SeeBeyond ICAN Suite

e*Way Intelligent Adapter for CICS User's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

Java Version

Contents

Chapter 1

Introduction

This chapter includes a brief description of IBM's Customer Information Control System™ (CICS™), an overview of the SeeBeyond e*Way Intelligent Adapter for CICS, as well as system requirements for using the CICS e*Way.

1.1 Intended Reader

The reader of this guide is presumed to be a developer or system administrator with responsibility for maintaining the e*Gate system, to have a working knowledge of Windows operations and administration, and to be thoroughly familiar with CICS.

1.2 Overview

CICS Transaction Server

IBM's Customer Information Control System (CICS), is IBM's most widely used proprietary, transaction monitor. CICS provides connectivity and online transaction management for mission-critical applications. It supports real-time distributed processing environments and online transaction processing (OLTP). According to IBM, CICS handles more than thirty billion transactions, processing more than one trillion dollars, per day.

CICS is the premier OLTP (On-Line Transaction Processing) product from IBM. It is used to access many file systems and databases including third party products. For IBM product, it interfaces with DB2, VSAM and IMS/DB. For non-IBM products, it interfaces with IDMS, ADABAS, DATACOM, to name a few. Most applications in CICS are written in COBOL, although it supports other languages such as PL/1.

OLTP systems provide accurate, up-to-date information within seconds, from terminals that give direct access to data held as either files or databases. CICS provides a company with numerous transaction processing and resource management functions, allowing the user to concentrate on developing application programs that meet that organization's specific business needs. CICS controls OLTP application programs in a distributed transaction processing (DTP) environment. CICS handles interactions between the terminal user and the application programs. Programs gain access to the CICS facilities with straightforward, high-level commands.

CICS provides:

- Communicationfunctions to terminals and systems required by application programs
- Control of concurrently running programs serving online users
- Facilities for accessing databases and files
- The ability to communicate with other CICS family members using Transmission Control Protocol/Internet Protocol (TCP/IP)
- Interactive facilities to configure specific systems
- Recovery processing and data protection, should a problem occur

The e*Way Intelligent Adapter for CICS

The e*Way Intelligent Adapter for CICS is an interface that enables remote bidirectional calls to CICS transactional programs. The CICS e*Way includes a build tool, the Cobol Copybook Converter, that creates an Event Type Definition (ETD) from a Cobol Copybook file and generates e*Gate Event Type Definitions (ETDs) for use within the e*Gate environment. The Copybook file structures are passed into the CICS environment as the data buffer (Commarea). The ETD files (.ssc) are converted into .xsc files that are compatible with the Java Collaboration Editor.

A fixed Event Type Definition, the CICSClient ETD (cicsclient.xsc), designed to expose various essential portions of the CICS Java API, provides available methods and properties, as well as access to all message attributes.

The e*Way enables both Synchronous and Asynchronous CICS program call handling using the CICSClient ETD.

The e*Way uses either the IBM CICS Transaction Gateway (**IBM CICS Transaction Gateway (CTG)** on page 10), or the SeeBeyond CICS Listener supported by the Java version of the CICS e*Way (**SeeBeyond CICS Listener (STCL)** on page 11), as the underlying connection transport for accessing z/OS CICS transactions:

IBM CICS Transaction Gateway (CTG)

CTG provides an API (the External Call Interface or ECI) to call CICS transactions on the mainframe. The ECI allows a non-CICS application program to call a CICS program in a CICS server. SeeBeyond's CICS e*Way uses this ECI method to connect to CICS. The CICS e*Way connects to CICS with CTG running on a local-host (Figure 2), on a second computer (Figure 1), or on the mainframe (**Figure 3 on page 11**).

Figure 1 e*Gate and CTG running on the same host

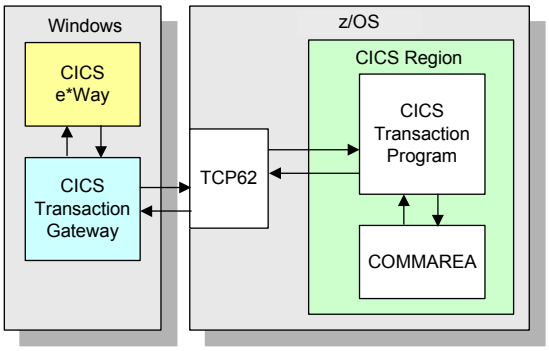


Figure 2 Remote connection with CTG on a UNIX or Windows host

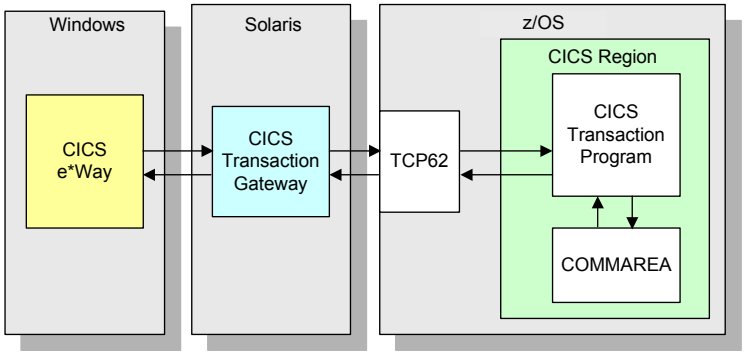
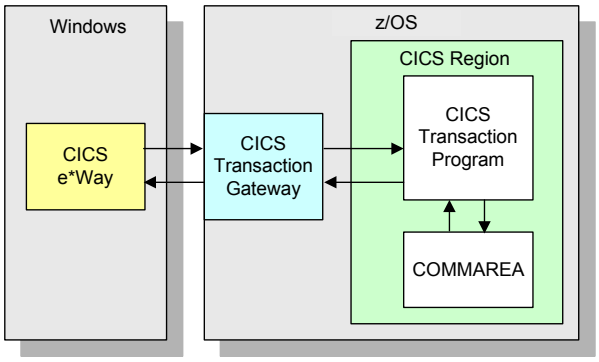


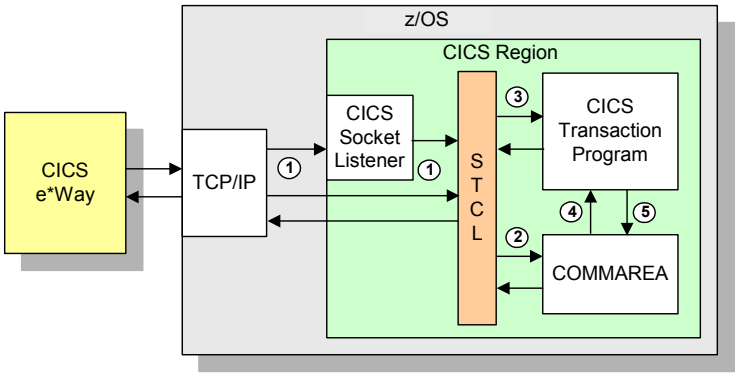
Figure 3 e*Gate connects with CTG running on the mainframe



SeeBeyond CICS Listener (STCL)

The CICS e*Way, running on a Windows 2000 or UNIX platforms connects to the IBM CICS Listener running on z/OS via the TCP/IP Sockets. The Listener accepts the incoming request and spawns a new process handing the socket connection off to the newly created process via TCP/IP **givesocket( )/takesocket( )** function calls. The spawned process invokes the user written CICS application program through an EXEC CICS LINK. The available methods in this version of the CICS e*Way are identical to the methods available when using the CICS Transaction Gateway, and provide compatibility with existing schemas. This provides the user with a means of accessing z/OS CICS transactions through TCP/IP without purchasing IBM's CICS Transaction Gateway product. The SeeBeyond CICS Listener is only available with the Java version of the CICS e*Way.

Figure 4 Using the SeeBeyond CICS Listener for Synchronous Transactions



The CICS e*Way (Java version) communicates with SeeBeyond CICS Listener for Synchronous Transactions (See Figure 4) as follows:

1. An incoming Connect request is handled by the IBM CICS Socket Listener, which starts the SeeBeyond CICS Listener Transaction STCL and hands off the incoming connection via the IBM TCP/IP Give Socket and Take Socket interface.
2. The SeeBeyond CICS Listener allocates a CICS COMMAREA and copies information from the CICS e*Way COMMAREA to the actual CICS COMMAREA.
3. The SeeBeyond CICS Listener issues an EXEC CICS LINK to requested CICS Transaction Program passing it the newly allocated COMMAREA.
4. The requested CICS Transaction obtains data from the COMMAREA, performs typical business rule processing and then returns its results in the COMMAREA and returns control back to the SeeBeyond CICS Listener.
5. The SeeBeyond CICS Listener copies information from the CICS COMMAREA back to the CICS e*Way COMMAREA.
6. The SeeBeyond CICS Listener goes into a listen mode and waits for the next incoming Transaction Program request.

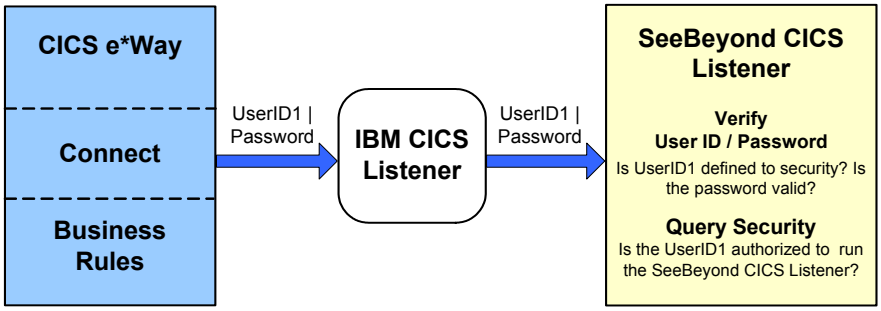
   The process continues until the SeeBeyond CICS Listener Timeout is exceeded or a disconnect request is received from the CICS e*Way.

1.2.1. z/OS CICS Security Considerations

Security Considerations for SeeBeyond CICS Listener

The CICS e*Way, using the SeeBeyond CICS Listener as the underlying connection transport, utilizes three modes of security with z/OS: Connection Logic, Request Link to Program, and Request Start Transaction. The userID and password are defined in the e*Way Connection configuration file. The connection manager uses the userID and password in the configuration file to start the SeeBeyond CICS Listener on z/OS. During Business Rules processing, requests that flow into the SeeBeyond CICS Listener can use the userID and password from the configuration file, or can be overwritten in the Collaborations.

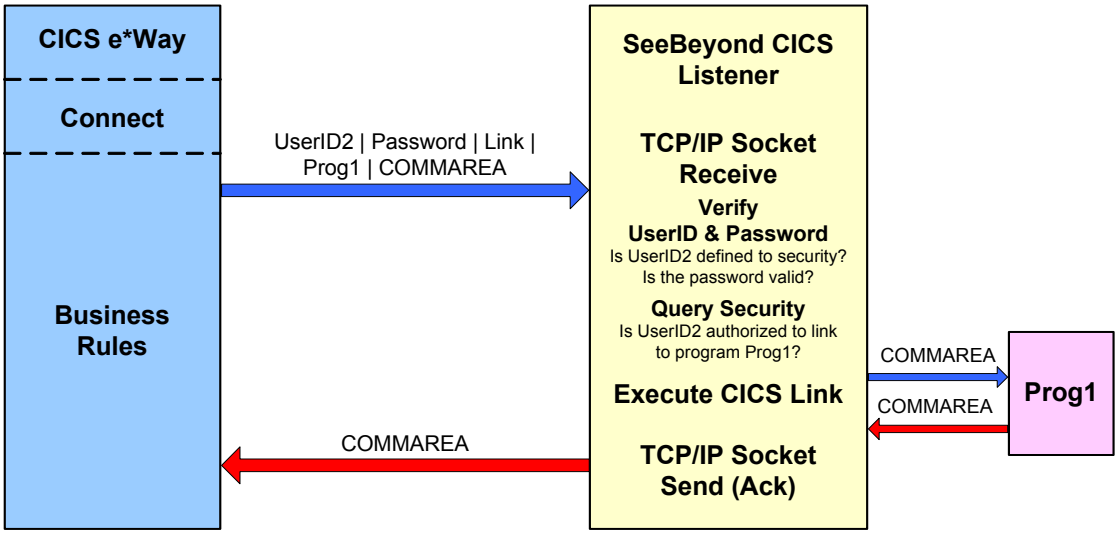
Figure 5 Connection Logic



Connection Logic

For the Connection Logic mode (Figure 5), the userID and password, passed from the CICS e*Way through the IBM CICS listener and into the SeeBeyond CICS Listener, must be defined for the z/OS security system (RACF, for example). The userID must be authorized by the z/OS security system to run CICS transaction "xxxx" inside of CICS. The default value for "xxxx" is STCL, and can be changed in the configuration of the Connection Manager in the CICS e*Way.

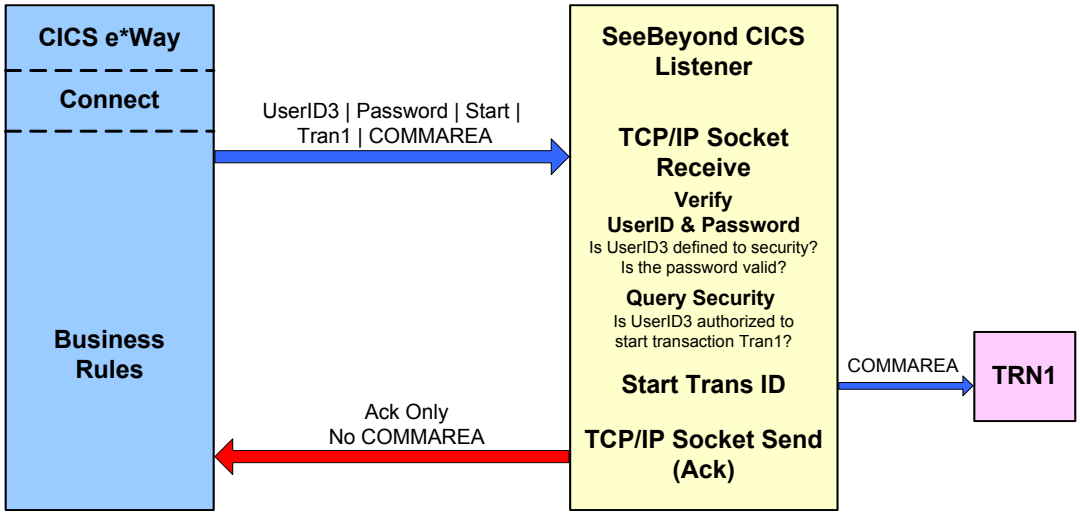
Figure 6 Business Rules Request to Program



Request Link to Program

For the Request Link to Program mode (Figure 6), the userID and password passed from the CICS e*Way to the SeeBeyond CICS Listener must be defined for the z/OS security system (RACF, for example). The userID must be authorized by the z/OS security system to run CICS program "prog1" inside of CICS. The default value for "prog1" is set in the configuration of the CICS e*Way, and can be overridden in the Collaboration for each request sent into the SeeBeyond CICS Listener.

Figure 7 Business Rules Request Start Transaction



Request Start Transaction

For the Request Start Transaction mode (Figure 7), the userID and password passed from the CICS e*Way to the SeeBeyond CICS Listener must be defined for the z/OS security system (RACF, for example). The userID must be authorized by the z/OS security system to start CICS transaction "TRN1" inside of CICS. The default value for "TRN1" is set in the configuration of the CICS e*Way, and can be overridden in the Collaboration for each request sent into the SeeBeyond CICS listener.

Security Considerations for IBM CICS Transaction Gateway

Security validation is not supported for IBM CICS Transaction Gateway 4.0.

For information on CICS Transaction Gateway 5.0 security validation refer to the following:

- *Readme.txt* for CTG 5.0 provided on the CTG 5.0 installation CD_ROM.
- *APAR II12217* and *APAR OW55570.*
- The *CICS Transaction Gateway Administration Guide* for your specific operating system, provided on the CICS Transaction Gateway Installation CD_ROM.

