SeeBeyond ICAN Suite

e*Way Intelligent Adapter for CORBA-VisiBroker (Client) User's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

Monk Version

Version 20050405222348.

Contents

Chapter 6

Chapter 1

Introduction

This document provides instructions for installing and configuring the SeeBeyond™ Technology Corporation's (SeeBeyond™) e*Way™ Intelligent Adapter for CORBA-VisiBroker (Client). This chapter provides an introduction to the e*Way.

1.1 Overview

The CORBA-VisiBroker Client e*Way provides a Monk programming language interface to external applications that have CORBA interfaces. This allows access to CORBA objects via the Monk scripting language.

The CORBA-VisiBroker Client e*Way provides functions and services to applications that use CORBA to implement interface APIs. Through its Monk extension, the CORBA-Visibroker e*Way acts as a client to a CORBA object server.

The CORBA-VisiBroker Client e*Way takes advantage of the CORBA Dynamic Invocation Interface (DII) in order to utilize CORBA capabilities and allow it to function dynamically. Through DII, the CORBA-VisiBroker Client e*Way has the ability to identify interfaces at runtime. DII also allows a scripting language such as Monk to make calls into CORBA objects without the static binding normally done with the IDL (Interface Definition Language) compiler.

When a Monk script accesses an operation on a CORBA object, the Monk engine can use DII to assemble a remote method invocation from the IDL, then pass the proper arguments to the method, and finally return the result to the script.

This Chapter Includes:

1.1.1 Components

The CORBA-VisiBroker Client e*Way comprises the following:

- **stcewgenericmonk.exe**, the executable component
- Configuration files, which the e*Way Editor uses to define configuration parameters
- Monk function scripts, discussed in **Chapter 5**
- Dynamic link library (DLL)—a monk script automatically loads the DLL when the e*Way starts
- CORBA Visibroker Client Converter, see **"CORBA VisiBroker Client Converter Build Tool" on page 31**.

A complete list of installed files appears in **Table 1 on page 11**.

1.1.2 Features

The CORBA-VisiBroker Client e*Way supports the following:

- This e*Way is based on VisiBroker 3.3 for C++, which is compliant with CORBA version 2.1.

1.2 Supported Operating Systems

The CORBA-VisiBroker Client e*Way is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- Sun Solaris 8 and 9

1.3 System Requirements

To use the CORBA VisiBroker Client e*Way, you need to meet the following requirements:

- An eGate Participating Host
- A TCP/IP network connection

The e*Way must be configured and administered using the e*Gate Schema Designer.

Note: *Additional disk space can be required to process and queue the data that this e*Way processes. The amount necessary can vary based on the type and size of the data being processed and any external applications doing the processing.*

1.4 External System Requirements

The CORBA-VisiBroker Client e*Way supports the following external system:

- CORBA applications

CORBA applications may have their own requirements; see the specific application's documentation for details.

Chapter 2

Installation

This chapter covers procedures for installing the CORBA-VisiBroker Client e*Way on Windows and UNIX systems. A list of the files and directories created during installation are also included.

This Chapter Explains:

2.1 Installing the CORBA e*Way on Windows

2.1.1 Pre-installation

- Exit all Windows programs before running the setup program, including any anti-virus applications.
- You must have Administrator privileges to install this e*Way.

2.1.2 Installation Procedure

To install the CORBA-VisiBroker Client e*Way on Windows systems

1. Log in as an Administrator on the workstation on which you want to install the e*Way.
2. Insert the installation CD-ROM into the CD-ROM drive.
3. If the CD-ROM drive's "Autorun" feature is enabled, the setup application should launch automatically; skip ahead to step 4. Otherwise, use Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the file **setup.exe** on the CD-ROM drive.
4. The InstallShield setup application launches. Follow the installation instructions until you come to the **Please choose the product to install** dialog box.
5. Select **e*Gate Integrator**, then click **Next**.

6 Follow the on-screen instructions until you come to the second **Please choose the product to install** dialog box.

7 Clear the check boxes for all selections except **Add-ons**, and then click **Next**.

8 Follow the on-screen instructions until you come to the **Select Components** dialog box.

9 Highlight (but do not check) **e*Ways**, and then click the **Change** button. The **SelectSub-components** dialog box appears.

10 Select the **CORBA-VisiBroker Client e*Way**. Click the continue button to return to the **Select Components** dialog box, then click **Next**.

11 Follow the rest of the on-screen instructions to install the CORBA-VisiBroker Client e*Way. Be sure to install the e*Way files in the suggested client installation directory. The installation utility detects and suggests the appropriate installation directory. **Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested installation directory setting.**

12 Copy the file named **stcjcs.jar** from **<*eGate*>\server\registry\repository\classes** to **<*eGate*>\client\classes,** where **<*eGate*>** is the directory in which e*Gate is installed.

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the **e*Gate Integrator User's Guide**.*

2.2 Installing the CORBA e*Way on UNIX

2.2.1 Pre-Installation

You do not require root privileges to install this e*Way. Log in under the user name that you wish to own the e*Way files. Be sure that this user has sufficient privilege to create files in the e*Gate directory tree.

2.2.2 Installation Procedure

To install the CORBA-VisiBroker Client e*Way on a UNIX system

1 Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.

2 If necessary, mount the CD-ROM drive.

3 At the shell prompt, type

cd /cdrom

4 Start the installation script by typing

setup.sh

5 A menu of options will appear. Select the **Install e*Way** option. Then, follow the additional on-screen directions.

Note: *Be sure to install the e*Way files in the suggested* ***client*** *installation directory. The installation utility detects and suggests the appropriate installation directory.* ***Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested "installation directory" setting.***

6 Copy the file named **stcjcs.jar** from **<*eGate*>\server\registry\repository\classes** to **<*eGate*>\client\classes,** where **<*eGate*>** is the directory in which e*Gate is installed.

7 After installation is complete, exit the installation utility and launch the Schema Designer.

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help system.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the* ***e*Gate Integrator User's Guide****.*

2.3 Files/Directories Created by the Installation

The CORBA-VisiBroker Client e*Way installation process will install the following files within the e*Gate "client" directory tree:

Table 1 Files created by the installation

e*Gate "Client" Directory	File(s)
\bin	stcewgenericmonk.exe stc_monkcorbavb.dll
\configs\stcewgenericmonk	stcewcorba.def
\monk_library\ewcorba	corba-stdver-eway-funcs.monk corba-struct-call.monk

Chapter 3

Configuration

The CORBA-VisiBroker Client e*Way must be configured before use. This chapter describes all of the configuration parameters, including Monk configuration for connection to the external system.

This Chapter Explains:

3.1 e*Way Configuration Parameters

The CORBA-VisiBroker Client e*Way configuration parameters are set using the e*Way Editor.

To configure the CORBA-VisiBroker Client e*Way:

1. In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties.
2. Under **Configuration File**, do one of three things:
   - Click **New** to create a new file. Then, from the **e*Way Template Selection** list, select **stcewcorba** and click **OK**.
   - Click **Find** to select an existing configuration file.
   - Click **Edit** to edit the currently selected file.
3. In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of the existing command-line string. Be careful not to change any of the default arguments unless you have a specific need to do so.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *Working with e*Ways* chapter in the *e*Gate Integrators User's Guide*.

The e*Way's configuration parameters are organized into the following sections:

- General Settings
- Communication Setup
- Monk Configuration

3.1.1 General Settings

The General Settings control basic operational parameters.

Journal File Name

Description

Specifies the name of the journal file.

Required Values

A valid file name, optionally including an absolute path (for example, **c:\temp\filename.txt**). If an absolute path is not specified, the file is stored in the e*Gate **SystemData** directory. See the *e*Gate Integrator System Administration and Operations Guide* for more information about file locations.

