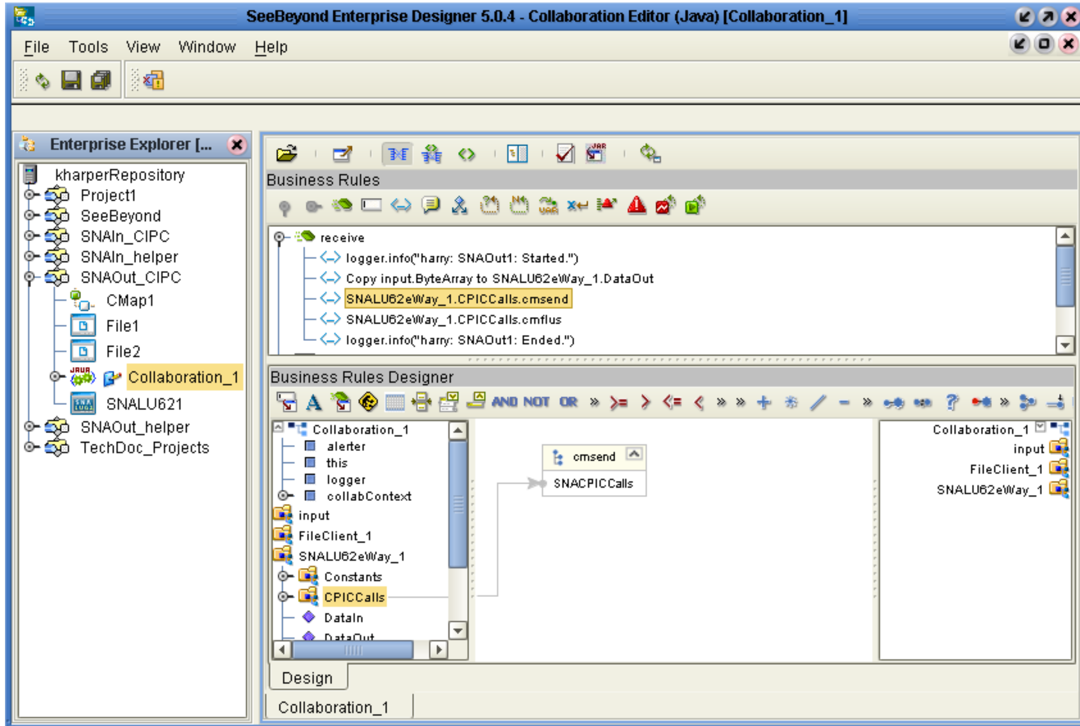
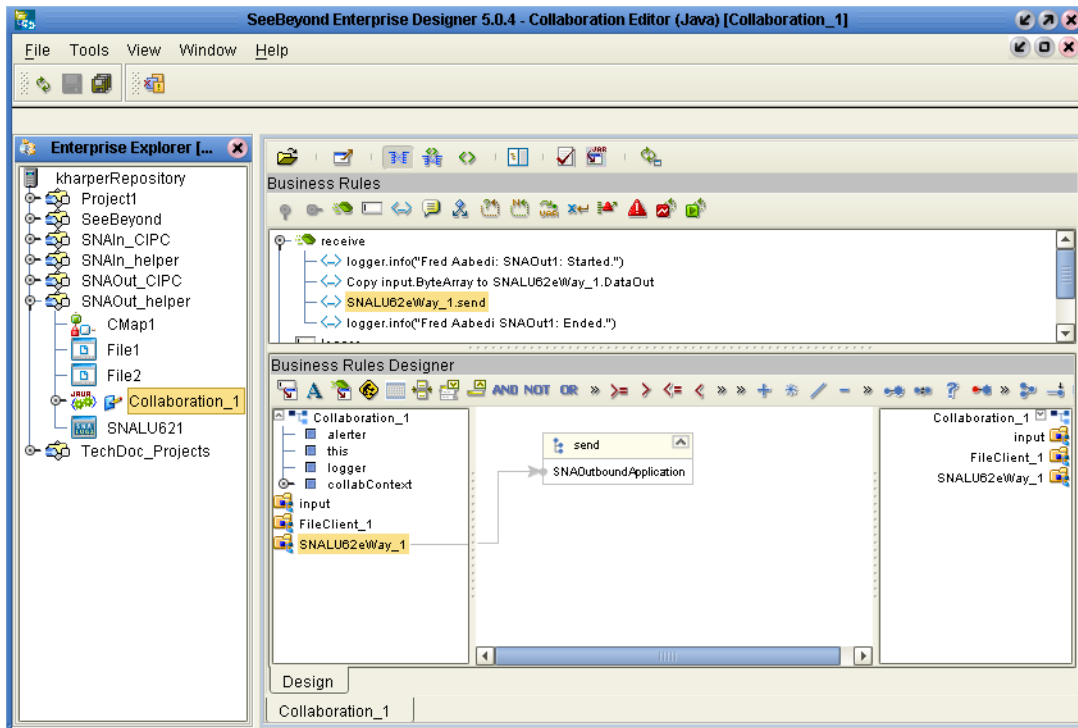


Figure 33 Outbound collaboration (SNAOut_helper)



5.5.2 Adding Connectivity Maps

To add Connectivity Maps

- In the **Project Explorer** tab of the Enterprise Designer, right-click the project for which you intend to create a connectivity map, click **New > Connectivity Map**.