1.3 Supported Operating System

The CICS e*Way is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- HP-UX 11.0 and 11i (PA-RISC)
- IBM AIX 5.1L and 5.2
- IBM z/OS V1.3 and V1.4
- Sun Solaris 8 and 9
- Japanese Windows 2000, Windows XP, and Windows Server 2003
- Japanese HP-UX 11.0 and 11i (PA-RISC)
- Japanese Sun Solaris 8 and 9
- Korean Windows 2000, Windows XP, and Windows Server 2003
- Korean HP-UX 11.0 and 11i (PA-RISC)

Important: *Open and review the **Readme.txt** for the CICS e*Way for any additional information or requirements, prior to installation. The Readme.txt is located on the Installation CD_ROM at setup\addons\ewcics.*

Note: *AIX 5.1, and HP-UX 11i are not supported by CICS Transaction Gateway 4.0.*

*When using HP-UX with CICS Transaction Gateway 4.0, append the following path to the SHLIB PATH: SHLIB_PATH=$(SHLIB_PATH}:<e*Gate>/client/ThirdParty/IBMctg/lib/Hpux32. ("e*Gate" denotes where e*Gate has been installed.)*

1.4 System Requirements

To use the CICS e*Way, you need the following:

- An e*Gate Participating Host.
- A TCP/IP network connection.
- Additional disk space for e*Way executable, configuration, library, and script files. The disk space is required on both the Participating and the Registry Host. Additional disk space is required to process and queue the data that this e*Way processes. The amount necessary varies based on the type and size of the data being processed and any external applications performing the processing.
- Open and review the **Readme.txt** for the CICS e*Way regarding any additional requirements prior to installation. The Readme.txt is located on the Installation CD_ROM at setup\addons\ewcics.

1.5 External System Requirements

The CICS e*Way connects to CICS using either the IBM CICS Transaction Gateway (CTG) or the SeeBeyond CICS Listener. External requirements depend on which is implemented.

1.5.1. CICS Server Requirements for the IBM CICS Transaction Gateway Implementation

To enable the e*Way to communicate correctly with CICS using CICS Transaction Gateway (CTG) the following are required:

- CICS Transaction Server V1.3 or greater.
- IBM CICS Transaction Gateway version 4.0 with the application of APAR PQ57730, or CICS Transaction Gateway 5.0.
- The CICS e*Way running on z/OS, requires that the CTG files, **libCTGJNI.so** and **libCTGJNI_g.so** be copied to the egate/client/bin directory. These files are located on the computer in the same directory where CTG is installed:

```
../usr/lpp/ctg50(or 40)/ctg/bin/..
```

z/OS Configuration Requirements for the CICS Server and CICS Transaction Gateway

For detailed directions on configuring z/OS to connect via TCP62, see the CICS Transaction Gateway, Client Administration manual for your specific platform. These details are found in the chapter "Setting Up Client/Server Communications."

The summarized requirements are as follows:

- Install any of the VTAM AnyNet® releases.
- Install a TCP major node, which defines the AnyNet interface between TCP/IP and VTAM. For further information about this, see the IBM's *Guide to SNA over TCP/IP*.
- Install a CDRSC major node, which defines the remote Client device and instructs VTAM to route any session requests through the TCP/IP Physical Unit (ALSLIST).
- Check that the Physical Unit (PU) for the AnyNet interface is active.
- On CICS, you must define an APPC connection to the client workstation. (The connection can be statically defined, or autoinstalled.)
- Add an entry to the VTAM logon mode (LOGMODE) table for the modename specified on the SESSIONS definition. This entry specifies the class of service required for the group of sessions.

1.5.2. CICS Server Requirements for the SeeBeyond CICS Listener Implementation.

To enable the e*Way to communicate correctly with CICS using the SeeBeyond CICS Listener the following are required:

- z/OS 1.3, or 1.4 (see **External System Requirements** on page 16).
- Resource Access Control Facility (RACF) or an equivalent security product.
- CICS Transaction Server V1.3 or greater.
- IBM MVS TCP/IP socket runtime libraries, installed and configured for each CICS region in which the SeeBeyond CICS Listener will be run. For more information please refer to IBM's TCP/IP V3R2 for MVS: CICS TCP/IP Socket Interface Guide.
- COBOL for the z/OS and Language Environment.

1.5.3. CICS Listener Requirements for Invoking DB2 Applications

There are two different techniques that c an be used for invoking DB2 application programs through the SeeBeyond CICS Listener.

- RCT entry for the STCL SeeBeyond CICS Listener Transaction
- PPT entry to redirect DB2 application to another AOR (Application Owning Region)

For detailed information refer to **CICS Listener Considerations for Invoking DB2 Applications** on page 41.

Chapter 2

Installation

This chapter contains installation information for the CICS e*Way, SeeBeyond CICS Listener, and IBM CICS Transaction Gateway.

Chapter Topics

Important: *Open and review the **Readme.txt** for the CICS e*Way for any additional information or requirements, prior to installation. The Readme.txt is located on the Installation CD_ROM at setup\addons\ewcics.*

2.1 Windows Installation

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 CICS e*Way on a Windows system

1. Log in as an Administrator to the workstation on which you are installing the e*Way.
2. Insert the e*Way installation CD-ROM into the CD-ROM drive.
3. If the CD-ROM drive's Autorun feature is enabled, the setup application launches automatically; skip ahead to step 4. Otherwise, use the 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 **Select Sub-components** dialog box appears.
10. Select the **CICS 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 Java-enabled CICS 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.***

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, 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 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 privileges to create files in the e*Gate directory tree.

2.2.2. Installation Procedure

To install the CICS 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/setup
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. After installation is complete, exit the installation utility and launch the Schema Designer.

Important: *For HP-UX systems with CICS Transaction Gateway 4.0 append the following path to the SHLIB PATH: SHLIB_PATH=$(SHLIB_PATH}:<e*Gate>/client/ThirdParty/IBMctg/lib/Hpux32. (*"e*Gate" is where e*Gate has been 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, 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 Java-enabled CICS e*Way installation process will install the following files, see Table 1, within the e*Gate directory tree. Files are installed within the **egate\client** tree on the participating host and committed to the **default** schema on the Registry Host.

Table 1 Files Created by the Installation

e*Gate Directory	File(s)
\classes\	stccics.jar stcutil.jar
\configs\cicsclient\	cicsclient.def
etd	cics.ctl
\etd\cicsclient	cicsclient.xsc
\ThirdParty\ibmctg\classes	ctgclient.jar ctgserver.jar
\ThirdParty\gnu-getopt\classes	gnu-getopt.jar

2.4 SeeBeyond CICS Listener Installation for z/OS

The following section provides directions for installing the SeeBeyond CICS Listener to a z/OS operating system from the installation CD-ROM or from a 3480 Tape. Directions are also included for installing CICS CEDA definitions, adding the CICS e*Way load modules to the CICS DFHRPL concatenation, and using the SeeBeyond CICS Listener monitoring screen to verify that all components are properly installed and working correctly.

2.4.1. Installing the SeeBeyond CICS Listener from CD z/OS

These instructions show how to restore the SeeBeyond CICS Listener files from the installation CD_ROM (setup\addons\ewcics\SBYN_Listner.zip) to your MVS system in a usable state. The files are packaged on MVS for transfer using the TSO transmit (XMIT) command to transmit them into a data set. This is done to turn a PDS into FB 80 files which can be sent by FTP. These files are downloaded to a PC and then compressed with PKZIP.

1. Download the **SBYN_Listener.zip file** to your PC.
2. Unzip the files using WinZip or the Zip program of your choice.
3. Create two MVS datasets to receive the files, as follows:

```
//DD1     DD DSN=USER.XMIT.CICSLOAD,
//           DISP=(NEW,CATLG,DELETE),
//           RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS,
//           SPACE=(3120,(48,5)),
//           UNIT=diskunit
```

```
//DD2     DD DSN=USER.XMIT.JCLLIB,
//           DISP=(NEW,CATLG,DELETE),
//           RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS,
//           SPACE=(3120,(30,5)),
//           UNIT=diskunit
```

4 Upload (FTP, IND$FILE) the unzipped files to MVS using a binary file transfer method (no CRLF or ASCII translation).

5 Restore the files to PDS by using the **Receive** command on MVS.

6 Issue command: TSO RECEIVE INDATASET(uploaded.dataset)

7 When prompted by the message:

```
INMR906A Enter restore parameters or 'DELETE' or 'END' +
```

enter:

```
DA(name.of.your.library) UNIT(unit) VOLUME(volume)
```

Note: *The UNIT() and VOLUME() operands are optional but shown in case your installation requires them.*

We suggest using the following names for your received datasets:

```
STC.XMIT.CICS.CICSLOAD
STC.XMIT.CICS.JCLLIB
```

These names match our documentation, but you may change them as your facility requires.

2.4.2. Installing the SeeBeyond CICS Listener from 3480 Tape

The SeeBeyond CICS Listener installation for z/OS is provided on an installation tape containing the following datasets (Table 2):

Table 2 z/OS Datasets

Dataset Name	Contents
TAPE.STC.RESTORE.JCL	Physical sequential datasets containing the JCL for this tape.
TAPE.STC.CICS.JCLLIB	Partition dataset that contains installation jobs and control cards for the CICS e*Way.
TAPE.STC.CICS.CICLOAD	Load library that contains the CICS load modules for the CICS e*Way.

Copying the Tape Contents to Disk

1 Create and submit the following job to copy the RESTORE JCL to disk:

```
//JOB CARD
//IEBGENER EXEC PGM-IEBGENER
//*
```

```
//*COPY RESTORE JCL TO DISK
//*
//SYSUT1 DD DSN=TAPE.STC.RESTORE.JCL,DISP=OLD,UNIT=TAPE,
// VOL=(,RETAIN,SER=STC390),LABEL=(1,SL)
//SYSUT2 DD DSN=customers.pds(restore),DISP=SHR
//SYSIN DD DUMMY
//
```

2 Customize and submit the RESTORE job to copy the entire contents of the Installation tape to disk.

2.4.3. Installing the CICS CEDA Definitions

Customize the file definition in the JCLLIB member CEDALCFG to conform to your site's file naming conventions. Customize and submit job STCLCEDA to create CICS CEDA definitions for the CICS e*Way.

2.4.4. Adding the CICS e*Way Load Modules to CICS DFHRPL Concatenation

Add the following data set to the DFHRPL concatenation under CICS:

```
//   DD DSN=&PREFIX..STC.CICS.CICSLOAD,DISP=SHR
```

2.4.5. Creating the STCCLCFG Configuration File

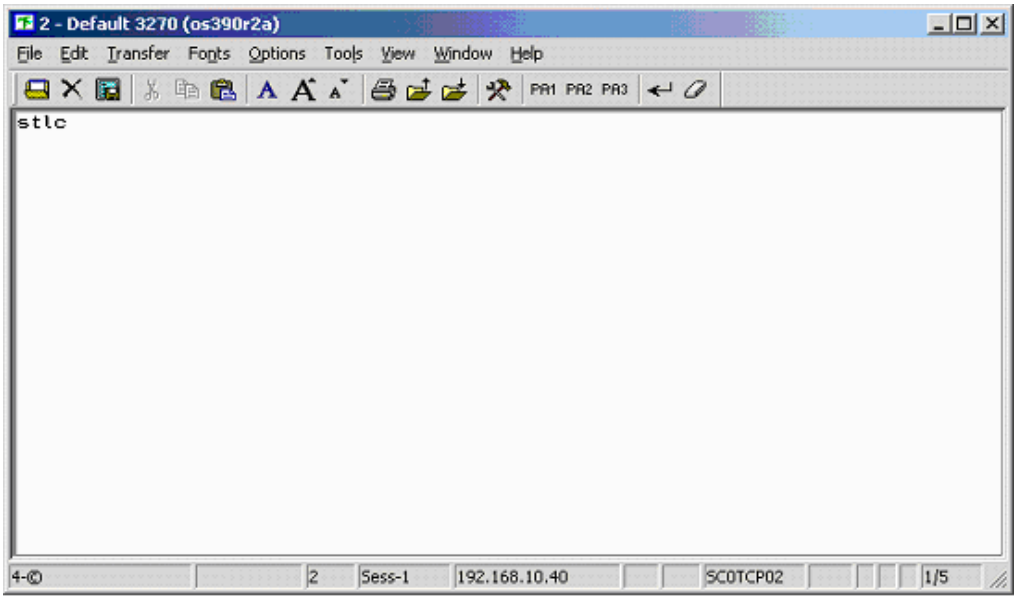
Customize and submit the **STCCLCFG** job found in **JCLLIB**. Make sure the FCT entry **STCCLCFG** is closed and disabled in CICS before running this job.

2.4.6. SeeBeyond CICS Listener Configuration File Maintenance Screen for z/OS

After the STCCLCFG Configuration File has been created and the CEDA file definition has been installed, use the SeeBeyond CICS Configuration File Maintenance screen to view and update the contents of the STCCLCFG configuration file.

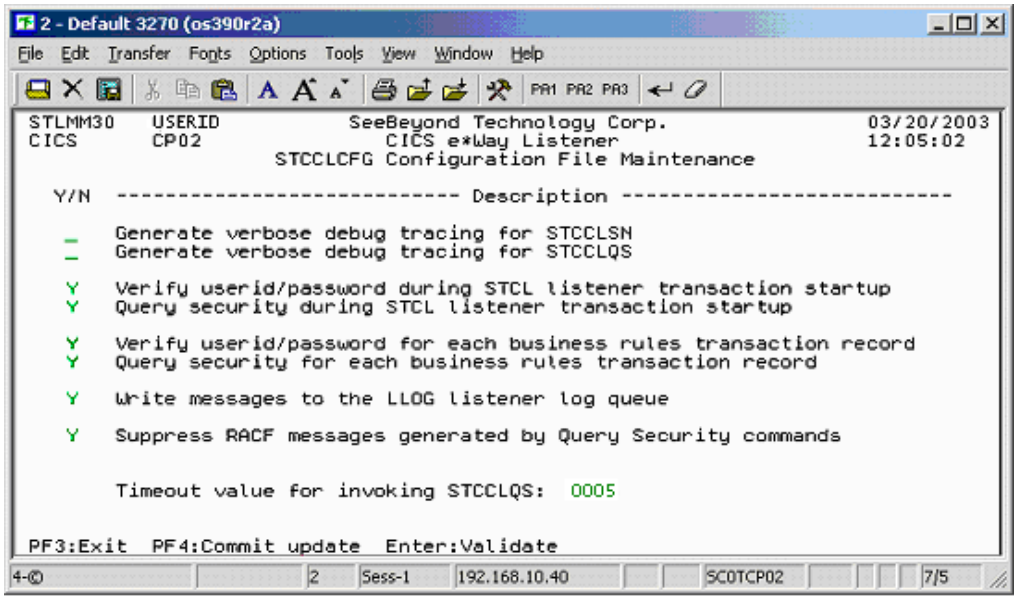
1 Logon to the CICS region that the SeeBeyond Listener is running in. Clear the screen, type in **stlc** as shown in Figure 8, and press **Enter**.

Figure 8



2 The **Configuration File Maintenance** screen appears as displayed in Figure 9.

Figure 9



3 The fields of the Listener Configuration Maintenance screen are described as follows:

- **Generate verbose debug tracing for STCCLSN**:This flag controls the logging of debug tracing messages issued by the STCCLSN SeeBeyond CICS Listener Program. These tracing messages are written to the LLOG transient data queue.

  If the "**Write messages to the LLOG listener log queue**" field on this screen is turned off, then debug tracing will NOT be logged.

  - **Y**: yes, log debug tracing messages.
  - **N or blank**: no, do not log debug tracing messages.
- **Generate verbose debug tracing for STCCLQS**:This flag controls the logging of debug tracing messages issued by the STCCLQS SeeBeyond CICS Listener Query Security Program. These tracing messages are written to the LLOG transient data queue.