Additional Information

An Event is journaled for the following conditions:

- When the number of resends is exceeded (see **Max Resends Per Message** in the next section)
- When its receipt is due to an external error, but **Forward External Errors** is set to **No**. (See **"Forward External Errors" on page 14** for more information.)

Max Resends Per Message

Description

Specifies the number of times the e*Way attempts to resend a message (Event) to the external system after receiving an error.

Required Values

An integer between 1 and 1,024. The default is 5.

Max Failed Messages

Description

Specifies the maximum number of failed messages (Events) that the e*Way allows. When the specified number of failed messages is reached, the e*Way shuts down and exits.

Required Values

An integer between 1 and 1,024. The default is 3.

Forward External Errors

Description

Selects whether error messages that begin with the string "DATAERR" that are received from the external system will be queued to the e*Way's configured queue. See **"Exchange Data with External Function" on page 26** for more information.

Required Values

Yes or **No**. The default value, **No**, specifies that error messages will not be forwarded.

See **"Schedule-driven Data Exchange Functions" on page 20** for information about how the e*Way uses this function.

3.1.2 Communication Setup

The Communication Setup parameters control the schedule by which the e*Way obtains data from the external system.

Note: *The schedule you set using the e*Way's properties in the Schema Designer controls when the e*Way executable will run. The schedule you set within the parameters discussed in this section (using the e*Way Editor) determines when data will be exchanged. Be sure you set the "exchange data" schedule to fall within the "run the executable" schedule.*

Start Exchange Data Schedule

Description

Establishes the schedule to invoke the e*Way's **Exchange Data with External** function.

Required Values

One of the following:

- One or more specific dates/times
- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Also required: If you set a schedule using this parameter, you must also define all three of the following:

- Exchange Data With External Function
- Positive Acknowledgment Function
- Negative Acknowledgment Function

If you do not do so, the e*Way will terminate execution when the schedule attempts to start.

Additional Information

When the schedule starts, the e*Way determines whether it is waiting to send an ACK or NAK to the external system (using the Positive and Negative Acknowledgment functions) and whether the connection to the external system is active. If no ACK/NAK

is pending and the connection is active, the e*Way immediately executes the **Exchange Data with External** function. Thereafter, the **Exchange Data with External** function will be called according to the **Exchange Data Interval** parameter until the **Stop Exchange Data Schedule** time is reached.

See **"Exchange Data with External Function" on page 26**, **"Exchange Data Interval" on page 15**, and **"Stop Exchange Data Schedule" on page 15** for more information.

Stop Exchange Data Schedule

Description

Establishes the schedule to stop data exchange.

Required Values

One of the following:

- One or more specific dates/times
- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Exchange Data Interval

Description

Specifies the number of seconds the e*Way waits between calls to the **Exchange Data with External** function during scheduled data exchanges.

Required Values

An integer between 0 and 86,400. The default is 120.

Additional Information

If **Zero Wait Between Successful Exchanges** is set to **Yes** and the **Exchange Data with External Function** returns data, The **Exchange Data Interval** setting will be ignored and the e*Way will invoke the **Exchange Data with External Function** immediately.

If this parameter is set to zero, there will be no exchange data schedule set and the **Exchange Data with External Function** will never be called.

See **"Start Exchange Data Schedule" on page 14** and **"Stop Exchange Data Schedule" on page 15** for more information about the data-exchange schedule.

Down Timeout

Description

Specifies the number of seconds that the e*Way will wait between calls to the External Connection Establishment function. See **"External Connection Establishment Function" on page 27** for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Up Timeout

Description

Specifies the number of seconds the e*Way will wait between calls to the External Connection Verification function. See **"External Connection Verification Function" on page 27** for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Resend Timeout

Specifies the number of seconds the e*Way will wait between attempts to resend a message (Event) to the external system, after receiving an error message from the external system.

Required Values

An integer between 1 and 86,400. The default is 10.

Zero Wait Between Successful Exchanges

Description

Selects whether to initiate data exchange after the **Exchange Data Interval** or immediately after a successful previous exchange.

Required Values

Yes or **No**. If this parameter is set to **Yes**, the e*Way will immediately invoke the **Exchange Data with External** function if the previous exchange function returned data. If this parameter is set to **No**, the e*Way will always wait the number of seconds specified by **Exchange Data Interval** between invocations of the **Exchange Data with External** function. The default is **No**.

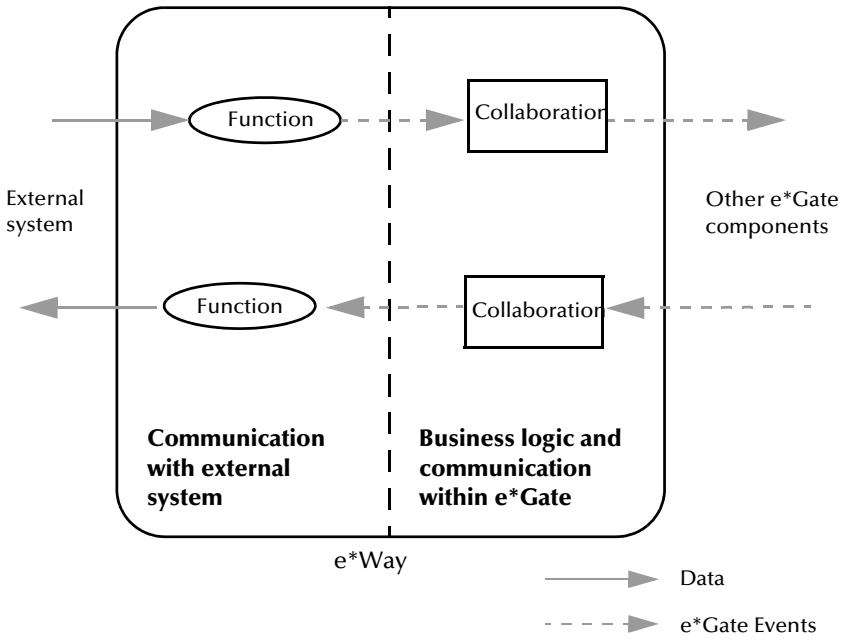
See **"Exchange Data with External Function" on page 26** for more information.

3.1.3 Monk Configuration

The parameters in this section help you set up the information the e*Way requires to utilize Monk for communication with the external system.

Conceptually, an e*Way is divided into two halves. One half of the e*Way (shown on the left in Figure 1, below) handles communication with the external system; the other half manages the Collaborations that process data and subscribe or publish to other e*Gate components.

Figure 1 e*Way internal architecture



The "communications half" of the e*Way uses Monk functions to start and stop scheduled operations, exchange data with the external system, package data as e*Gate "Events" and send those Events to Collaborations, and manage the connection between the e*Way and the external system. The **Monk Configuration** options discussed in this section control the Monk environment and define the Monk functions used to perform these basic e*Way operations. You can create and modify these functions using the SeeBeyond Collaboration Rules Editor or a text editor (such as **Notepad** or UNIX **vi**).

The "communications half" of the e*Way is single-threaded. Functions run serially, and only one function can be executed at a time. The "business logic" side of the e*Way is multi-threaded, with one executable thread for each Collaboration. Each thread maintains its own Monk environment; therefore, information such as variables, functions, path information, and so on cannot be shared between threads.