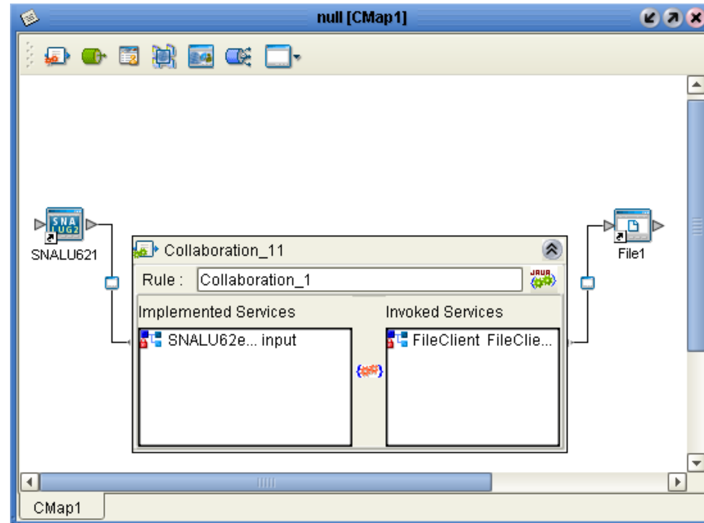
5.5.3 Building Inbound Connectivity Maps

To build inbound SNA Connectivity Maps

1. From the connectivity map, click the **External Applications** icon then select **SNALU62 External Application**.
2. Add other components such as other eWays and collaborations to the connectivity map.
3. Drag the inbound collaboration from the **Project Explorer** tab into the collaboration icon in the connectivity map.
4. Link and configure the SNA eWays. Refer to **"Configuring the eWay" on page 20** for eWay configuration details.
5. Link and configure all components. For details, refer to the *eGate Integrator User's Guide*.

The figure below shows an example of an inbound SNA connectivity map. To explore the connectivity map for an actual project, import the SNA sample project as described in **"Importing Projects" on page 55**.

Figure 34 Inbound SNA connectivity map



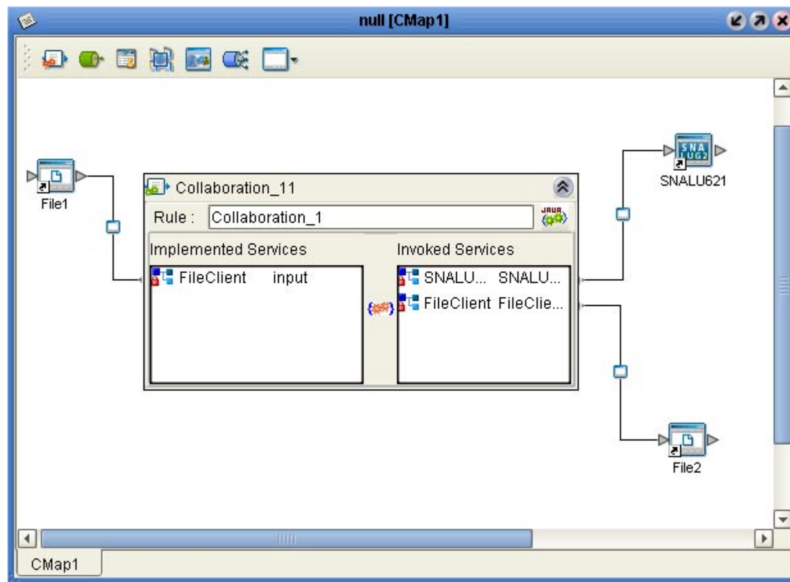
5.5.4 Building Outbound Connectivity Maps

To build outbound SNA connectivity maps

1. From the connectivity map, click the **External Applications** icon then select **SNALU62 External Application**.
2. Add other components such as other eWays and collaborations to the connectivity map.
3. Drag the outbound collaboration from the **Project Explorer** tab into the collaboration icon in the connectivity map.
4. Link and configure all components. For details, refer to the *eGate Integrator User's Guide*.

The figure below shows an example of an outbound SNA connectivity map. To explore the connectivity map for an actual project, import the SNA sample project as described in **"Importing Projects" on page 55**.

Figure 35 Outbound SNA connectivity map



5.5.5 SNA Collaborations

Newly created Java collaborations that implement the SNA OTD consist of skeleton SNA functionality required to use the collaboration with your project. The collaboration code generated upon SNA Java collaboration creation is provided in Figure 36.