  If the "**Write messages to the LLOG listener log queue**" field on this screen is turned off, then debug tracing will NOT be logged.

  - **Y**: yes, log debug tracing messages.
  - **N or blank**: no, do not log debug tracing messages.
- **Verify userid/password during STCL listener transaction startup:**This flag controls userid/password verification at the STCL SeeBeyond Listener Transaction startup. The userid and password are passed in from the CICS e*Way to the IBM Listener program in the user area of Connection Request Record. The IBM Listener passes this user area to the SeeBeyond CICS Listener when it spawns it as a child listener task.

  If this flag is turned on, the SeeBeyond Listener Program startup logic issues a Verify Password command to check whether the userid is defined for the RACF or any equivalent security system, and if the password is valid and current.

  - **Y**: yes, verify userid/password.
  - **N or blank**: no, do not verify userid/password.
- **Query Security during STCL listener transaction startup**: This flag controls Query Security processing at the STCL SeeBeyond Listener Transaction startup. The userid and password are passed in from the CICS e*Way to the IBM Listener program in the user area of Connection Request Record. The IBM Listener passes this user area to the SeeBeyond CICS Listener when it spawns it as a child listener task.

  If this flag is turned on, the SeeBeyond Listener Program startup logic invokes the STLQ SeeBeyond Listener Query Security transaction to check if the userid is authorized to run the STCL SeeBeyond Listener transaction.

  - **Y**: yes, perform Query Security processing.
  - **N or blank**: no, do not Query Security processing

- **Verify userid/password for each business rules transaction record**: This flag controls userid/password verification for each Business Rules transaction sent in from the CICS e*Way. The userid and password are passed in from the CICS e*Way in the Application Control Record (ACR) that contains the Business Rules transaction data.

  If this flag is turned on, the SeeBeyond CICS Listener program Business Rules logic issues a Verify Password command to check whether the userid is defined for the RACF or any equivalent security system, and if the password is valid and current.

  - **Y**: yes, verify userid/password.
  - **N or blank**: no, do not verify userid/password.

- **Query Security for each business rules transaction record**: This flag controls Query Security processing for each Business Rules transaction sent in from the CICS e*Way. The userid and password are passed in from the CICS e*Way in the Application Control Record (ACR) that contains the Business Rules transaction data.

  If this flag is turned on, the SeeBeyond CICS Listener Program Business Rules logic invokes the STLQ SeeBeyond Listener Query Security transaction to check whether the userid is authorized to run the requested customer application program or transaction that is specified in the ACR.

  - **Y**: yes, perform Query Security processing.
  - **N or blank**: no, do not Query Security processing.

- **Write messages to the LLOG listener log queue**: This flag controls the logging of messages to the LLOG listener log queue.

  If this flag is turned off, NO messages (normal information as well as debug tracing) will be written by either the STCCLSN (SeeBeyond CICS Listener) or STCCLQS (SeeBeyond Listener Query Security) programs.

  - **Y**: yes, write log messages to the LLOG listener log queue.
  - **N or blank**: no, do not write log messages to the LLOG listener log queue.

- **Suppress RACF messages generated by Query Security commands**: This flag controls RACF informational message logging for Query Security exceptions.

  If this flag is turned off, then every Query Security command that results in a negative result for Control, Alter, Update, or Read will cause respective RACF information messages to be written to the system message log. This could create unnecessary, high volume logging on the system message log files.

  It is recommended that this flag be turned off for debugging or low volume testing purposes only.

  In a production environment, it is recommended that this flag be turned on to

suppress the RACF informational messages generated by Query Security exception conditions.

- Y: yes, suppress RACF messages generated by Query Security processing.
- N or blank: no, do not suppress (in other words, allow) RACF messages to be generated by Query Security processing.

- **Timeout value for invoking STCCLQS**: This value is used to control the amount of time the STCCLSN SeeBeyond CICS Listener program waits for the STLQ Listener Query Security Transaction to return a response. Valid values are **0000** to **9999**. Since the STLQ transaction, under normal CICS processing conditions, executes within one second, it is recommended that the value be set above 1 and below 10. However, if the CICS region that the SeeBeyond CICS Listener is running in experiences occasional performance bottle necks, you may choose to set this value higher to avoid unnecessary STLQ timeout errors during peek load conditions.

  This value does not affect the time STLQ takes to respond. It simply sets a timeout threshold above which the STCCLSN stops waiting for a response from STLQ and issues an error message back to the CICS e*Way indicating that an STLQ timeout has occurred. For example, if this timeout value is set to 5, and STLQ executes in 1 second, then STCCLSN will "wake up" after 1 second, not the full 5 seconds.

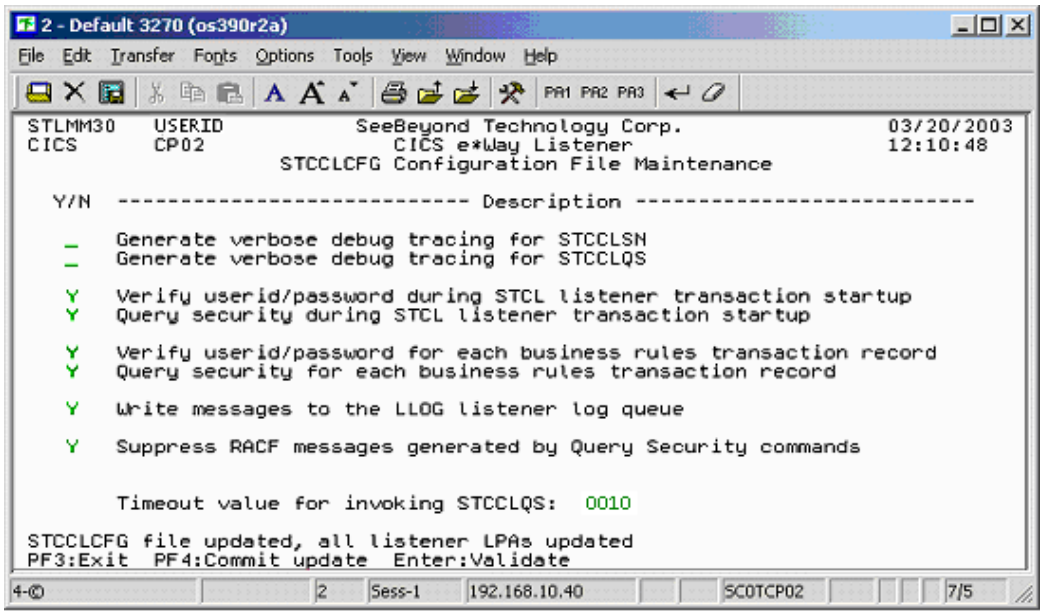
4 The pfkeys for this screen are as follows:

- **PF3**: exit.
- **PF4**: commit the updates on the screen to the STCCLCFG file and update all existing Listener Program Areas (LPA's) in CICS storage.
- **Enter**: validates the onscreen data for errors.

The user may make any necessary changes, then use the **Enter** key to validate any new values that have been entered. IF no error conditions occur for the data on the screen, then the user must press the PF4 key to commit the updates to the STCCLCFG Listener Config VSAM File. The updated screen appears as displayed in **Figure 10 on page 28**. A message appears at the bottom of the screen that indicating the file and whether any LPA's were updated.

During commit processing, the program will also automatically propagate the new data to all existing SeeBeyond Listener Program Areas (LPA's) in the same CICS region. There is one LPA for each instance of the SeeBeyond CICS Listener in the region. The STCCLSN (SeeBeyond CICS Listener) program uses its copy of the LPA to store monitoring statistics and to retrieve data used to control the execution of the SeeBeyond CICS Listener. Live updates to the LPAs provide the capability for real time control of the configuration flags and timers for all active SeeBeyond CICS Listeners executing at that time.

Figure 10



5 Continue making any necessary changes, committing them by pressing **PF4**. To exit the screen press **PF3**.

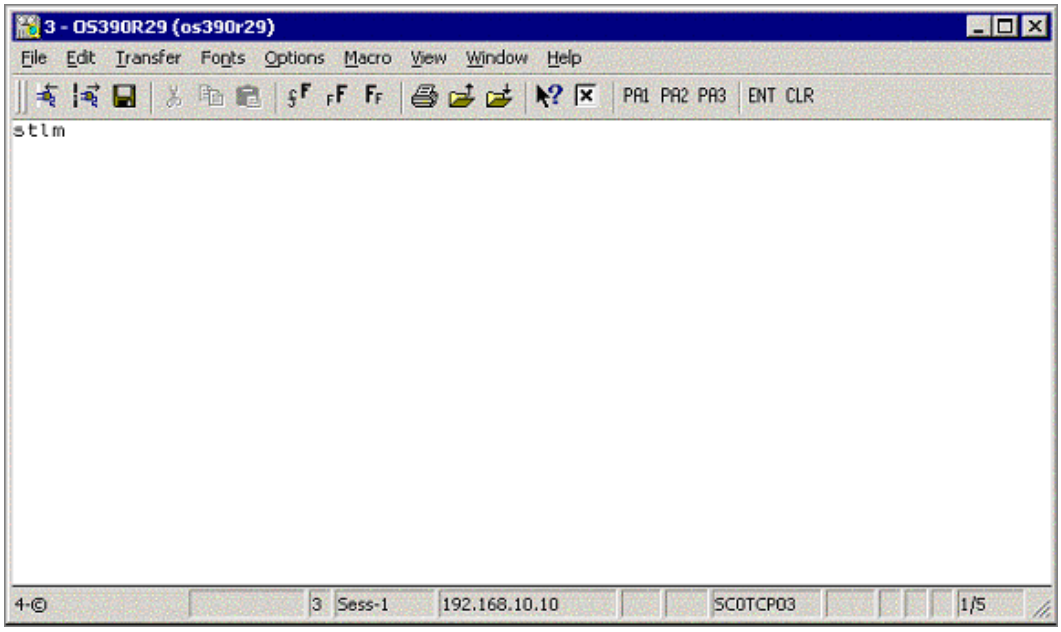
2.4.7. The SeeBeyond CICS Listener Monitor Screen for z/OS

After the components are installed, use the SeeBeyond CICS Listener monitoring screen to verify that all components are properly installed and working correctly.

Note: *An error message, "STCCLPAT tsq not found," may be displayed if no data is found in the tsq. If this occurs, it does not necessarily mean that the SeeBeyond CICS Listener installation was unsuccessful. Try opening the monitor again after running the CICS e*Way. Once data is present in the tsq the error will be resolved.*

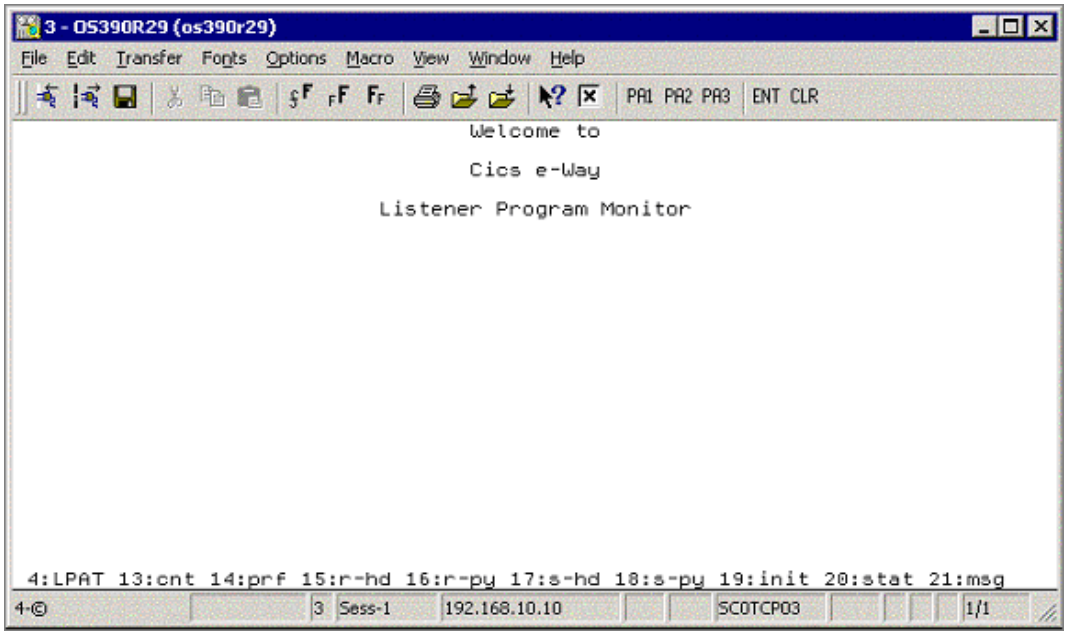
1 Logon to the CICS region that the SeeBeyond Listener is running in. Clear the screen, and type in **stlm** as displayed in Figure 11. Press **Enter**.

Figure 11



2 The menu screen appears as displayed in **Figure 12 on page 30**.

Figure 12

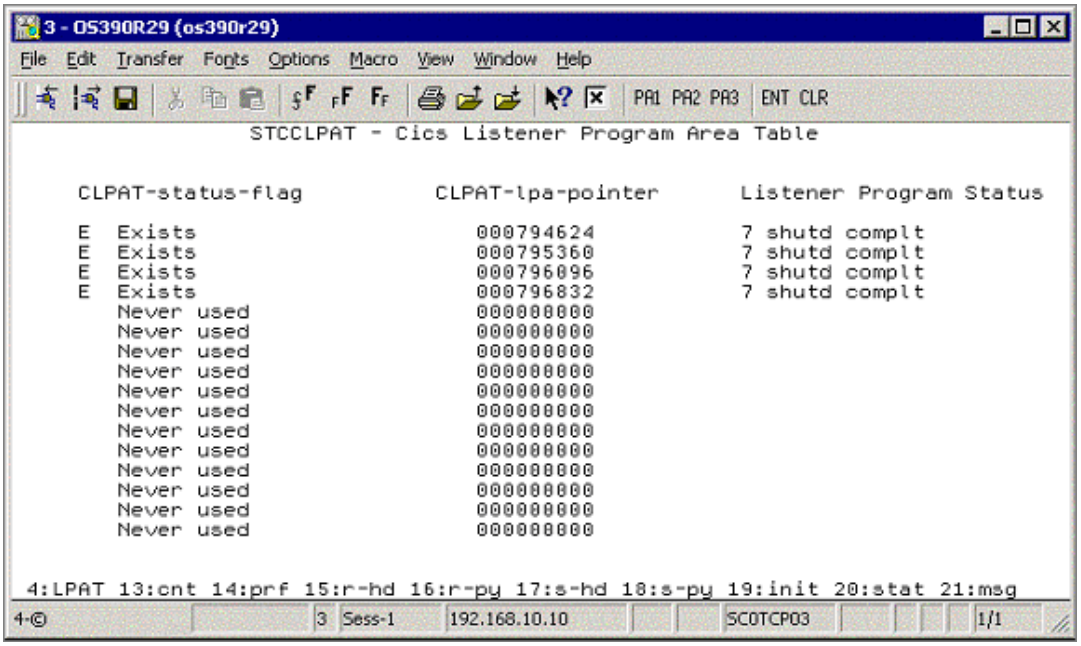


3 The **pfkeys** for this screen, and all of the monitor screens shown in this document, are as follows:

- **PF4**: displays the LPAT (Listener Program Area Table).
- **PF13**: displays the LPA counts and polling rates.
- **PF14**: displays the LPA performance statistics.
- **PF15**: displays the LPA last request header received from the CICS e*Way.
- **PF16**: displays the LPA last request payload received from the CICS e*Way.
- **PF17**: displays the LPA last response header sent to the CICS e*Way.
- **PF18**: displays the LPA last response payload sent to the CICS e*Way.
- **PF19**: displays the LPA initial record sent from the IBM Listener (EZACIC02).
- **PF20**: displays the LPA status.
- **PF21**: displays the LPA last error message sent to the CICS e*Way.