Operational Details

The Monk functions in the "communications half" of the e*Way fall into the following groups:

Type of Operation	Name
Initialization	**Startup Function** on page 25 (also see **Monk Environment Initialization File** on page 24)
Connection	**External Connection Establishment Function** on page 27 **External Connection Verification Function** on page 27 **External Connection Shutdown Function** on page 28

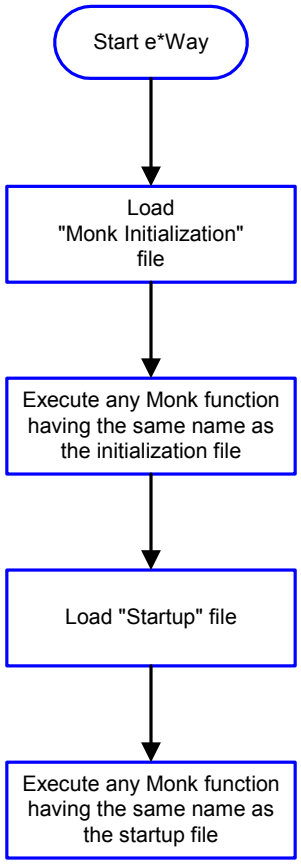
Type of Operation	Name
Schedule-driven data exchange	**Exchange Data with External Function** on page 26 **Positive Acknowledgment Function** on page 28 **Negative Acknowledgment Function** on page 29
Shutdown	**Shutdown Command Notification Function** on page 29
Event-driven data exchange	**Process Outgoing Event Function** on page 25

A series of figures on the next several pages illustrates the interaction and operation of these functions.

Initialization Functions

Figure 2 illustrates how the e*Way executes its initialization functions.

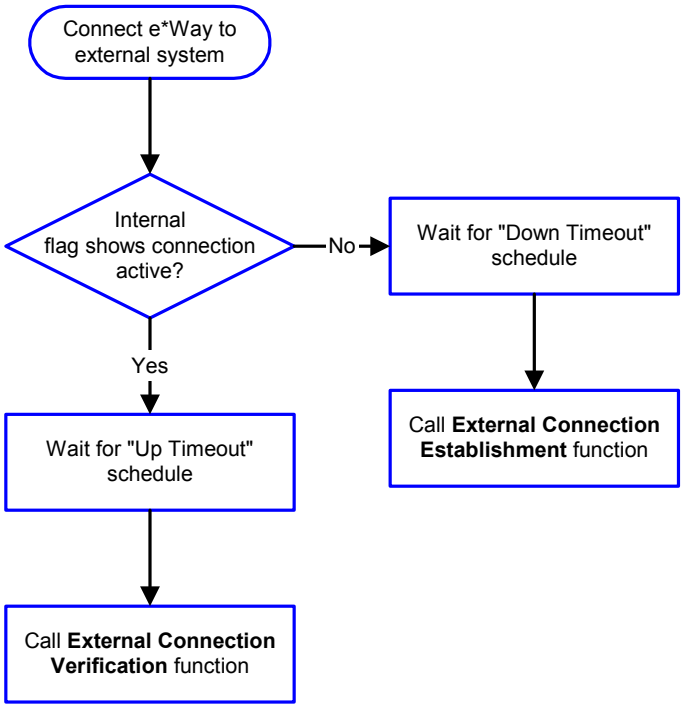
Figure 2 Initialization Functions



Connection Functions

Figure 3 illustrates how the e*Way executes the connection establishment and verification functions.

Figure 3 Connection establishment and verification functions

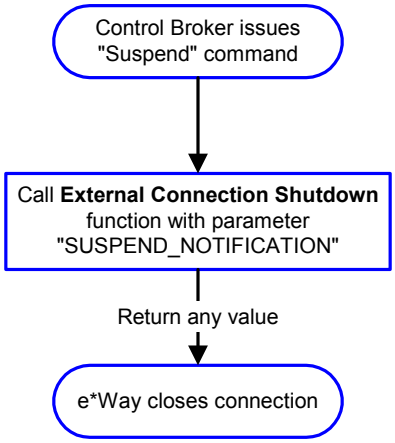


Note: *The e*Way selects the connection function based on an internal "up/down" flag rather than a poll to the external system. See* **Figure 5 on page 21** *and* **Figure 7 on page 23** *for examples of how different functions use this flag.*

User functions can manually set this flag using Monk functions. See **send-external-up** *on page 39 and* **send-external-down** *on page 40 for more information.*

Figure 4 illustrates how the e*Way executes its "connection shutdown" function.

Figure 4 Connection shutdown function



Schedule-driven Data Exchange Functions

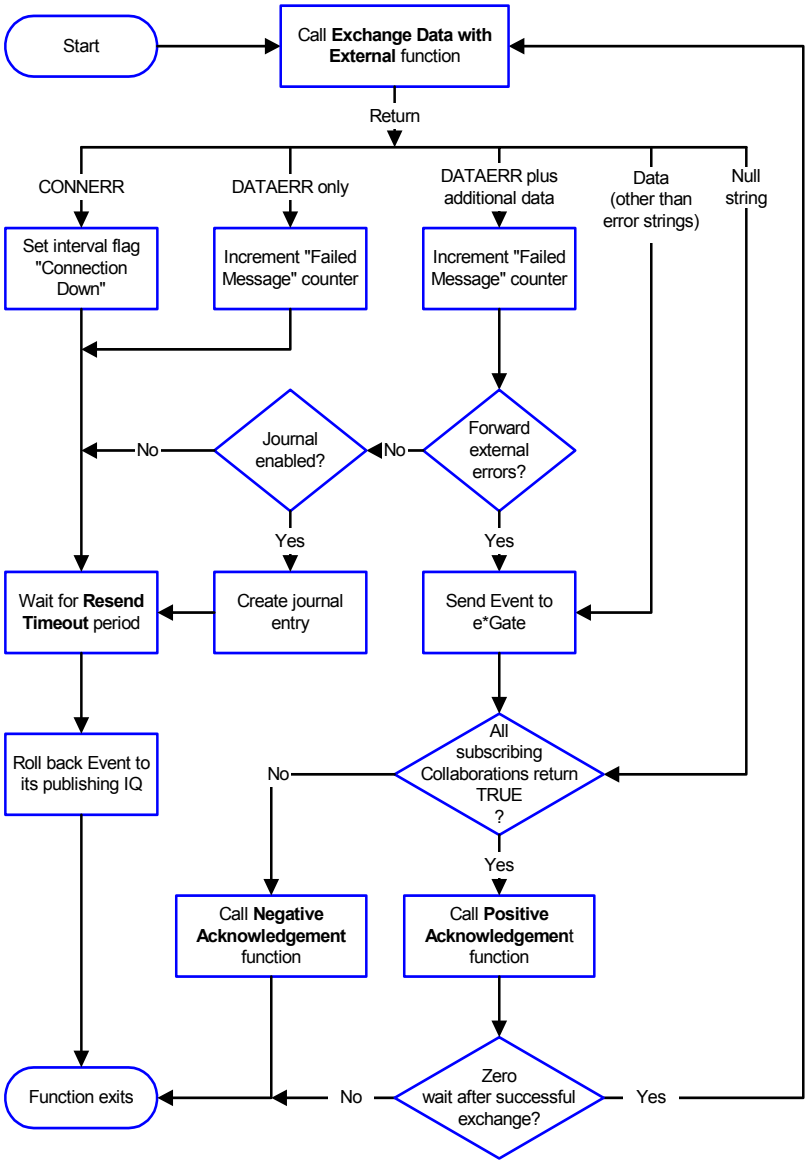
Figure 5 (on the next page) illustrates how the e*Way performs schedule-driven data exchange using the **Exchange Data with External Function**. The **Positive Acknowledgement Function** and **Negative Acknowledgement Function** are also called during this process.

"Start" can occur in any of the following ways:

- The "Start Data Exchange" time occurs
- Periodically during the data-exchange schedule (after "Start Data Exchange" time, but before "Stop Data Exchange" time), as set by the Exchange Data Interval
- The **start-schedule** Monk function is called

After the function exits, the e*Way waits for the next "start schedule" time or command.