Figure 36 Default Collaboration Code

```
package TechDocSNAtdSNA;
public class Collaboration_1
{
   public void receive(
     com.stc.connector.snalu62.inbound.SNAInboundApplication input,
     com.stc.connector.snalu62.outbound.SNAOutboundApplication SNALU62eWay_1
   )
     throws Throwable
     {}
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

}
```

Chapter 6

SNA eWay Javadocs

For a complete list of the Java methods available for use with SNALU62 eWay Object Type Definitions, refer to the **Javadoc**. You can download the Javadoc while you are installing the eWay. For complete instructions, see the *SeeBeyond ICAN Suite Installation Guide*.

Index

Numerics

32-bit **62**
64-bit **62**, **63**
  JVM **59**

A

Always Create New Connection **27**
At Fixed Rate **21**, **23**
Auto Disconnect Connection **28**
Auto Reconnect Upon Matching Failure **28**

B

Best Practices **50**
bootstrap **61**
building
  Collaborations **63**

C

collaborations
  default **35**
Collaborations, building **63**
Connection Establishment **24**, **27**
  Always Create New Connection **27**
  Auto Disconnect Connection **28**
  Auto Reconnect Upon Matching Failure **28**
  Max Connection Retry **24**, **28**
  Retry Connection Interval **24**, **28**
Connection Level **26**, **30**
Connection Pool Size **24**
Connectivity Map
  inbound **20**
  outbound **27**
Connectivity Maps
  adding, eGate **66**
  inbound, eGate **66**
  outbound, eGate **67**
conventions, document **14**
Conversation State **12**, **51**
CPIC **53**
Custom Handshake Class Name **25**, **29**

D

Deallocation Type **25**, **29**
default collaborations **35**
Delay **22**, **23**
deploying Projects **61**
Deployment Profiles, creating **60**
document conventions **14**
documentation, installing **17**

E

Environment Properties
  inbound
    SNA Settings **31**
  outbound **32**
    SNA Settings **33**
Environments
  creating **57**

F

finding sample Projects **55**

G

General Settings **26**, **30**, **32**

H

Host Name **31**, **33**

I

IBM AIX **62**
importing sample Projects **55**
inbound
  Connectivity Maps, eGate **66**
  Environment Properties
    SNA Settings **31**
Inbound Connection Manager **24**
  Connection Pool Size **24**
  Scope of Connection **25**
inbound connectivity map **20**
inbound eway properties **21**, **22**, **23**, **24**, **25**, **26**
InBound Schedules
  Listner Schedule **21**
Inbound Schedules **21**
Initial Conversation **26**, **29**
installation
  SNALU62eWay.sar **17**
installing
  documentation **17**
  sample Projects **17**

J

Javadoc **69**
Javadoc, obtaining **69**
JNI
  upload JAR file **58**
JVM
  64-bit **59**

L

Listner Schedule **21**
  At Fixed Rate **21**
  Delay **22**
  Period **22**
  Scheduler **22**
Local LU Name **31**, **33**
Local TP Name
  SNA Settings
    Local TP Name **31**, **33**
Logical Host
  configure **58**
logical host **61**
  JVM **59**

M

Max Connection Retry **24**, **28**

O

Object Type Definition **34**
organization of information, document **13**
OTD **34**
OTD Level **26**, **30**
outbound
  connectivity map **27**
  Connectivity Maps (eGate) **67**
  Environment Properties **32**
    SNA Settings **33**
outbound eway properties **27**, **28**, **29**, **30**
overview
  sample Projects **54**

P

Packet Size **26**, **29**
Period **22**, **23**
Persistent Storage Location **32**, **33**

R

Resource Adapter Level **26**, **30**
Retry Connection Interval **24**, **28**

S

sample Projects
  deploying **61**
  Deployment Profiles **60**
  Environments **57**
  finding **55**
  importing **55**
  installing **17**
  overview **54**
SAR File
  SNALU62eWay.sar **17**
Schedule Type **23**
Scheduler **22**, **23**
Scope of Connection **25**
Scope of State **26**, **30**
Service Schedule **22**
  At Fixed Rate **23**
  Delay **23**
  Period **23**
  Schedule Type **23**
  Scheduler **23**
SNA Settings **25**, **29**
  Deallocation Type **25**, **29**
  Environment Properties **31**, **33**
  Initial Conversation **26**, **29**
  Local LU Name **31**, **33**
  Packet Size **26**, **29**
  Symbolic Destination Name **32**, **33**
  Synchronization Level **26**, **30**
  Timeout **26**, **30**
SNA_eWay_Sample.zip **55**
Sparc **62**, **63**
SPARC64
  JVM **59**
Symbolic Destination Name **32**, **33**
Synchronization Level **26**, **30**

T

Timeout **26**, **30**

W

Windows **61**