Press **PF4**. The **Listener Program Area Table** screen appears as displayed in **Figure 13 on page 31**.

Figure 13

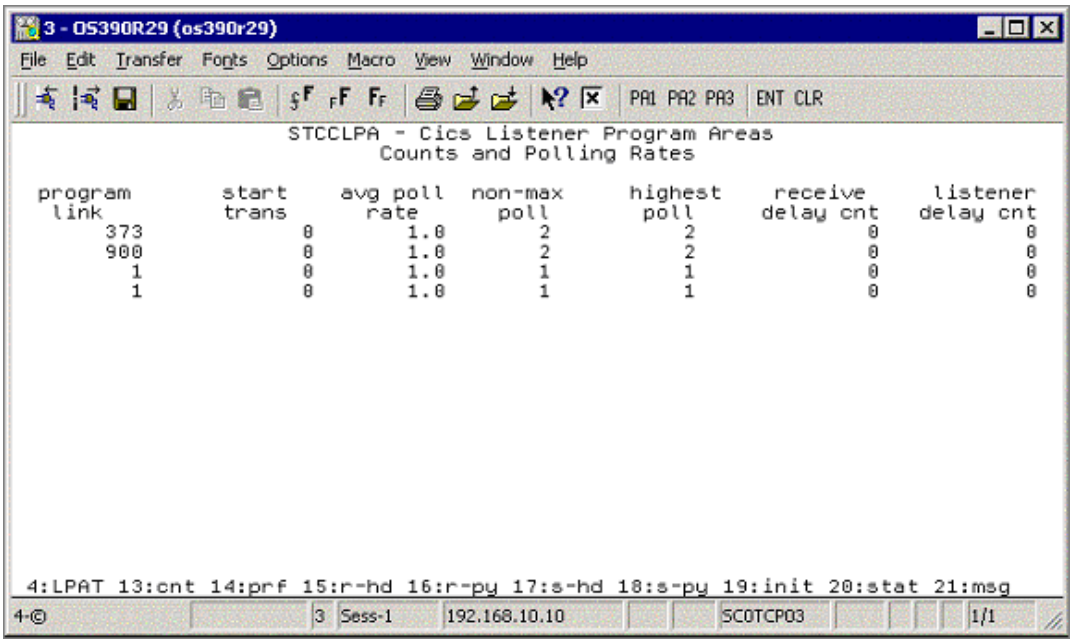


4 This screen displays the Listener Program Area Table (LPAT). Each line contains information about a Listener Program Area (LPA). There is one LPA associated with each instance of the SeeBeyond CICS Listener Program that is running or has run in this CICS region. The fields on the screen are:

- **CLPAT-status-flag**: shows the current status of the LPAT entry as follows:
  - **E**: Exists.
  - **N**: Never used.
  - **C**: Corrupted.
- **CLPAT-lpa-pointer**: shows the address of the most recent LPA in this entry.
- **Listener Program Status**: shows the last status of the Listener Program using this LPA.

Press **PF13.** The **LPA Counts and Polling Rates** screen appears as displayed in **Figure 14 on page 32**.

Figure 14



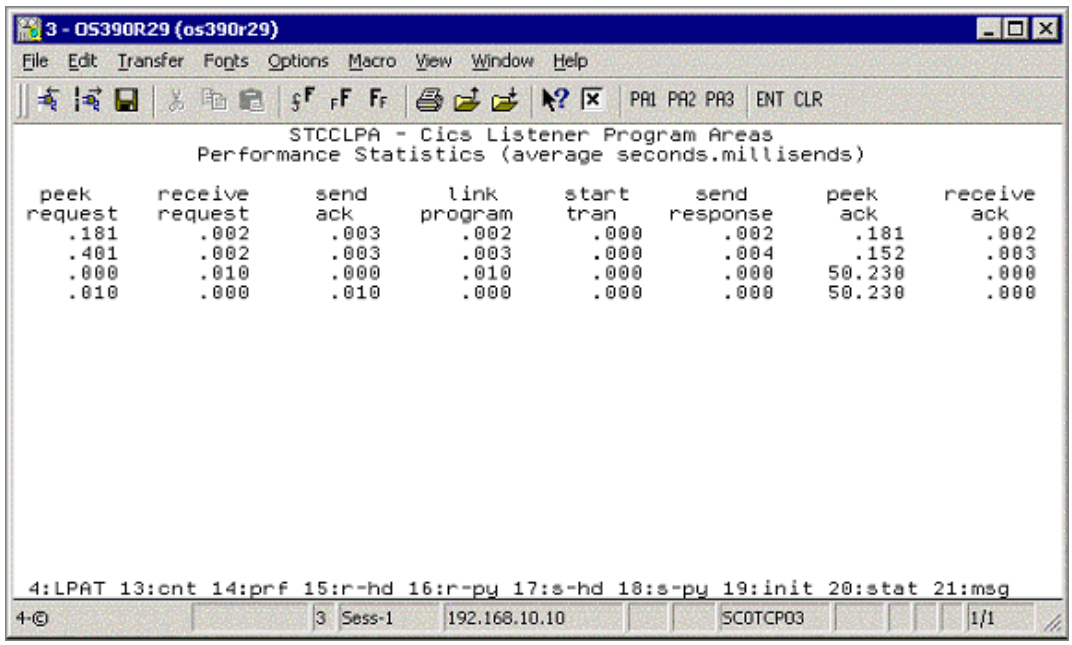
5 This screen displays counts and polling rates information for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program).

The fields on the screen are as follows:

- **program link**: the number of program links that have been requested.
- **start trans**: the number of transaction starts that have been requested.
- **avg poll rate**: the average number of receives per polling cycle to satisfy a full message receive from the CICS e*Way.
- **non-max cnt**: the 'high water mark' of receives within a polling cycle without reaching the polling rate (polling rate is sent in from the CICS e*Way when it initializes connection with the CICS Listener program).
- **highest poll**: the 'high water mark' of receives within a polling cycle that can include reaching the maximum allowed polling rate.
- **receive delay cnt**: the number of one second delays that occurred due to the Comm Timeout limit being reached (Comm. Timeout is sent in from the CICS e*Way when it initializes connection with the CICS Listener program).
- **listener delay cnt**: number of one second delays that occur due to the Listener Timeout limit being reached (Listener Timeout is sent in from the CICS e*Way when it initializes connection with the CICS Listener program).

Press **PF14**. The **LPA Performance Statistics** screen appears as displayed in **Figure 15 on page 33**.

Figure 15



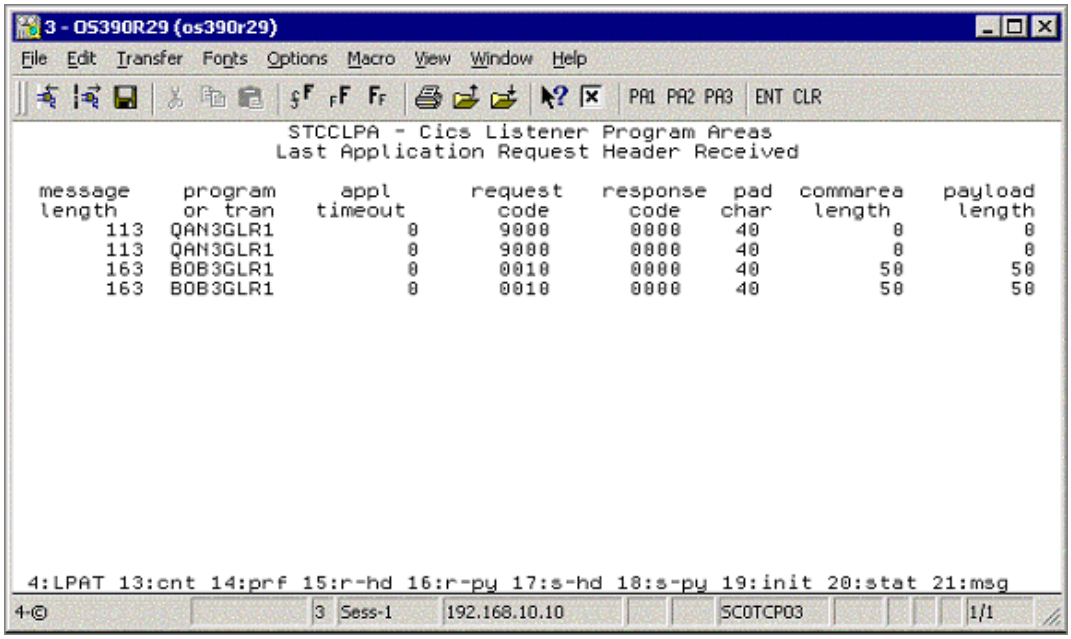
6 This screen (Figure 15) displays the performance statistics for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program).

All times shown are in sss.mmm format (sss = seconds, mmm=milliseconds). The fields on the screen are as follows:

- **peek request**: the average time spent peeking for the next incoming application request message.
- **receive request**: the average time spent receiving the next incoming application request message.
- **send ack**: the average time spent sending an ack for an incoming application request message.
- **link program**: the average time spent link to requested application programs.
- **start tran**: the average time spent starting requested application transactions.
- **send response**: the average time spent sending the application response (including the updated COMMAREA for program links) back to the e*Way.
- **peek ack**: the average time spent peeking for the incoming ack to the preceding send response.
- **receive ack**: the average time spent receiving the incoming ack to the preceding send response.

Press **PF15.** The **Last Application Request Header Received** screen appears as displayed in **Figure 16 on page 34**.

Figure 16

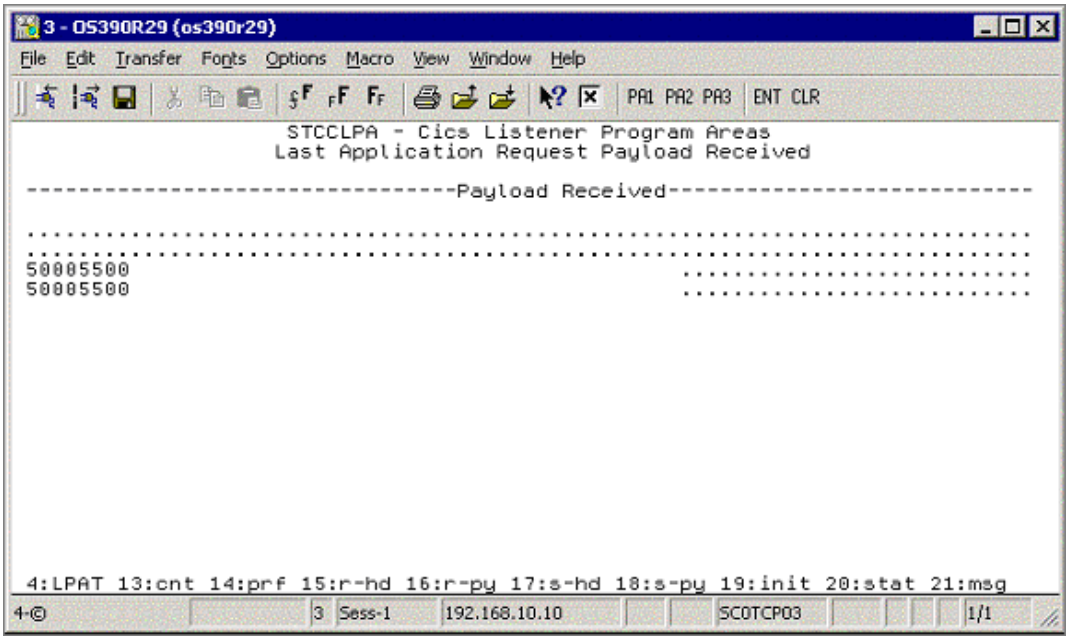


7 This screen (Figure 16) displays the last application request header received from the CICS e*Way for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program). The fields on the screen are as follows:

- **message length**: the entire length of the incoming message including the header and payload.
- **program or tran**: the requested application program or transaction.
- **appl timeout**: (reserved for future development).
- **request code**: which action is being requested.
  - 0010 = link to application program.
  - 0020 = start application transaction.
  - 0111 = ack for a link response.
  - 0121 = ack for a start response.
  - 9000 = shutdown the Listener program.
- **response code**: always set to zero from the CICS e*Way.
- **pad char**: padding character for the COMMAREA if the payload length is less than the COMMAREA length.
- **commarea length**: the length of the data to pass to the application program (link) or the application transaction (start).
- **payload length**: the length of the payload portion of the incoming message.

Press **PF16.** The **Last Application Request Payload Received** screen appears as displayed in **Figure 17 on page 35**.

Figure 17

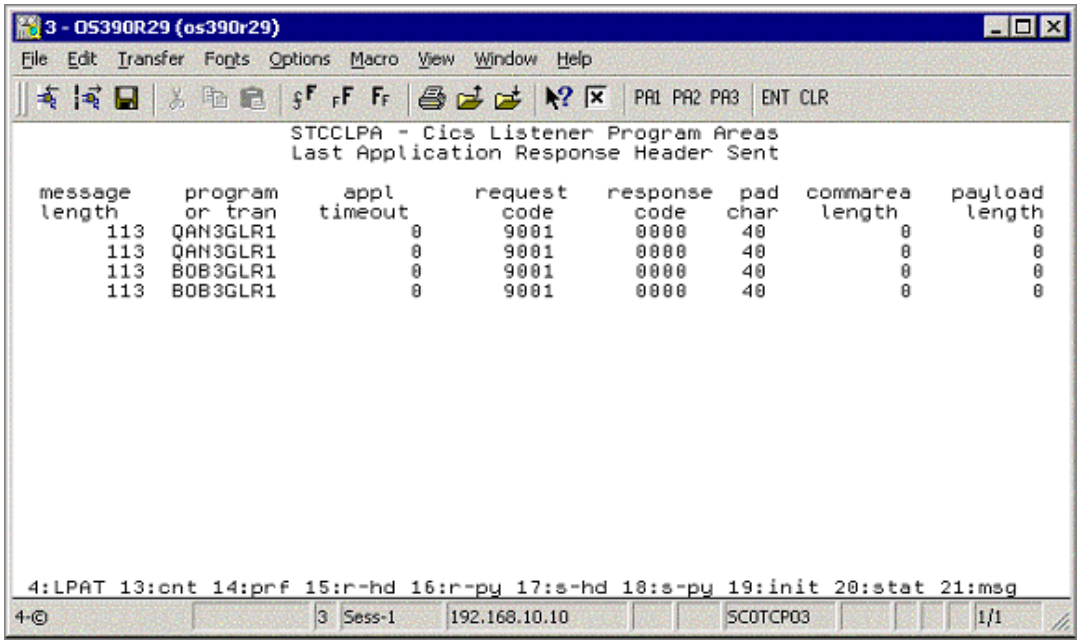


8 This screen (Figure 17) displays the last application request payload received from the CICS e*Way for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program). There is one field on the screen:

- **Payload Received**: the incoming payload from the CICS e*Way.

Press **PF17**. The **Last Application Response Header** screen appears as displayed in Figure 18.