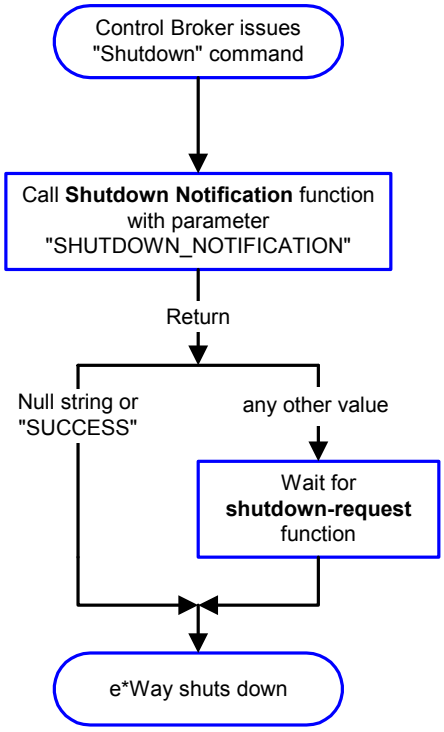
Figure 5 Schedule-driven data exchange functions



Shutdown Functions

illustrates how the e*Way implements the shutdown request function.

Figure 6 Shutdown functions



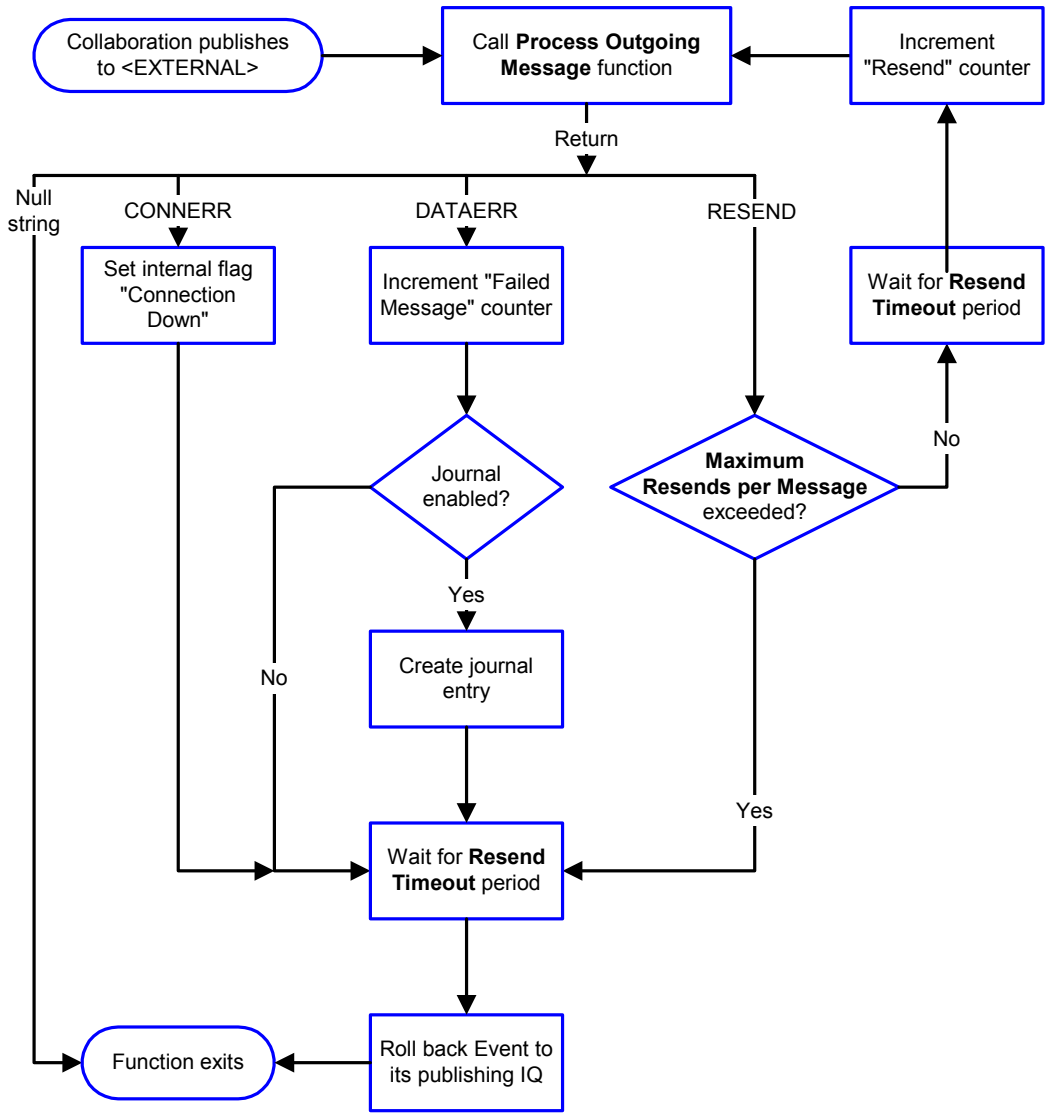
Event-driven Data Exchange Functions

Figure 7 on the next page illustrates event-driven data-exchange using the **Process Outgoing Message Function**.

Every two minutes, the e*Way checks the "Failed Message" counter against the value specified by the **Max Failed Messages** parameter. When the "Failed Message" counter exceeds the specified maximum value, the e*Way logs an error and shuts down.

After the function exits, the e*Way waits for the next outgoing Event.

Figure 7 Event-driven data-exchange functions



How to Specify Function Names or File Names

Parameters that require the name of a Monk function will accept either a function name or a file name. If you specify a file name, be sure that the file has one of the following extensions:

- .monk
- .tsc
- .dsc

Additional Path

Description

Specifies a path to be appended to the "load path," the path Monk uses to locate files and data (set internally within Monk). The directory specified in Additional Path will be searched after the default load paths.

Required Values

A pathname, or a series of paths separated by semicolons. This parameter is optional and may be left blank.

Additional information

The default load paths are determined by the "bin" and "Shared Data" settings in the .egate.store file. See the *e*Gate Integrator System Administration and Operations Guide* for more information about this file.

To specify multiple directories, manually enter the directory names rather than selecting them with the "file selection" button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

Auxiliary Library Directories

Description

Specifies a path to auxiliary library directories. Any **.monk** files found within those directories will automatically be loaded into the e*Way's Monk environment. This parameter is optional and may be left blank.

Required Values

A pathname, or a series of paths separated by semicolons. This parameter is optional and may be left blank.

Additional information

To specify multiple directories, manually enter the directory names rather than selecting them with the "file selection" button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

Monk Environment Initialization File

Specifies a file that contains environment initialization functions, which will be loaded after the auxiliary library directories are loaded. Use this feature to initialize the e*Way's Monk environment (for example, to define Monk variables that are used by the e*Way's function scripts).

Required Values

A filename within the "load path", or filename plus path information (relative or absolute). If path information is specified, that path will be appended to the "load path." See **"Additional Path" on page 24** for more information about the "load path."

Additional information

Any environment-initialization functions called by this file accept no input, and must return a string. The e*Way will load this file and try to invoke a function of the same base name as the file name (for example, for a file named **my-init.monk**, the e*Way would attempt to execute the function **my-init**).

Typically, it is a good practice to initialize any global Monk variables that may be used by any other Monk Extension scripts.

The internal function that loads this file is called once when the e*Way first starts up (see **Figure 2 on page 18**).

Startup Function

Description

Specifies a Monk function that the e*Way will load and invoke upon startup or whenever the e*Way's configuration is reloaded. This function should be used to initialize the external system before data exchange starts.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is optional and may be left blank.

Additional information

The function accepts no input, and must return a string.

The string "FAILURE" indicates that the function failed; any other string (including a null string) indicates success.