Figure 18



9 The **Last Application Response Header** screen (**Figure 18 on page 35**) displays the last response header sent to the CICS e*Way for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program). The fields on the screen are as follows:

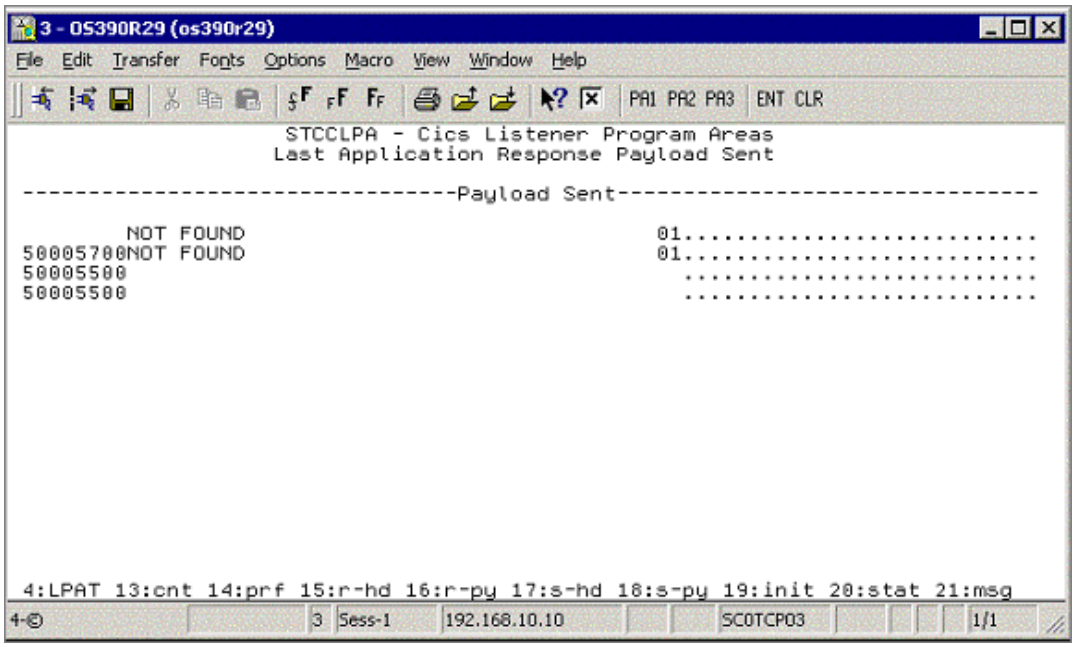
- **message length**: the entire length of the outgoing message including the header and payload.
- **program or tran**: the application program or transaction that was executed or started.
- **appl timeout**: (reserved for future development).
- **request code**: what action is being requested.
  - 0001 = ack the init request.
  - 0011 = ack for a link program request.
  - 0021 = ack for a start trans response.
  - 0110 = response from a linked application program (including COMMAREA).
  - 0120 = response from starting an application transaction (no COMMAREA included).
  - 9001 = ack for a shutdown request.
- **response code**: response code indicating what occurred while processing the previous incoming request from the e*Way.
  - 0000 = ok.
  - 0011 = bad startcode during initialization.
  - 0012 = bad password on init record.
  - 0013 = bad comm timeout on init record.
  - 0014 = bad appl timeout on init record.
  - 0015 = bad Listener timeout on init record.
  - 0016 = error reading the init record from transient data queue.
  - 0017 = error retrieving the init record from the start data.
  - 0018 = error while setting blocking mode.
  - 0051 = bad payload length.
  - 0052 = error while linking to a program.
  - 0053 = error while starting a transaction.
  - 0054 = bad request code.
  - 0055 = bad response code.
  - 0056 = partial message received.
  - 0057 = bad program name.
  - 0058 = bad message length.

- 0059 = bad commarea length.

- **pad char**: padding char that was sent in from the CICS e*Way.
- **commarea length**: COMMAREA length that was sent in from the CICS e*Way.
- **payload length**: payload length that was sent in from the CICS e*Way (the Listener sets this to zero if this is a response to a start transaction request).

Press **PF18**. The **Last Application Response Payload Sent** screen appears as displayed in Figure 19.

Figure 19

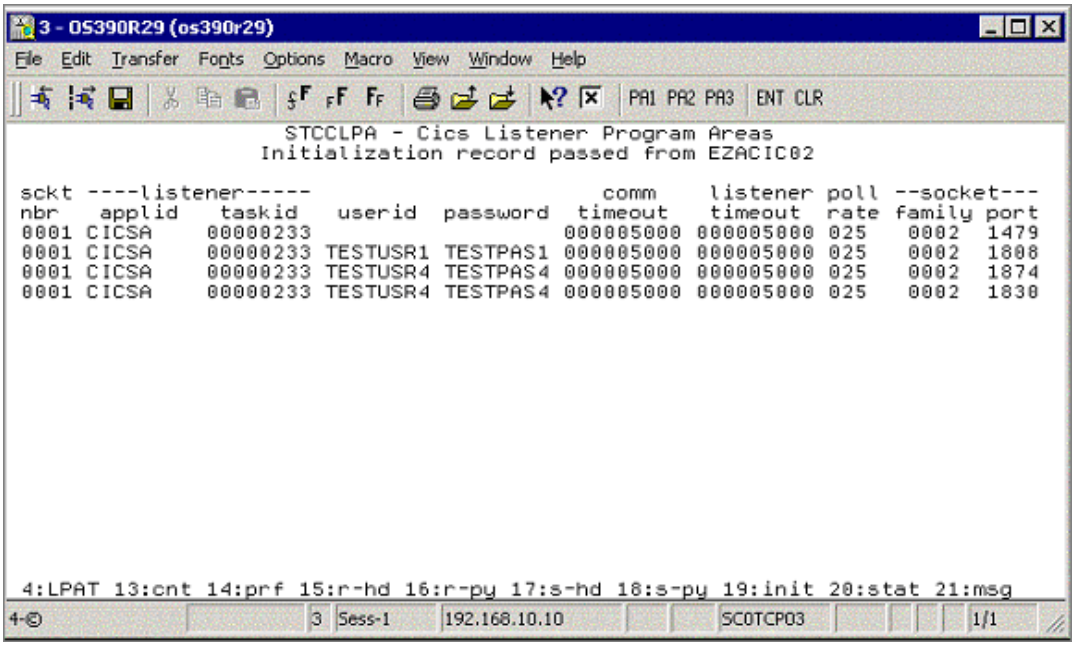


10 This screen (Figure 19) displays the last application response payload sent to the CICS e*Way for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener Program). There is one field on the screen:

- Payload sent: the outgoing payload being sent to the CICS e*Way.

Press **PF19**. The **Initialization Record Passed from EZACIC02** screen appears as displayed in **Figure 20 on page 38**.

Figure 20

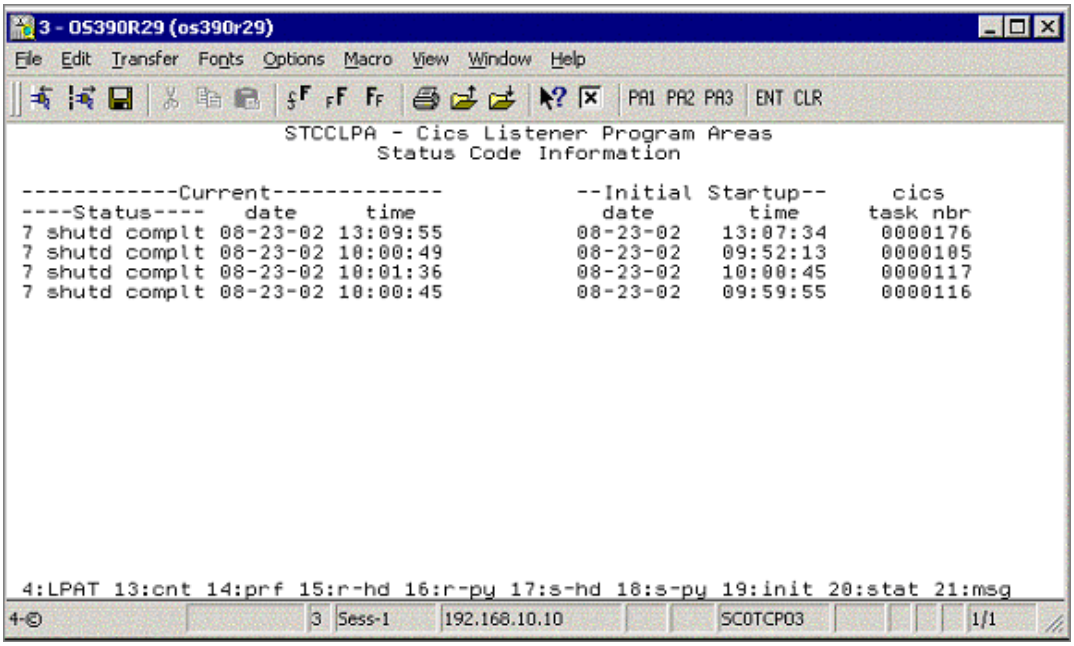


11 This screen (Figure 20) displays the init record that was passed to the SeeBeyond Listener program from the IBM Listener program (ezacic02) for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener program). The fields on the screen are as follows:

- **sckt nbr**: the socket number that is passed to the SeeBeyond Listener.
- **Listener applid**: the applid of the CICS region that the IBM Listener is executing in that started this instance of the SeeBeyond Listener.
- **Listener tasked**: the CICS taskid for the IBM Listener program that started this instance of the SeeBeyond Listener.
- **userid**: userid passed in from the CICS e*Way.
- **password**: user password passed in from the CICS e*Way.
- **comm. Timeout**: this timeout value is the threshold limit for waiting for all the bytes of an incoming message.
- **Listener timeout**: this timeout value is the threshold limit for waiting for a new incoming message.
- **poll rate**: how many times the SeeBeyond Listener will perform a receive loop to satisfy a complete message receive, after which it will wait one second before trying again.
- **socket family**: the TCP/IP family to which this socket belongs.
- **socket port**: the port this instance of the SeeBeyond Listener is using.

Press **PF20**. The **Status Code Information** screen appears as displayed in **Figure 21 on page 39**.

Figure 21



12 This screen (Figure 21) shows the status information for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener program). The fields on the screen are as follows:

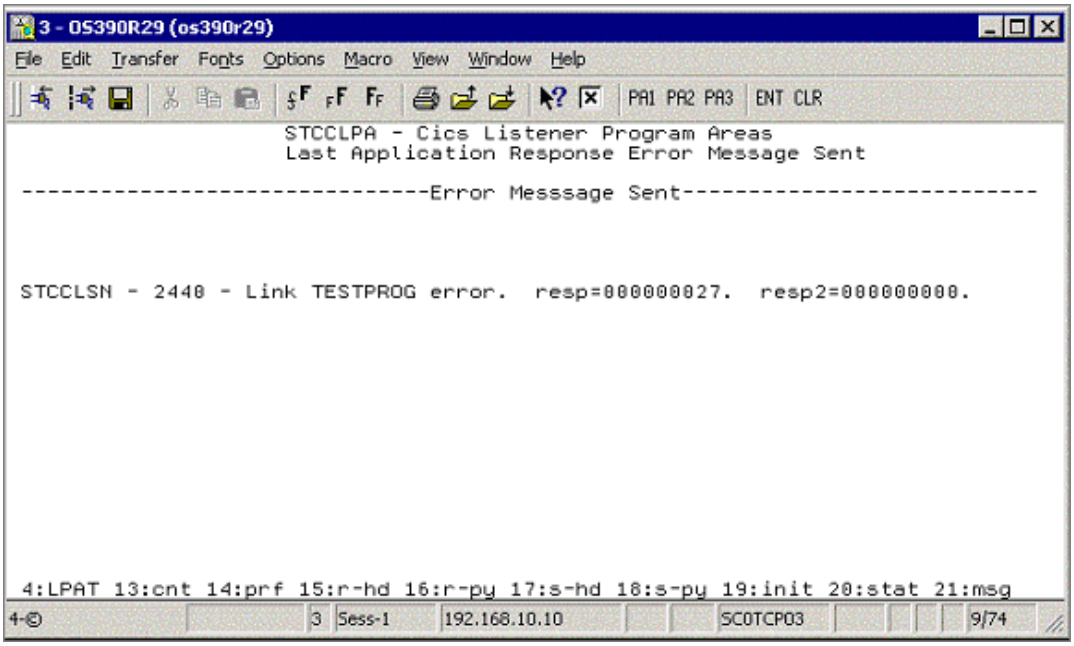
- **Current status**: the current status of an active SeeBeyond Listener program, or the last known status of a previously executing SeeBeyond Listener program.
  - C = LPA is initialized.
  - E = about to get ezacic02 data.
  - G = about to take socket from IBM Listener program.
  - I = about to set mode to blocking.
  - K = about to send ack for init request.
  - M = peeking for length of next incoming request.
  - O = peeking for entire incoming request.
  - Q = receiving full incoming request message.
  - S = sending ack for application request.
  - U = linking to requested application program.
  - W = starting a request application transaction.
  - Y = sending a application response.
  - 0 = peeking for application response ack.
  - 2 = receiving an application response ack.
  - 4 = a shutdown request has been received.
  - 6 = sending an ack for the shutdown request.

- 7 = shutdown is complete, this LPA is now available for reuse.
- 8 = SeeBeyond Listener program appended.

- **Current Status date**: date for this status.
- **Current Status time**: time for this status.
- **Initial Startup date**: date this instance of the SeeBeyond Listener was started.
- **Initial Startup time**: time this instance of the SeeBeyond Listener was started.
- **cics task nbr**: the CICS eibtaskn for this instance of the SeeBeyond Listener.

Press **PF21**. The **Error Message Sent** screen appears as displayed in Figure 22.

Figure 22



13 This screen (Figure 22) shows the last error message sent to the CICS e*Way for each LPA. Each line shows one LPA (one for each instance of the SeeBeyond Listener program). There is one field on the screen:

- **Error Message Sent**: the last error message sent to the CICS e*Way for this instance of the SeeBeyond Listener program.

2.4.8. CICS Listener Considerations for Invoking DB2 Applications

One of the following two techniques are required when using the CICS e*Way to run CICS applications that invoke a DB2 table:

- RCT entry for the STCL SeeBeyond CICS Listener Transaction
- PPT entry to redirect DB2 application to another AOR

RCT entry for the STCL SeeBeyond CICS Listener Transaction

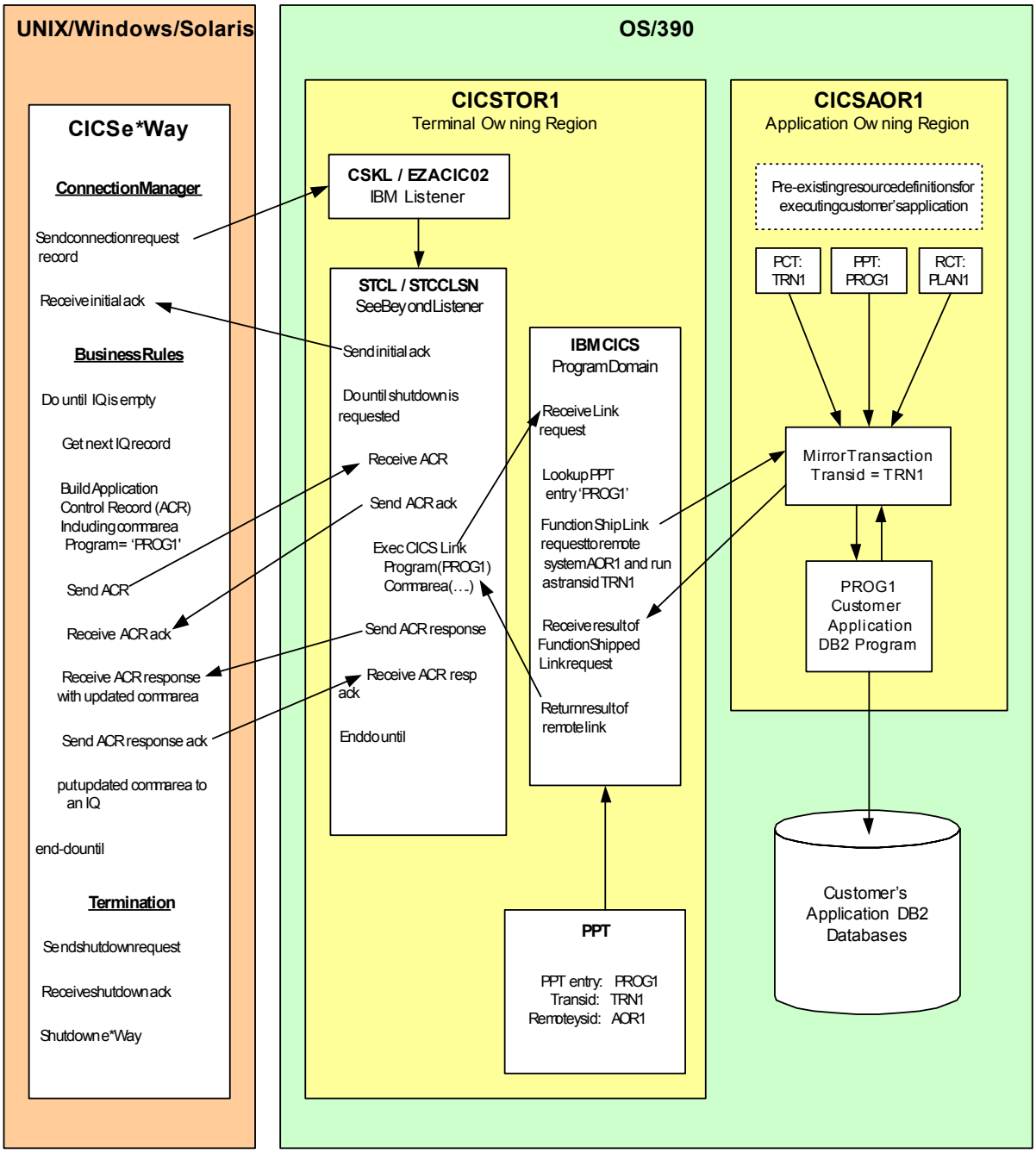
To run DB2 application programs under the STCL SeeBeyond CICS Listener transid, an RCT entry for the STCL transaction must be created using an RCT macro definition similar to the one shown below or by using an equivalent RDO entry.

```
DSNCRCT TYPE=ENTRY,TXID=STCL,THRDM=0,THRDA=0,THRDS=0,
TWAIT=POOL,PLAN=HD45LE06,AUTH=(SIGNID,*,*)
```

PPT entry to redirect DB2 application to another AOR

To redirect DB2 application programs to another AOR, the PPT entry for the requested DB2 application program in the TOR (where the STCL SeeBeyond CICS Listener transaction is running) must be defined to run the DB2 application program in a different region (usually an AOR) and under a specific PCT in that AOR. Refer to **Figure 23 on page 42** for a detailed flow diagram showing the use of the PPT and PCT.