This function will be called after the e*Way loads the specified "Monk Environment Initialization file" and any files within the specified **Auxiliary Directories**.

The e*Way will load this file and try to invoke a function of the same base name as the file name (see **Figure 2 on page 18**). For example, for a file named **my-startup.monk**, the e*Way would attempt to execute the function **my-startup**.

Process Outgoing Event Function

Description

Specifies the Monk function responsible for sending outgoing messages (Events) from the e*Way to the external system. This function is event-driven (unlike the **Exchange Data with External Function**, which is schedule-driven).

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. ***You may not leave this field blank.***

Additional Information

The function requires a non-null string as input (the outgoing Event to be sent) and must return a string.

The e*Way invokes this function when one of its Collaborations publishes an Event to an <EXTERNAL> destination (as specified within the Schema Designer). The function returns one of the following (see **Figure 7 on page 23** for more details):

- Null string: Indicates that the Event was published successfully to the external system.
- "RESEND": Indicates that the Event should be resent.
- "CONNERR": Indicates that there is a problem communicating with the external system.
- "DATAERR": Indicates that there is a problem with the message (Event) data itself.

If a string other than the above is returned, the e*Way will create an entry in the log file indicating that an attempt has been made to access an unsupported function.

Note: *If you wish to use* ***event-send-to-egate*** *to enqueue failed Events in a separate IQ, the e*Way must have an inbound Collaboration (with appropriate IQs) configured to process those Events. See* **event-send-to-egate** *on page 41 for more information.*

Exchange Data with External Function

Description

Specifies a Monk function that initiates the transmission of data from the external system to the e*Gate system and forwards that data as an inbound Event to one or more e*Gate Collaborations. This function is called according to a schedule (unlike the **Process Outgoing Message Function**, which is event-driven).

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is optional and may be left blank.

Additional Information

The function accepts no input and must return a string (see **Figure 5 on page 21** for more details):

- Null string: Indicates that the data exchange was completed successfully. No information will be sent into the e*Gate system.
- "CONNERR": Indicates that a problem with the connection to the external system has occurred.

- "DATAERR": Indicates that a problem with the data itself has occurred. The e*Way handles the string "DATAERR" and "DATAERR" plus additional data differently; see **Figure 5 on page 21** for more details.
- Any other string: The contents of the string are packaged as an inbound Event. The e*Way must have at least one Collaboration configured suitably to process the inbound Event, as well as any required IQs.

This function is initially triggered by the **Start Data Exchange** schedule or manually by the Monk function **start-schedule**. After the function has returned true and the data received by this function has been ACKed or NAKed (by the **Positive Acknowledgment Function** or **Negative Acknowledgment Function**, respectively), the e*Way checks the **Zero Wait Between Successful Exchanges** parameter. If this parameter is set to **Yes**, the e*Way will immediately call the **Exchange Data with External** function again; otherwise, the e*Way will not call the function until the next scheduled "start exchange" time or the schedule is manually invoked using the Monk function **start-schedule** (see **start-schedule** on page 39 for more information).

External Connection Establishment Function

Description

Specifies a Monk function that the e*Way will call when it has determined that the connection to the external system is down.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. ***This field cannot be left blank.***

Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.
- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Down Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Verification** function (see below) is called when the e*Way has determined that its connection to the external system is up.

External Connection Verification Function

Description

Specifies a Monk function that the e*Way will call when its internal variables show that the connection to the external system is up.

Required Values

The name of a Monk function. This function is optional; if no **External Connection Verification** function is specified, the e*Way will execute the **External Connection Establishment** function in its place.

Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.
- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Up Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Establishment** function (see above) is called when the e*Way has determined that its connection to the external system is down.

External Connection Shutdown Function

Description

Specifies a Monk function that the e*Way will call to shut down the connection to the e*Way.

Required Values

The name of a Monk function. This parameter is optional.

Additional Information

This function requires a string as input, and may return a string.

This function will only be invoked when the e*Way receives a "suspend" command from a Control Broker. When the "suspend" command is received, the e*Way will invoke this function, passing the string "SUSPEND_NOTIFICATION" as an argument.

Any return value indicates that the "suspend" command can proceed and that the connection to the external system can be broken immediately.

Positive Acknowledgment Function

Description

Specifies a Monk function that the e*Way will call when ***all*** the Collaborations to which the e*Way sent data have processed and enqueued that data successfully.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined.

Additional Information

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the Positive Acknowledgment function will be called again, with the same input data.
- Null string: The function completed execution successfully.

After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Positive Acknowledgment function (otherwise, the e*Way executes the Negative Acknowledgment function).

Negative Acknowledgment Function

Description

Specifies a Monk function that the e*Way will call when the e*Way fails to process and queue Events from the external system.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined.

Additional Information

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the function will be called again.
- Null string: The function completed execution successfully.

This function is only called during the processing of inbound Events. After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is not completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Negative Acknowledgment function (otherwise, the e*Way executes the Positive Acknowledgment function).

Shutdown Command Notification Function

Description

Specifies a Monk function that will be called when the e*Way receives a "shut down" command from the Control Broker.

Required Values

The name of a Monk function. This parameter is optional.

Additional Information

When the Control Broker issues a shutdown command to the e*Way, the e*Way will call this function with the string "SHUTDOWN_NOTIFICATION" passed as a parameter.

The function accepts a string as input and must return a string:

- A null string or "SUCCESS": Indicates that the shutdown can occur immediately.

- Any other string: Indicates that shutdown must be postponed. Once postponed, shutdown will not proceed until the Monk function **shutdown-request** is executed (see **shutdown-request** on page 41).

Note: *If you postpone a shutdown using this function, be sure to use the (**shutdown-request**) function to complete the process in a timely manner.*

3.2 External Configuration Requirements

There are no configuration changes required in the external system. All necessary configuration changes can be made within e*Gate.

Chapter 4

Implementation

This chapter describes information pertinent to implementing the CORBA-VisiBroker Client e*Way in a production environment, including procedures for using the CORBA VisiBroker Client Converter Build tool to create an Event Type Definition (**.ssc** file) from an IDL. A list of the supported IDL primitive types is also provided.

This Chapter Includes:

4.1 Overview

The CORBA-VisiBroker Client e*Way utilizes existing Monk capabilities to directly call CORBA object interfaces using extensions within Monk scripts. After loading the monk_corba DLL, Monk scripts can use the **corba-invoke** function to establish and communicate with CORBA objects. See **corba-invoke** on page 42.

4.1.1 Method Invocation

Invoking remote methods on a CORBA object is simplified through the use of Monk defined message structures (Event Type Definitions). The e*Way provides two components to implement simplified method invocation.

- CORBA VisiBroker Client Converter Build tool
- The Monk script **corba-struct-call.monk** (see **corba-struct-call** on page 43)

4.2 CORBA VisiBroker Client Converter Build Tool

The CORBA VisiBroker Client Converter Build tool (**stcir2ssc.exe**) can create an Event Type Definition (**.ssc**) file automatically by interrogating the VisiBroker Interface Repository. The nodes of the Event Type Definition are used to both represent details of the interface definition (method name, parameter names/data-types/direction) and to hold the argument values passed during invocation.

The Build tool can be run from the e*Gate ETD Editor, or directly from a command line or a command script.