Figure 23 SeeBeyond CICS Listener invoking DB2 programs



2.5 CICS Transaction Gateway 4.0 and 5.0 Configuration

IBM CICS Transaction Gateway properties are set using the CTG Configuration Tool. The Configuration Tool is located under the CICS Transaction Gateway program menu.

For system specific settings consult the CICS Transaction Gateway Documentation or visit the IBM CICS Library Website at **http://www-4.ibm.com/software/ts/cics/library/manuals/ctg40dl.html#configs** for information on CTG 4.0, or **http://www-3.ibm.com/software/ts/cics/library/cicstgv5.html** for CTG 5.0.

Chapter 3

CICSClient ETD Overview

This chapter provides an overview of the CICSClient ETD (cicsclient.xsc) hierarchy structure, including available methods, properties, and their application.

3.1 The CICSClient ETD

The CICSClient ETD (cicsclient.xsc), used within a request/reply schema is shown in Figure 24. The ETD is designed to be read only. A "top level" view of the ETD illustrates methods and attributes that assist the Collaboration writer in composing Business Rules to invoke the CICS program. Two Connection Establishment Transport modes are provided to communicate between the e*Way client and the CICS server with synchronous or asychronous calls, the CICS Transaction Gateway and the SeeBeyond CICS Listener.

3.1.1. CICSClient ETD Layout

CICS Communication Methods

connect on page 140
isConnected on page 165
disconnect on page 140
getServerList on page 158
execute on page 141
sendRequest on page 168

CICS Data Conversion Methods

getCommAreaString on page 144
getEncodedCommAreaString on page 148
toPackedDecimal on page 182
commAreaToPackedDecimal on page 138
packedDecimalToString on page 166
commAreaZonedToString on page 139
zonedToString on page 184
toZoned on page 183

Payload Nodes

CommArea on page 45
CommAreaLength on page 45

Asynchronous Call Handling Nodes and Methods

Node Descriptions

The nodes take their initial value from the e*Way Connection configuration parameters, but can be changed in the Collaboration using the Collaboration Rules Editor and calling the execute methods.

Url
CTG specific. The Url node contains the URL of the CICS Transaction Gateway.

Port
CTG specific. The Port node contains the port number necessary to communicate with the CICS Transaction Gateway.

SslClass
CTG specific. Specifies the full classname of the SSL KeyRing class.

SslPassword
CTG specific. Specifies the password for the encrypted KeyRing class.

EciSync
Specifies whether a call is synchronous (true) or asynchronous (false).

UserId
Used to authenticate access to the CICS program. The UserId node contains the CICS Userid under which that CICS program runs. under.

Password
Used to authenticate access to the CICS program. The Password node contains the CICS Userid that the CICS program runs under. The password gets encrypted when it is stored in the configuration file.

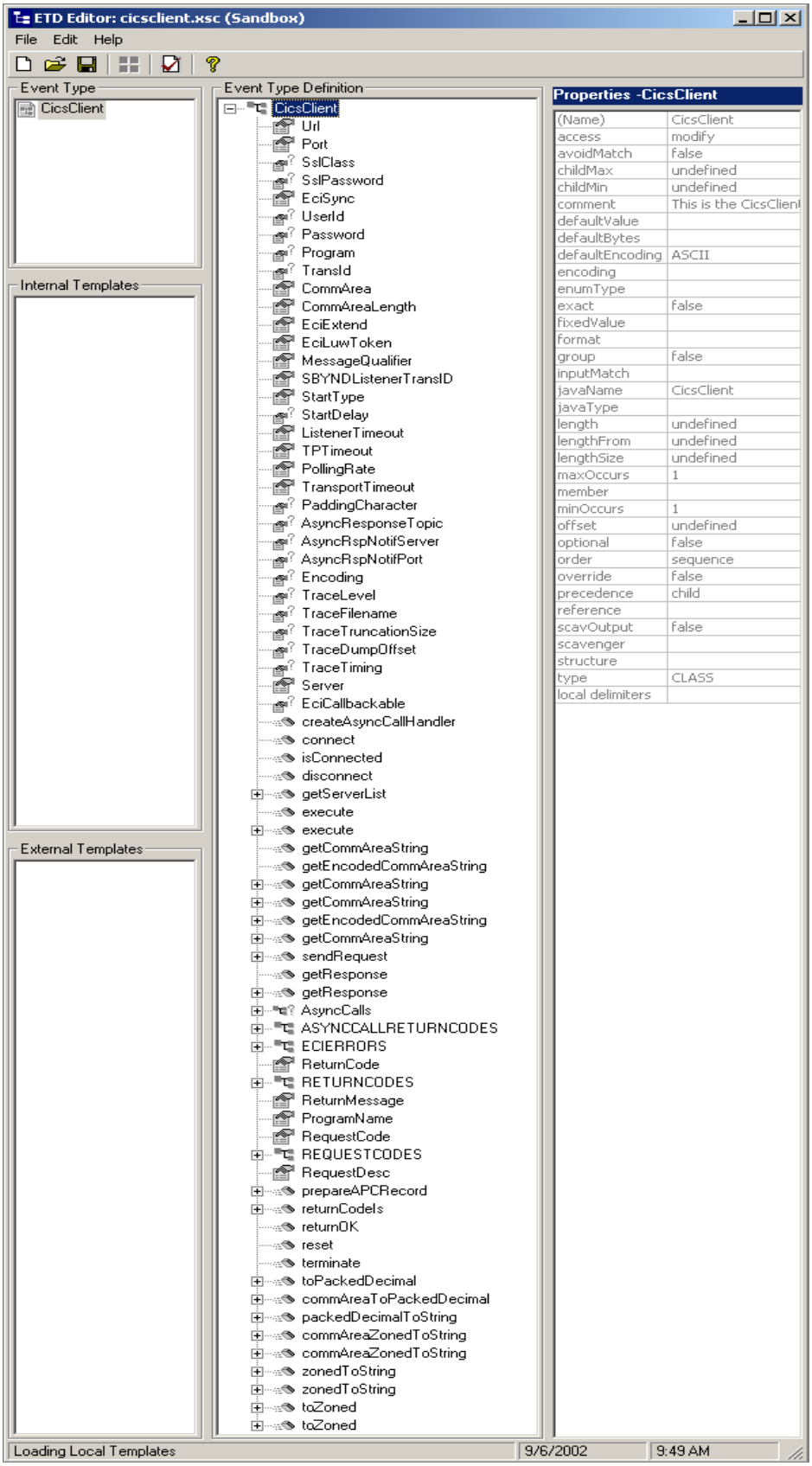
Program
The Program node contains the name of CICS program to be executed.

TransId
The TransId node contains the name of the CICS Transaction that the CICS program is executed under.

CommArea
CommArea is a payload node.

CommAreaLength
CommAreaLength is a payload node. Specifies the length (in bytes) of the Commarea passed to the ECI.

Figure 24 The CICSClient ETD



EciExtend
The EciExtend node is Boolean flag that sets the ECI mode to extended. True sets the mode to extended.

EciLuwToken
The EciLuwToken node contains the ECI logical unit of work token value.

MessageQualifier
The MessageQualifier node contains the Message Qualifier associated with a request.

SBYNDListenerTransID
Specifies the TransId of the SeeBeyond CICS Listener on the mainframe host. This is the CICS Transaction that the SeeBeyond CICS Listener is installed under.

StartType
Specifies the startup type. This can be either **IC** for CICS interval control or **TD** for CICS transient data. This is the CICS Startup type for the program being executed.

StartDelay
Specifies the hours, minutes and seconds (interval of time) to delay starting the transaction program (TP) on the CICS server for the IC Start Type. This field is optional but must specify all 6 digits (HHMMSS) if used.

ListenerTimeout
Specifies the estimated amount of time (in milliseconds) for the SeeBeyond CICS Listener to wait for the next incoming transaction program request from the CICS e*Way.

TPTimeout
Specifies the amount of time the CICS e*Way will wait for the SeeBeyond CICS Listener to return results for a current transaction program request.

PollingRate
Specifies the polling rate. This is the number of times the SeeBeyond CICS Listener will query the current TCP connection for incoming traffic before issuing an EXEC CICS DELAY for one second.

TransportTimeout
Specifies the timeout used by both the local and host side for receive on the socket.

PaddingCharacter
SeeBeyond CICS Listener Specific.The EBCDIC character used to pad the COMMAREA when the CICS program is called. The value must be coded in Hexadecimal. For example, 40 for Blanks, 00 for Low Values, FF for High Values, and so forth.

AsyncResponseTopic
Specifies a topic name for the response Event of an asynchronous CICS program call.

AsyncRspNotifServer
Specifies the host where the JMS server for asynchronous call completion event publishing and subscribing is running.

AsyncRspNotifPort
The port where the JMS server for asynchronous call completion event publishing and subscribing is listening.

Encoding
The Encoding node contains the default encoding used for the various COMMAREA methods.

TraceLevel
CTG specific. The TraceLevel node contains the debugging trace level.

TraceFilename
CTG specific. The TraceFilename node contains the name of the trace file to be used when TraceLevel has been set.

TraceTruncationSize
CTG specific. The TraceTruncationSize node contains the trace truncation size of the trace file that is written when TraceLevel and TraceFilename are set.

TraceDumpOffset
CTG specific. The TraceTruncationSize node contains the offset value for the trace dumping.

TraceTiming
CTG specific. The TraceTiming node contains the debugging trace timing.

Server
The Server node contains the server identity on which the CICS program is running. This is ignored when the Connection Transport is set to SeeBeyond CICS LISTENER.

EciCallbackable
The EciCallbackable node contains the asynchronous call handler. For CTG this is com.stc.eways.CICS.CTGReplyHandler. For SeeBeyond CICS Listener this is com.stc.eways.CICS.SBYNDListenerReplyHandle. This is used with createAsyncCallHandler( ) to create an appropriate asynchronous call handler and set it into this node (See **EciCallbackable** on page 56.)

AsyncCalls
Acts as a list of "**AsyncCall**" objects representing all of the outstanding calls initiated from the ETD instance. (See **AsyncCalls** on page 56.)

ASYNCCALLRETURNCODES
This node contains the return code or primary return code for a completed asynchronous call. It contains two values, CALL_OK, and CALL_ERROR. If it is the primary return code, it is represented by the sub-node ReturnCode in the **AsyncCalls** node. (See **ASYNCCALLRETURNCODES** on page 58.)

ECIERRORS
The ECIERRORS node contains all of the error codes in com.ibm.ctg.client.ECIReturncode.

ECI_NO_ERROR
ECI_ERR_INVALID_DATA_LENGTH
ECI_ERR_INVALID_EXTEND_MODE
ECI_ERR_NO_CICS

ECI_ERR_REQUEST_TIMEOUT
ECI_ERR_NO_REPLY
ECI_ERR_RESPONSE_TIMEOUT
ECI_ERR_TRANSACTION_ABEND
ECI_ERR_EXEC_NOT_RESIDENT
ECI_ERRLUW_TOKEN
ECI_ERR_SYSTEM_ERROR
ECI_ERR_NULL_WIN_HANDLE
ECI_ERR_THREAD_CREATE_ERROR
ECI_ERR_INVALID_CALL_TYPE
ECI_ERR_ALREADY_ACTIVE
ECI_ERR_RESOURCE_SHORTAGE
ECI_ERR_NO_SESSIONS
ECI_ERR_NULL_SEM_HANDLE
ECI_ERR_INVALID_DATA_AREA
ECI_ERR_INVALID_VERSION
ECI_ERR_UNKNOWN_SERVER
ECI_ERR_CALL_FROM_CALLBACK
ECI_ERR_INVALID_TRANSID
ECI_ERR_MORE_SYSTEMS
ECI_ERR_NO_SYSTEMS
ECI_ERR_SECURITY_ERRORS
ECI_ERR_MAX_SYSTEMS
ECI_ERR_MAX_SESSIONS
ECI_ERR_ROLLEDBACK
ECI_ERR_NO_MSG_QUALS
ECI_ERR_MSG_QUAL_IN_USE

These error codes are used with the secondaryReturnCodeIs (secondaryReturnCode) method in the **AsyncCalls** node, and allow the user to "drag & drop" code to generate Collaboration Rules for checking the error code of a completed asynchronous call. (See **ECIERRORS** on page 58.)

ReturnCode

The top level return code for a synchronous call. This node is checked for the return status of a synchronous call. This value should be checked against the constant values in RETURNCODES.

RETURNCODES

Provides the following ReturnCodes used to check the ReturnCode field in regard to synchronous calls:

```
SBYND_LISTENER_RC_OK
SBYND_LISTENER_RC_BAD_COMMAREA
SBYND_LISTENER_RC_CANNOT_GET_INIT_PARAM
SBYND_LISTENER_RC_PASSWD_ENCRYPT_ERROR
SBYND_LISTENER_RC_INVALID_TRANSPORT_TIMEOUT
SBYND_LISTENER_RC_INVALID_APPL_TIMEOUT
SBYND_LISTENER_RC_INVALID_LISTENER_TIMEOUT
SBYND_LISTENER_RC_LINK_ERROR
SBYND_LISTENER_RC_TRANS_START_ERROR
SBYND_LISTENER_RC_INVALID_REQCODE
SBYND_LISTENER_RC_INVALID_RSPCODE
SBYND_LISTENER_RC_PARTIAL_MSG
SBYND_LISTENER_RC_INVALID_PROGNAME
```

```
SBYND_LISTENER_RC_INVALID_MSG_LENGTH
SBYND_LISTENER_RC_INVALID_CA_LENGTH
SBYND_LISTENER_RC_BAD_COMMAREA
SBYND_LISTENER_RC_INIT_ERR_STARTUP_TYPE_TD
SBYND_LISTENER_RC_INIT_ERR_STARTUP_TYPE_IC
SBYND_LISTENER_RC_ERR_SET_NON_BLOCKING
```

ReturnMessage
Error text for the corresponding return code.

ProgramName
The name of the CICS program to be run in the CICS region.

RequestCode
The request code of the response.

REQUESTCODES
Provides the following RequestCodes used to populate the RequestCode field:

```
setSBYND_LISTENER_REQCODE_LSTNR_RSP4INIT
setSBYND_LISTENER_REQCODE_REQ_SYNC
setSBYND_LISTENER_REQCODE_LSTNR_RSP4SYNC
setSBYND_LISTENER_REQCODE_REQ_ASYNC
setSBYND_LISTENER_REQCODE_LSTNR_RSP4ASYNC
setSBYND_LISTENER_REQCODE_LSTNR_RETURN4SYNC
setSBYND_LISTENER_REQCODE_RSP4RETURN
setSBYND_LISTENER_REQCODE_REQ4SHUTDOWN
setSBYND_LISTENER_REQCODE_LSTNR_RSP4SHUTDOWN
setSBYND_LISTENER_REQCODE_LSTNR_RSP4ASYNCSTART
setSBYND_LISTENER_REQCODE_RSP4ASYNCLINKRSP
```

RequestDesc
The description for the request code.

3.2 Synchronous and Asynchronous Call Handling

Two underlying connection transport modes are provided to communicate between the e*Way client and the CICS server with synchronous or asychronous calls, the **CICS Transaction Gateway** and the **SeeBeyond CICS Listener**.

- **Synchronous mode**, in which the e*Way waits for the transaction to complete, and return the contents of the specified program's COMMAREA. Any data passed to the transaction will be inserted into the COMMAREA. This can be thought of as analogous to a phone call in which the caller makes the call and waits for a response.
- **Asynchronous mode**, in which a request is sent by the e*Way, but the sender does not wait for a response. The e*Way is able to move on to other tasks until the transaction is complete. This can be thought of as analogous to a mail message in which mail is sent and forgotten until sometime later when a response is received.

Asychronous Call Handling

The CICS Client ETD is designed to accommodate both connection transport options, IBM CICS Transaction Gateway (CTG), and the server side program, SeeBeyond CICS

Listener (STCL). The Asynchronous call notification mechanism differs between the two. The following section illustrates how the ETD is notified when an asynchronous call completes (Call Completion Event).

The Sub/Pub Model of the Call Completion Event

Collaborations that do asychronous call handling subscribe to a topic, AsyncResponseTopic, created in the schema for specific calls that are issued by Collaborations in the schema. The Collaboration that issues the call is called the "Call Initiator." The Collaboration that subscribes to the Call Completion Event for a specific call is called the "Call Subscriber." There can be multiple Collaborations subscribing to a Call Completion Event, including the Call Initiator.

Call completion events are generated by the underlying ETD implementation using interfaces provided by the underlying connection transport components, CTG or SeeBeyond CICS Listener. The components of the sub/pub model are:

1. A Call Completion Event (Event Type) must be created for each specific asynchronous call, to distinguish each.
2. A Call Subscriber must subscribe to the Call Completion Events in which it has interest.
3. A Call Initiator must provide the topic name that the Call Completion Event will be published to, prior to making the call (this must be the same as the Event Type name created in step 1). This information is obtained from the ETD node, AsyncResponseTopic, whose value can be set from the Collaboration.
4. The Call Initiator makes the call (for example, by calling **execute( )** on the CICS ETD). The underlying ETD implementation registers the call in the ETD node, **AsyncCalls**, which acts as a list of "**AsyncCall**" objects representing all of the outstanding calls initiated from the ETD instance. The information encapsulated in each of the **AsyncCall** objects include Program, StartTime, ReturnTime, ReturnCode, SecondaryReturnCode, ReturnMessage, CommArea, CommAreaLength, and Topic.
5. When a call completes, the underlying ETD implementation populates the call's information into the corresponding "**AsyncCall**" instance in the ETD node **AsyncCalls**. This information includes (as listed in step 4) status information such as **ReturnCode**, **SecondaryReturnCode**, **ReturnMessage** (text indicating the nature of an error), ReturnTime, CommArea, and CommAreaLength. The ETD implementation also publishes the CommArea, the payload returned from the CICS program as a message, to the Topic, so that subscribers are notified. Additional information, such as **ReturnCode**, **SecondaryReturnCode**, and **ReturnMessage** are attached to the message as properties.
6. The Call Completion Event is "consumed" by the Call Initiator, directly from the instance in the **AsyncCalls** list, or by the Call Subscribers, triggered by an asynchronous event, from **AsyncResponsTopic**. When the Completion Event is consumed directly from the **AsyncCall** instance, several of the methods and nodes are available to facilitate "harvesting" the information. The method is**Done( )** can be used to test whether the outstanding call has completed. If "True" is returned (completed), navigation methods, such as **hasNext( )**, **next( )**, and **remove( )** allow

the Collaboration to search the pool, check outstanding calls, and harvest results or status information as needed.

Figure 25 Asynchronous Call Handling - Completion Event

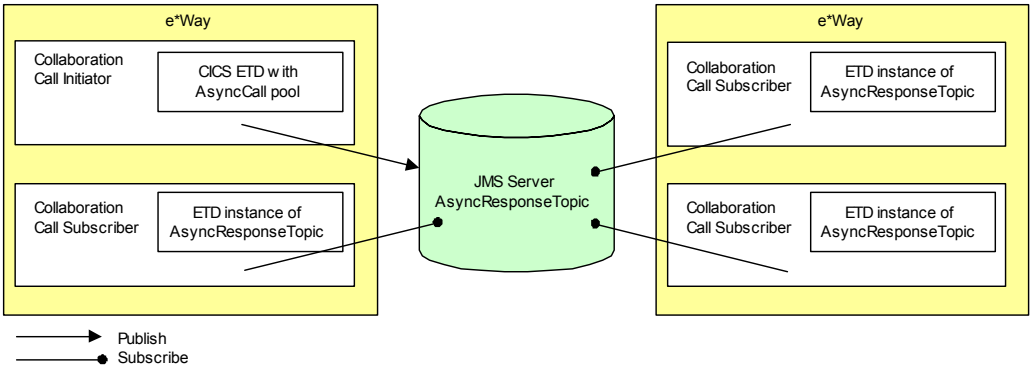


Figure 25 illustrates how Collaborations interact with the asychronous calls Completion Event in the context of a schema.

Asynchronous Call Handling Using CICS Transaction Gateway

With CICS Transaction Gateway as the underlying connection transport, the CICSClient ETD instance instantiates and keeps a reference of the **com.stc.eways.cics.CTGReplyHandler** (implements **com.ibm.ctg.client.CallBackable** and java.lang.Runnable) which contains reply handling logic running in its own thread. This is started at the completion of an asynchronous call. The reply logic is as follows:

- Check the status information in the **com.ibm.ctg.client.GatewayRequest** object, and populate this information in the **AsyncCall** object in the **AsyncCalls** pool.
- Generate a byteMessage with appropriate status information as its properties and publish the message to the topic (as an attribute of the **AsyncCall** object).

Figure 26 Asynchronous Call Handling - CICS Transaction Gateway

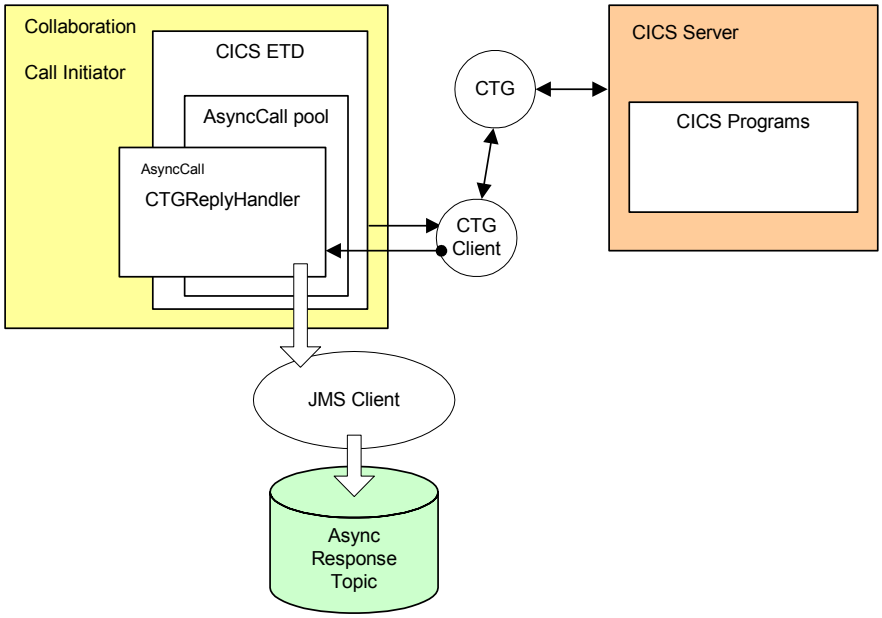


Figure 26 illustrates how CICSClient ETD communicates with the underlying transport, IBM CICS Transaction Gateway, to make a call and get notification when the call completes.

Asynchronous Call Handling Using the SeeBeyond CICS Listener

When using the SeeBeyond CICS Listener as the underlying connection transport, a worker thread is created using com.stc.eways.cics.ListenerReplyHandler as the target. The corresponding **AsyncCall** object keeps a reference of this worker thread, which is started by the ETD logic (Collab thread) when an asynchronous call is issued. To the Collaboration, the call is asynchronous, but "under the hood" of the CICSClient ETD the call to the CICS program is actually issued as a synchronous call. The main thread, that is, the Collaboration thread, spawns a worker thread which is blocked by the **getResponse( )** method for a response (for this asynchronous call). After the Collaboration thread spawns the worker thread it proceeds to execute other Collaboration logic, achieving the asynchronous call effect. (See Figure 27.)

Figure 27 Asynchronous Call Handling - SeeBeyond CICS Listener

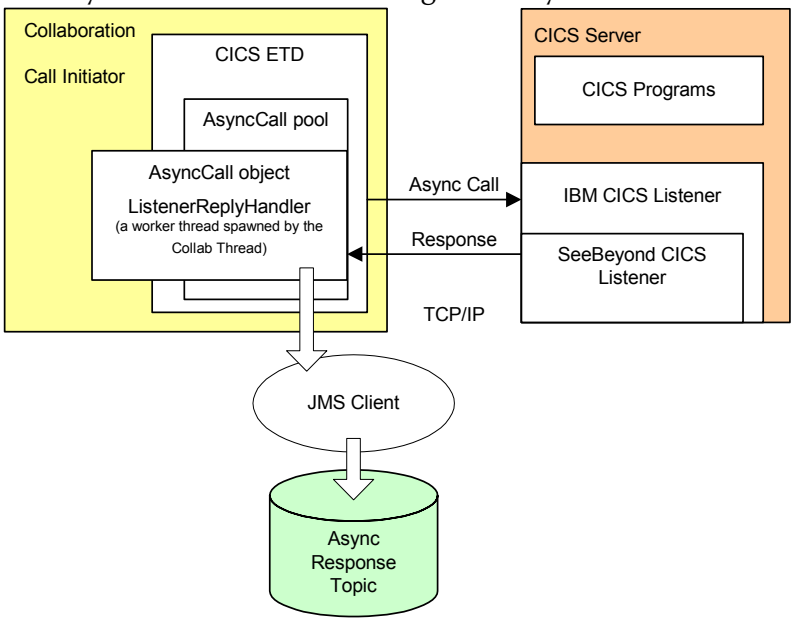
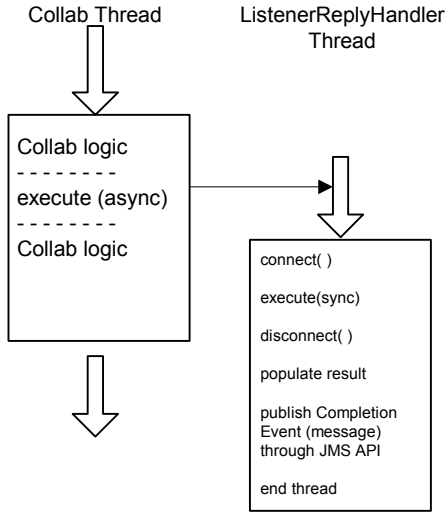


Figure 27 illustrates how the CICSClient ETD communicates with the underlying transport, SeeBeyond CICS Listener, to make a call and get notification when the call completes. Figure 28 presents a thread diagram showing how an asynchronous call is implemented when the SeeBeyond CICS Listener is used as the connection transport.

Figure 28 Asynchronous Call Handling - Thread Diagram



CICSClient ETD Asynchronous Configuration

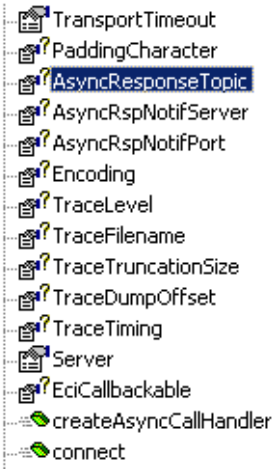
The configuration parameter, Async Response Topic, located in CICS e*Way Connection configuration, CICS Client section, is a JMS topic. The CICSClient ETD's Asynchronous call handling logic publishes a call to the topic when completes. The topic is subscribed to by any Collaboration that wants to be notified when an asynchronous program call is returned. The default value for the Async Response Topic parameter is blank, which means that JMS messaging is not used for

asynchronous call handling, but the Call Initiator can still harvest asynchronous call results by iterating through the **AsyncCalls** pool (see **Asychronous Call Handling** on page 50 for details).

ETD Nodes Associated with Asynchronous Call Handling

The Async Response Topic, Async Call JMS Server Host, and Async Call JMS Server Port, configuration parameters provide the initial settings for the CICSClient ETD nodes **AsyncResponseTopic**, **AsyncRspNotifServer**, **AsyncRspNotifPort** (see Figure 29). The settings for these nodes can be changed in the Collaboration by calling their corresponding set methods. If a Collaboration is making a specific asynchronous call, and other Collaborations are interested in the result of the call, an Event Type with a name such as "XYZ" can be created in the schema, and interested parties can subscribe to "XYZ." Before the call is issued, the Call Initiator (the Collaboration issuing the call) sets the **AsyncResponseTopic** as "XYZ," and sets **AsyncRspNotifServer** and **AsyncRspNotifPort** accordingly.

Figure 29 CICSClient ETD - Asynchronous Response Topic



Additional nodes and methods used specifically for asynchronous call handling include **AsyncCalls**, **ASYNCCALLRETURNCODES**, **ECIERRORS**, and **createAsyncCallHandler( )**. The EciCallbackable node is used to hold an **AsyncCallHandler** object used by the subsequent asynchronous call. Each asynchronous call has its own instance of an **AsyncCallHandler** object. The following section defines how each of these nodes and methods are used in an asynchronous call.

createAsyncCallHandler( )

Each time the **createAsyncCallHandler( )** method is called, it returns an instance of a proper handler class, depending on whether CTG or the SeeBeyond CICS Listener is used.

- When CTG is used - an instance of **com.stc.eways.cics.CTGReplyHandler** is instantiated and returned.
- When SBYND Listener is used - an instance of **com.stc.eways.cics.SBYNDListenerReplyHandler** is instantiated and returned.

This object is runnable, executed in a separate thread when the asynchronous call returns.

EciCallbackable

The **EciCallbackable** node is used to hold the current asynchronous call handler. It is typically generated by invoking **createAsyncCallHandler( )**. When set with an instance of **AsyncCallHandler**, the handler instance will be used by the subsequent **execute( )**.

In addition, the Collaboration can call another **execute(…… asyncCallhandler)** and feed the asynchronous call handler directly from the **EciCallbackable** node or invoking the method **createAsyncCallHandler( )**.

The Collaboration uses the following gestures to tell the ETD how to handle an asynchronous call.