4.2.1 Using the Build Tool

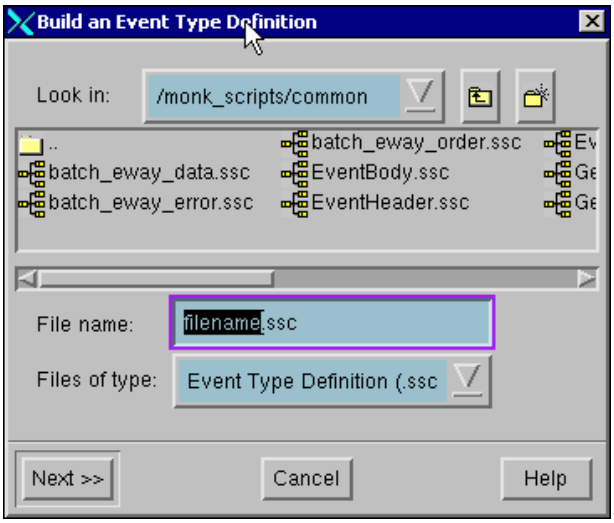
Use one of the following procedures to create an Event Type Definition from an IDL file.

To create an ETD using the Build tool from the GUI:

1. Launch the ETD Editor.
2. On the ETD Editor's Toolbar, click **Build**.

   The **Build an Event Type Definition** dialog box appears (see Figure 8).

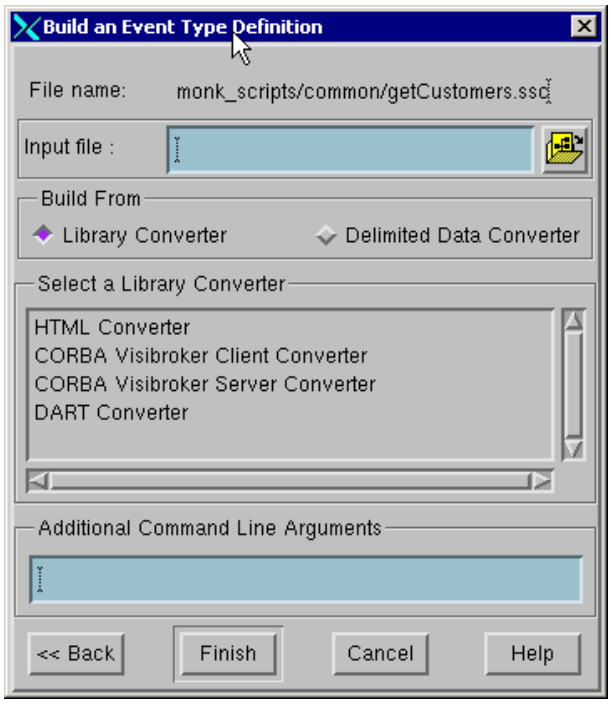
Figure 8 Build an Event Type Definition Dialog



3. In the **File name** field, type the name of the ETD file you wish to build. *Do not specify any file extension*—the Build tool supplies the **.ssc** extension automatically.
4. Click **Next**. A new dialog displays (see Figure 9). In the Input file field, type the name and path of the input file upon which you want to base the ETD or browse to the an existing file by clicking on the Browse button next to the Input file field.
5. Under **Build From**, select **Library Converter**.
6. Under **Select a Library Converter**, select **CORBA VisiBroker Client Converter**.
7. Click **Finish**.

   The Converter Wizard will launch.
8. Follow the Wizard's instructions to finish building the ETD file.

Figure 9 Build an Event Type Definition - CORBA VisiBroker Client Converter



To create an ETD using the Build tool from a command line:

From a command line enter:

```
stcir2ssc.exe VisiBroker_args outputSSCfileName repositoryName
```

where:

Visibroker_args represents any optional arguments required by your VisiBroker configuration, such as the Smart Agent port number. See the VisiBroker documentation for syntax and usage.

outputSSCfileName is the name of the file that will be written without the .ssc extension. If the file already exists it will be overwritten. (The Build tool adds the .ssc extension automatically.)

repositoryName is an optional argument to specify the name of the Interface Repository to use. If there is more than one repository on the network, the default VisiBroker behavior is to "round-robin" between all the repository servers running. We recommend that you use this parameter to avoid confusion since different repository objects may be started to make different IDLs available for different projects.

The Build tool reads the Interface Repository and generates a corresponding structure file suitable for use with **corba-struct-call**.

Example

1 Start the repository object for your IDL. For example,

```
irep -console AccountRepository account.idl
```

2 Run the Build tool to read the repository and write an .ssc file. The entire command should be typed on one line; it is illustrated wrapping here due to page size. For example,

```
stcir2ssc.exe monk_scripts\common\corba-account.ssc
AccountRepositry
```

4.3 CORBA-Visibroker Supported Types

The CORBA-VisiBroker Client e*Way supports the following IDL primitive types as IN, OUT and INOUT arguments to methods.

Table 2 IDL Primitive Types

IDL Type	VisiBroker Type	Monk Type
short	CORBA::Short	integer
long	CORBA::Long	integer
unsigned short	CORBA::UShort	integer
unsigned long	CORBA::ULong	integer
longlong		ASCII numeric string
float	CORBA::Float	real
double	CORBA::Double	real
char	CORBA::Char	character
octet		
boolean	CORBA::Boolean	boolean
string	CORBA::String	string
sequence		
struct		

In addition to these IDL primitive types, the e*Way supports object references and can handle CORBA and user-defined exceptions.

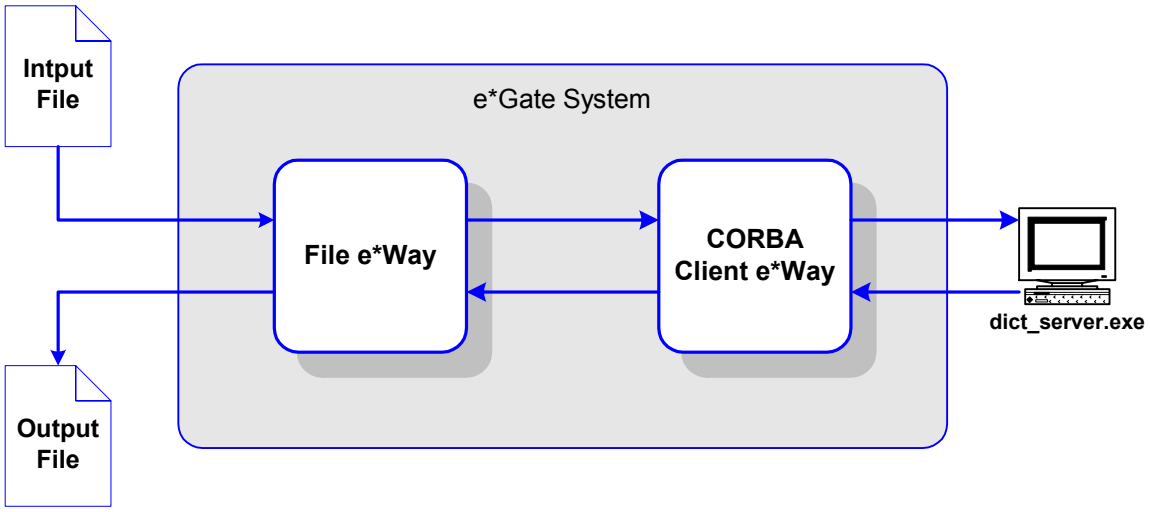
4.4 Installing the CORBA Client Sample Schema

The e*Gate Installation CD contains a sample scenario to demonstrate a simple Schema using the CORBA Client e*Way. The sample Schema can be installed on a local registry host for demonstration purposes.

4.4.1 CORBA Client Sample Schema Overview

The CORBA Client sample Schema uses a sample application (**dict_server.exe**) to process a request for a term and return a definition for that term. The sample Schema uses a file e*Way to load in the request (the word "network"). The CORBA Client e*Way forwards this request to the **dict_server** program. The sample external application returns the definition of the word to the CORBA Client e*Way. The file e*Way writes the definition of the term to a text file.

Figure 10 CORBA Client Sample Schema



4.4.2 Create the Sample Application

The sample application used in this schema is included with the **VisiBroker 3.3 for C++** from Borland.

To create the sample application

1. Install the **VisiBroker 3.3 for C++** application from Borland.

   Follow the installation instructions that accompany the VisiBroker application.
2. In Windows Explorer, navigate to the location where the VisiBroker examples were installed. The default location is **C:\Inprise\vbroker\examples**.
3. Open the folder containing the Smart Stub example: **C:\Inprise\vbroker\examples\sstub**.
4. Follow the instructions found in **sstub.html** to compile the sample code and generate the **dict_server.exe** sample executable.

4.4.3 Create Sample Data File

You must create a sample data file before running the schema.

To create a sample data file:

1. Create the file **CORBA_data_In.fin** in file d:**eGate\client\data\CORBA** directory (where **d:** is the letter drive of your eGate participating host), and create the contents, consisting of five words separated by a new line. The five words are:

```
distribute
object
network
application
infrastructure
```

2. Save the **CORBA_data_In.fin** file.

4.4.4 Import the Sample Schema

After you create a sample data file, you must import the sample schema.

To import the sample Schema:

1. Start e*Gate Schema Designer and create a new schema.
2. On the **File** menu, click **Import Definitions from File**.
3. On the **Import Wizard Introduction** window, click **Next**.
4. Click **Schema**, then click **Next**.
5. Specify the sample schema file name, and then click **Next**. The installation CD path name of this file should be **\samples\ewvbcorbaclient\ewvbcorbaclient.zip**.

4.4.5 Run the Sample Schema

Running the sample scenario is a two step process. The sample CORBA executable (**dict_server.exe**) must be started and the Control Broker must be started.

To start the CORBA executable:

1. From the command line navigate to the ***d:*\Inprise\vbroker\examples\sstub** directory (where ***d:*** is the drive letter for your Inprise Visibroker installation).
2. The the following at the command prompt:

```
start dict_server
```

The dict_server application will start up. (See Figure 11 below).

Figure 11 The **dict_server** Window

To start the Control Broker

From the command line, type the following command:

```
stccb -ln logical_name -rh registry -rs CORBA_Client -un user_name
-up password
```

where

logical_name is the logical name of the Control Broker,

registry is the name of the Registry Host, and

user_name and password are a valid e*Gate username/password combination.

To verify the results:

Use a text editor to view the output that was written to the output file (**CORBA_output0.dat**) in the ***d:*\eGate\client\data\CORBA** directory (where ***d:*** is the drive letter for your e*Gate participating host).

Note: *The configuration for the output file is **CORBA_output*%d.*dat**--the "**%d**" represents a counter or sequence number, as evidenced by the "**0**" in the file above.*

Chapter 5

Functions

The functions described in this chapter control the CORBA-VisiBroker Client e*Way's basic operations as well as those needed to interface with the CORBA server object.

5.1 Overview

The functions described in this section can only be used by the functions defined within the e*Way's configuration file. None of the functions are available to Collaboration Rules scripts executed by the e*Way.

The CORBA-Visibroker Client e*Way's functions fall into the following categories:

- **Basic Functions** on page 38
- **CORBA-VisiBroker Monk Functions** on page 42

5.2 Basic Functions

The functions in this category control the e*Way's most basic operations.

The basic functions are

start-schedule on page 39

stop-schedule on page 39

send-external-up on page 39

send-external-down on page 40

get-logical-name on page 40

event-send-to-egate on page 41

shutdown-request on page 41

start-schedule

Syntax

```
(start-schedule)
```

Description

start-schedule requests that the e*Way execute the "Exchange Data with External" function specified within the e*Way's configuration file. Does not effect any defined schedules.

Parameters

None.

Return Values

None.

Throws

None.

stop-schedule

Syntax

```
(stop-schedule)
```

Description

stop-schedule requests that the e*Way halt execution of the "Exchange Data with External" function specified within the e*Way's configuration file. Execution will be stopped when the e*Way concludes any open transaction. Does not effect any defined schedules, and does not halt the e*Way process itself.

Parameters

None.

Return Values

None.

Throws

None.

send-external-up

Syntax

```
(send-external-up)
```

Description

send-external-up instructs the e*Way that the connection to the external system is up.

Parameters

None.

Return Values

None.

Throws

None.

send-external-down

Syntax

```
(send-external-down)
```

Description

send-external-down instructs the e*Way that the connection to the external system is down.

Parameters

None.

Return Values

None.

Throws

None.

get-logical-name

Syntax

```
(get-logical-name)
```

Description

get-logical-name returns the logical name of the e*Way.

Parameters

None.

Return Values

string
Returns the name of the e*Way (as defined by the Schema Designer).

Throws

None.

event-send-to-egate

Syntax

```
(event-send-to-egate string)
```

Description

event-send-to-egate sends data that the e*Way has already received from the external system into the e*Gate system as an Event.

Parameters

Name	Type	Description
string	string	The data to be sent to the e*Gate system

Return Values

Boolean
Returns true (#t) if the data is sent successfully; otherwise, returns false (#f).

Throws

None.

Additional information

This function can be called by any e*Way function when it is necessary to send data to the e*Gate system in a blocking fashion.

shutdown-request

Syntax

```
(shutdown-request)
```

Description

shutdown request requests the e*Way to perform the shutdown procedure when there is no outstanding incoming/outgoing event. When the e*Way is ready to act on the shutdown request, in invokes the **Shutdown Command Notification Function** (see **"Shutdown Command Notification Function" on page 29**). Once this function is called, the shutdown proceeds immediately.

Parameters

None.

Return Values

None.

Throws

None.

5.3 CORBA-VisiBroker Monk Functions

The CORBA VisiBroker Monk functions are:

corba-invoke

Syntax

```
(corba-invoke IDL-name objectname methodname args-vector
exception-vector)
```

Description

corba-invoke invokes a CORBA object.

Parameters

Name	Type	Description
IDL-name	String	The name of the interface as registered in the interface repository, for example IDL:Account:1.0
objectname	String	An optional parameter used to specify a particular CORBA server object. For more information see "Additional information", below.
methodname	String	The name of the method in the server to be called
args-vector	Vector	A vector containing all the required arguments to the method and the return type expected. For more information see "Additional information", below.
exception-vector	Vector	A vector that describes the types of exceptions that may be thrown by the method.

Return Values

Upon return, the first element in the vector has the return type and each element that was "out" or "inout" has a value in the third element of that sub vector.

Throws

None.

Additional Information

If *objectname* is omitted, the VisiBroker ORB will search for any server object that matches the IDL-name. (VisiBroker's behavior is to "round-robin" requests between all instances of the IDL-name on the network.)

The first element in the *args-vector* vector is the string name of the return type. The remaining elements are one per argument. Each element is itself a vector comprising the input/output specifier, the parameter data type, parameter name and value:

```
#("type of return value" #("in"/"out"/
"inout" "name of parameter" value)...)
```

For example, to invoke a method declared in the IDL as:

```
long getOrderStatus (in string orderNum, inout string custName, out f
loat accountBalance)
```

args-vector would look like:

```
# ("long" # ("in" "string" "orderNum" "AB01234") # ("inout" "string"
"custName" "John Doe" # ("out" "float" "accountBalance"))
```

Examples

Here is an example of the use of this function:

```
(define args  #("float" #("in" "custom" "jay")))
(corba-invoke "IDL:Account:1.0" "balance" args)
```

By way of explanation, the following code example includes coding and comments (designated with ';') for a Monk script that calls two methods within a CORBA object:

```
;; Load the monk extension that provides an interface to corba
(load-extension "monk_corba.dll")

;; define the parameter vector
;; the first item is the return type. in this case float
;; the second is the first input parameter,
;; which is an "in" (input) parameter and is of type "string",
;; the name of the input parameter is "param1" and its
;; value is "value"
(define vec #( "float" #("in" "string" "param1" "value") ))

;; Here we actually make the call
;; the first parameter is the name of the object
;; the second is the name of the method that we are invoking
;; and finally the third is the vector that we created above.
(corba-invoke "IDL:Test:1.0" "test" vec)

;; display the return value which will be a float
(display (vector-ref vec 0))
```

The following example represents a more complicated usage.

```
;; Now an output parameter is the argument
;; the "out" indicates it is an out parameter
(define vec2 #( "string" #("out" "float" "pounds")))

(corba-invoke "IDL:Test:1.0" "price" vec2)

;; The output value will be in the vec2
;; it is item 3 of item 1 of the vector
(display (vector-ref 3 (vector-ref vec2 1)))
```

corba-struct-call

Syntax

```
(corba-struct-call method-path)
```

Description

corba-struct-call uses the specified *method-path* to generate a call to **corba-invoke** (see **corba-invoke** on page 42).

The Monk script **corba-struct-call.monk** provides a helper function for building a CORBA method invocation based on the information held in the generated Event Type Definition (ETD). Before invoking **corba-struct-call**, in and inout data nodes are primed with the required values. The CORBA method is then called by passing the *method-path* for the method name node to the **corba-struct-call** function. Upon return from a successful invocation, the values placed in the return value and in/inout data nodes can be retrieved.

Parameters

Name	Type	Description
method-path	String	A path reference to a node corresponding to a method in an ETD generated by the CORBA VisiBroker Client Converter Build tool.

Return Values

Upon return from a successful invokation, the return value holder and the OUT/INOUT parameter value holders are updated. The **corba-struct-call** expression evaluates to the expression value of the **corba-invoke** call.

Additional Information

The ETD contains nodes named "data" to hold the data values for method arguments, and may be used with the SeeBeyond Collaboration Rules Editor to graphically construct method invocations.

corba-struct-call-obj

Syntax

```
(corba-struct-call-obj method-path object-handle)
```

Description

corba-struct-call-obj provides a helper function for building a CORBA method invocation based on the information held in the generated ETD, functioning in the same manner as **corba-struct-call**. This function, however, takes an additional parameter which is an object reference returned from a previous method call as either a return value or an output parameter.

Parameters

Name	Type	Description
method-path	String	A path reference to a node corresponding to a method in an ETD generated by the CORBA VisiBroker Client Converter Build tool.
object-handle	String	An object reference returned from a previous method call as either a return value or an output parameter.

Return Values

Upon return from a successful invocation, the return value holder and the OUT/INOUT parameter value holders are updated. The **corba-struct-call-obj** expression evaluates to unspecified.

Additional Information

The ETD contains nodes named "data" to hold the data values for method arguments, and may be used with the SeeBeyond Collaboration Rules Editor to graphically construct method invocations.

Example

This function can be useful for inter-orb communication, for example, between the CORBA-Visibroker (Client) e*Way and an object written using Iona's Orbix. Given the Orbix server object's IOR string, a call to the object can be made as follows:

```
(define objHandle (string->corbaobject IORstring))
(corba-struct-call-obj method-name objHandle)
```

corba-struct-call-imp

Syntax

```
(corba-struct-call-imp method-path implementation-name)
```

Description

corba-struct-call-imp provides a helper function for building a CORBA method invocation based on the information held in the generated message structure, functioning in the same manner as **corba-struct-call**. This function, however, takes an additional parameter which is the implementation name of a server object. This may be used to invoke methods on the specified server object, instead of the default behavior of corba-struct-call which will locate any server object that implements the IDL.

Parameters

Name	Type	Description
method-path	String	A path reference to a node corresponding to a method in an ETD generated by the CORBA VisiBroker Client Converter Build tool.
implementation-name	string	The implementation name of the server object.

Return Values

Upon return from a successful invocation, the return value holder and the OUT/INOUT parameter value holders are updated. The **corba-struct-call-imp** expression evaluates to unspecified.

Additional Information

The ETD contains nodes named "data" to hold the data values for method arguments, and may be used with the SeeBeyond Collaboration Rules Editor to graphically construct method invocations.

Chapter 6

Troubleshooting

This chapter discusses troubleshooting for the CORBA VisiBroker Client e*Way. The range of customizations that you can apply to any e*Way makes it impossible for a single chapter to present an exhaustive list of everything that could possibly go wrong with any given e*Way. However, we can give you some guidelines to follow when troubleshooting any e*Way's operation or performance.

6.1 General Troubleshooting

In the initial stages of developing your e*Gate system, most problems with e*Ways can be traced to configuration.

In the Schema Designer:

- Does the e*Way have the correct Collaborations assigned?
- Do those Collaborations use the correct Collaboration Services?
- Is the logic correct within any Collaboration Rules employed by this e*Way's Collaborations?
- Do those Collaborations subscribe to and publish Events appropriately?
- Are all the components that "feed" this e*Way properly configured, and are they sending the appropriate Events correctly?
- Are all the components that this e*Way "feeds" properly configured, and are they subscribing to the appropriate Events correctly?

In the e*Way Editor:

- Check that *all* configuration options are set appropriately. Be sure that any settings that you changed are set correctly; that you have overlooked no required changes; and that any defaults are acceptable for your installation.

On the Participating Host supporting the e*Way:

- Check that the Participating Host is operating properly, and that it has sufficient disk space to hold the IQ data that this e*Way's Collaborations publish.

In the external application with which the e*Way communicates:

- Check that the application is configured correctly, is operating properly, and is sending or receiving the correct data appropriately.

- Check that the connection between the external application and the e*Way is functioning appropriately.

Once the e*Way is up and running properly, operational problems can be due to external influences (network or other connectivity problems), problems in the operating environment (low disk space or system errors), problems or changes in the data the e*Way is processing. There may also be corrections required to Collaboration Rules scripts that become evident in the course of normal operations.

One of the most important tools in the troubleshooter's arsenal is the e*Way log file. Please see the *e*Gate System Administration and Operations Guide* for an extensive discussion of log files, debugging options, and how to use the Schema Monitoring system to monitor both operations and performance.

6.2 Password Problems

Problem

The e*Way's configuration file contains the correct password, but the remote server reports that the password is incorrect.

Solution

The e*Way Editor encrypts the password based upon the entry in the "user name" configuration field; this problem can occur if the correct password is entered before the user name is entered.

1. In the Schema Designer, display the e*Way's properties.
2. Under **Configuration file**, click **Edit**.
3. The e*Way Editor will launch. Use the parameter- and section-selection controls to navigate to the user-name and password options.
4. Make sure that the correct user name is entered (re-enter it if necessary).
5. Enter the password, replacing any entry that may already exist.
6. Save the configuration file, and exit the e*Way Editor.
7. When you return to the e*Way's property sheet in the Schema Designer, click **OK** to apply the changes and close the property sheet.
8. Restart the e*Way if necessary.

6.3 Operating System Problems

Problem

When running the CORBA-VisiBroker e*Way on Solaris, a message similar to the following appears:

```
The referenced symbol _RT_CLASS_xyz_vtbl cannot be found
```

Additionally, the dynamic link editor **ld.so.1** fails to load the required library, probably due to differences in symbol table format between operating system versions. This is despite the directory **/usr/lib** already being in LD_LIBRARY_PATH.

Solution

Rename the executable file for the e*Way and replace it with a wrapper script that exports the environment variable

```
LD_PRELOAD=/usr/lib/libC.so.5
```

and then calls the renamed executable.

If this does not solve the problem, load the Solaris patch that updates the file **/usr/lib/libC.so.5**.

For more information, see the Solaris documentation on **ld.so.1**.

The problem may occur on some versions of Solaris but is dependent on the version and patches installed. This problem does not occur on other operating systems.

Index

S

U

Z