When **execute( )** or **execute(... ... asyncCallHandler)** is called:

```
IF AsyncCallHandler is NULL
{
      // no async call completion handling
      // is needed on eway side, just launch
      // the CICS program
}
ELSE {
      // async call completion handling
      // is needed, a AsyncCall object
      // will be put into the outstanding
      // async call pool (AsyncCalls)
      // with a reference to its
      // AsyncCallHandler:
      // either:
      // com.stc.eways.cics.CTGReplyHandler
      // or
      // com.stc.eways.cics.SBYNDListenerReplyHandler
      //
      IF AsyncResponseTopic is NULL
      {
            // This means though async call
            // completion needs to be
            // handled but no need to publish
            // it - only populate the result
            // and status information into
            // the pool, and can be harvested
            // by the same collaboration later
      }
      ELSE {
            // This means when async call
            // completes, the result and status
            // are populated into the pool
            // and also the completion event
            // is published to the topic;
      }
}
```

AsyncCalls

The **AsyncCalls** node represents a pool of outstanding asynchronous calls issued via the ETD instance.

The methods used to navigate the asynchronous calls pool are:

hasNext( )
Checks to see if there is next call object in the iterator.

next( )
Makes the next call object in the iterator the current call object.

remove( )
Removes the current call object from the iterator. Typically this is called after a call is completed and the result has been harvested.

The following methods are used to probe for the call status and result. These are all implicitly applied to the current **AsyncCall** object in the pool:

isDone( )
Returns true when the asynchronous call is returned.

needReply( )
Returns true when the asynchronous call has a handler registered. A non-null asynchronous call handler in an **AsyncCall** node is a gesture from the Collaboration that the asynchronous call completion needs to be handled.

returnCodeIs(code)
Checks the primary return code for the current call object. This is valid only when **isDone( )** returns true.

secondaryReturnCodeIs(code)
Checks the secondary return code for the current call object. This is valid only when **isDone( )** returns true.

The following attributes (sub nodes of **AsyncCalls**) are provided as properties for the current call object:

Topic
The event type name (JMS topic) that the ETD publishes to when the call completes.

Program
The name of the invoked CICS program.

StartTime
The time when the call request was issued by the Collaboration.

CommArea
The COMMAREA for this specific asynchronous call. Typically, it contains the data passed to the CICS program before the call completes, and contains the data passed back from the CICS program after the call completes.

CommAreaLength
The length of the COMMAREA for this specific asynchronous call.

ReturnTime
The time when the call completed.

ReturnCode
The primary return code for the current call. If it is **CALL_ERROR**, check **SecondaryReturnCode** for more details.

SecondaryReturnCode
When CTG is used, the value must be in **ECIERRORS**. Otherwise, the value is in **RETURNCODES**.

ReturnMessage
The corresponding text message for the SecondaryReturnCode.

ASYNCCALLRETURNCODES
This node contains the return code or primary return code for a completed asynchronous call. It contains two values, **CALL_OK**, and **CALL_ERROR**. If it is the primary return code, it is represented by the sub-node ReturnCode in the **AsyncCalls** node.

ECIERRORS
The **ECIERRORS** node contains all of the error codes in **com.ibm.ctg.client.ECIReturncode**. These error codes are used with the **secondaryReturnCodeIs**(**secondaryReturnCode**) method in the **AsyncCalls** node, and allow the user to "drag & drop" code to generate Collaboration Rules for checking the error code of a completed asynchronous call.

Connection Management and Asynchronous Call Handling

When issuing asynchronous calls through a CICS e*Way Connection, the ETD implementation treats asynchronous (and synchronous) calls differently depending on whether CTG or SeeBeyond Listener is used.

CICS Transaction Gateway

With the CICS Transaction Gateway, every time an asynchronous call is issued, a new instance of **com.ibm.ctg.client.JavaGateway** is created and used to send the request (asynchronous call). The **CTGReplyHandler** instance (implements **com.ibm.ctg.client.Callbackable**), registered with the request, is responsible for closing the gateway object and releasing all the resources.

SeeBeyond CICS Listener

With the SeeBeyond CICS Listener, a separate TCP/IP socket connection is created whenever an asynchronous call is issued, and a new session of the SBYND CICS listener (a new instance of the listener) is created. The request is issued over this session as a synchronous call. This is done by an instance of a ListenerReplyHandler associated with the asynchronous call. After the call is returned, the session is closed.

For an asynchronous call, the connection to the CICS server is established for the call, and disconnected and released after the asynchronous call completes. This connection is independent of the connection for the ETD which is managed by Connection Management. In other words, the connection for an asynchronous call is not managed by the Connection Management mechanism.

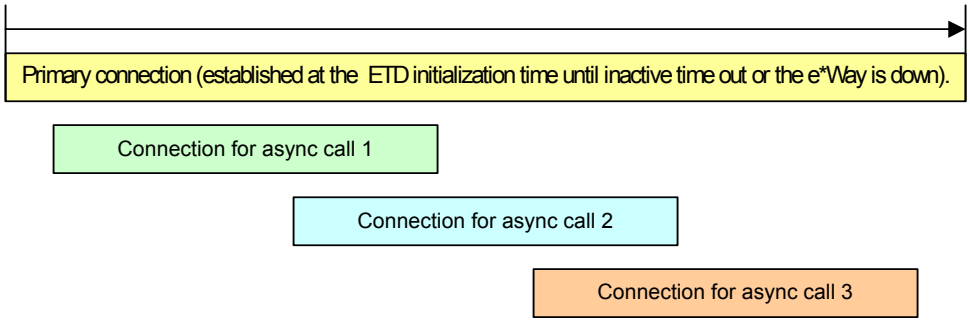
The following illustrates the relationship between the connection in the ETD (the primary connection), and connections for asynchronous calls (outstanding).

Automatic Connection Establishment Mode

When the primary connection's Connection Establishment Mode is set as Automatic, the primary connection and possible asynchronous call connections co-exist in parallel.

The Collaboration can initiate additional asynchronous calls that establish their own connections until their completion, and at the same time, do synchronous calls on the primary connection (as illustrated in Figure 30).

Figure 30 Connection Establishment Mode - Automatic

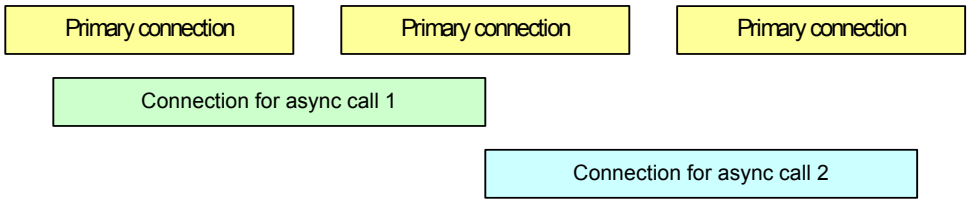


On Demand Connection Establishment Mode

When set as On Demand, the primary connection is established whenever there is a demand (invoking **executeBusinessRules( )**) and released after it is used.

The Collaboration can initiate additional asynchronous calls that establish their own connection to the CICS server until their completion, and at the same time, do synchronous calls over the primary connection (as illustrated in Figure 31).

Figure 31 Connection Establishment Mode - On Demand



Manual Connection Establishment Mode

The Manual connection mode is similar to the On Demand mode, except that if there is a primary connection it is established by the Collaboration business logic.

Chapter 4

e*Way Configuration

This chapter describes how to configure the following components of the CICS e*Way:

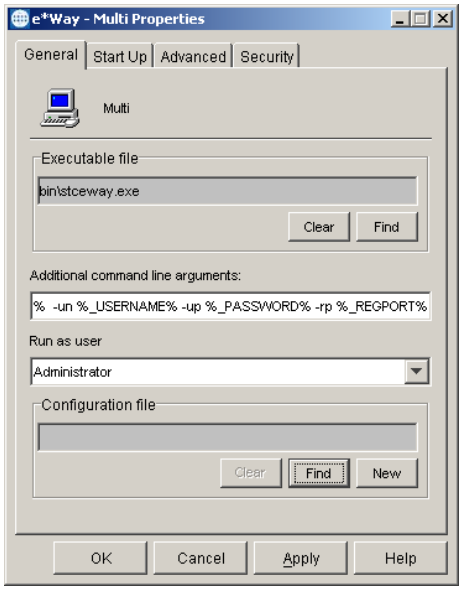
4.1 Multi-Mode e*Way Configuration

Multi-Mode e*Way properties are set using the Schema Designer.

To create and configure a New Multi-Mode e*Way:

1. Select the Navigator's Components tab.
2. Open the host and control broker on which you want to create the e*Way.
3. On the Palette, click on the **Create a New e*Way** button.
4. The New e*Way Component window opens. Enter the name of the new e*Way, then click **OK**.
5. Right-click the new e*Way and select **Properties** edit its properties.

Figure 32 Multi-Mode e*Way Properties



6 When the e*Way Properties window opens, click on the **Find** button beneath the **Executable File** field, and select an executable file. For the purposes of the sample select **stceway.exe** (**stceway.exe** is located in the "bin\" directory).

7 Under the **Configuration File** field, click on the **New** button. When the Settings page opens, set the configuration parameters for this configuration file.

8 After selecting the desired parameters, save the current configuration. Close the **.cfg** file and select **OK** to close the e*Way Properties Window.

Multi-Mode e*Way Configuration Parameters

The Multi-Mode e*Way configuration parameters are arranged in the following sections:

- **JVM Settings** on page 61
- **General Settings** on page 65

4.1.1. JVM Settings

The JVM Settings control basic Java Virtual Machine settings.

- **JNI DLL Absolute Pathname** on page 61
- **CLASSPATH Prepend** on page 62
- **CLASSPATH Override** on page 62
- **CLASSPATH Append From Environment Variable** on page 63
- **Initial Heap Size** on page 63
- **Maximum Heap Size** on page 63
- **Maximum Stack Size for Native Threads** on page 63
- **Maximum Stack Size for JVM Threads** on page 64
- **Disable JIT** on page 64
- **Remote Debugging port number** on page 64
- **Suspend option for debugging** on page 64
- **Auxiliary JVM Configuration File** on page 64

JNI DLL Absolute Pathname

Description

Specifies the absolute pathname to where the JNI DLL installed by the *Java 2 SDK* 1.3.1_02 is located on the Participating Host.

Required Values

A valid pathname.

Additional Information

The JNI dll name varies on different O/S platforms:

OS	Java 2 JNI DLL Name
Windows	jvm.dll
Solaris 2.6, 2.7, 2.8	libjvm.so
z/OS	libjvm.so
HP-UX	libjvm.sl
AIX 4.3.3 and 5.1	libjvm.a

The value assigned may contain a reference to an environment variable. To do this, enclose the variable name within a pair of % symbols. For example:

```
%MY_JNIDLL%
```

Such variables are used when multiple Participating Hosts are used on different platforms.

Note: *To ensure that the JNI DLL loads successfully, the Dynamic Load Library search path environment variable must be set appropriately to include all the directories under the Java 2 SDK (or JDK) installation directory that contain shared libraries (UNIX) or DLLs.*

CLASSPATH Prepend

Description

Specifies the paths to be prepended to the CLASSPATH environment variable for the JVM.

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

If left unset, no paths are prepended to the CLASSPATH environment variable.

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_PRECLASSPATH%
```

CLASSPATH Override

Description

Specifies the complete CLASSPATH variable to be used by the JVM. This parameter is optional. If left unset, an appropriate CLASSPATH environment variable (consisting of required e*Gate components concatenated with the system version of CLASSPATH) is set.

Note: *All necessary JAR and ZIP files needed by both e*Gate and the JVM must be included. It is advised that the* ***CLASSPATH Prepend*** *parameter be used.*

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_CLASSPATH%
```

CLASSPATH Append From Environment Variable

Description

Specifies whether the path is appended for the CLASSPATH environmental variable to jar and zip files needed by the JVM.

Required Values

YES or **NO**. The configured default is YES.

Initial Heap Size

Description

Specifies the value for the initial heap size in bytes. If set to 0 (zero), the preferred value for the initial heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Heap Size

Description

Specifies the value of the maximum heap size in bytes. If set to 0 (zero), the preferred value for the maximum heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Stack Size for Native Threads

Description

Specifies the value of the maximum stack size in bytes for native threads. If set to 0 (zero), the default value is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Stack Size for JVM Threads

Description

Specifies the value of the maximum stack size in bytes for JVM threads. If set to 0 (zero), the preferred value for the maximum heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Disable JIT

Description

Specifies whether the Just-In-Time (JIT) compiler is disabled.

Required Values

YES or **NO**.

Note: *This parameter is not supported for Java Release 1.*

Remote Debugging port number

Description

Specifies the port number by which the e*Gate Java Debugger can connect with the JVM to allow remote debugging.

Required Values

An unused port number in the range 2000 through 65535. If not specified, the e*Gate Java Debugger is not able to connect to this e*Way.

Suspend option for debugging

Description

Allows you to specify that the e*Way should do no processing until an e*Gate Java Debugger has successfully connected to it.

Required Values

YES or **No**. YES suspends e*Way processing until a Debugger connects to it. NO enables e*Way processing immediately upon startup.

Auxiliary JVM Configuration File

Description

Specifies an auxiliary JVM configuration file for additional parameters.

Required Values

The location of the auxiliary JVM configuration file.

4.1.2. General Settings

For more information on the General Settings configuration parameters see the *e*Gate Integrator User's Guide*. The General Settings section contains the following parameters:

- **Rollback Wait Interval** on page 65
- **Standard IQ FIFO** on page 65

Rollback Wait Interval

Description

Specifies the time interval to wait before rolling back the transaction.

Required Values

A number within the range of 0 to 99999999, representing the time interval in milliseconds.

Standard IQ FIFO

Description

Specifies whether the highest priority messages from all STC_Standard IQs will be delivered in the first-in-first-out (FIFO) order.

Required Values

Select **YES** or **NO**. YES indicates that the e*Way will retrieve messages from all STC_Standard IQs in the first-in-first-out (FIFO) order. NO indicates that this feature is disabled. NO is the configured default.

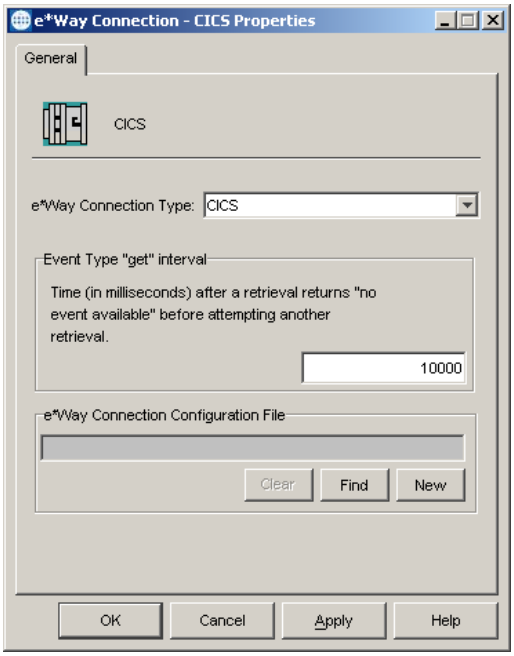
4.2 e*Way Connection Configuration

e*Way Connections are set using the Schema Designer.

To create and configure a CICS e*Way Connection:

1. In the Schema Designer's Component editor, select the **e*Way Connections** folder.
2. On the palette, click the **Create a New e*Way Connection** button. The New e*Way Connection Component dialog box opens. Enter a name for the new e*Way Connection and click **OK**.
3. Double-click on the new e*Way Connection. The e*Way Connection Properties dialog box opens.

Figure 33 e*Way Connection Properties



4. From the e*Way Connection Type drop-down box, select **CICS**.
5. Enter the Event Type "get" interval in the dialog box provided. The configured default is 10000 milliseconds.
6. From the e*Way Connection Configuration File, click **New** to create a new Configuration File for this e*Way Connection. (To use an existing file, click **Find**.)
7. The **e*Way Connection Configuration Editor** opens. Make any necessary changes to the CICS e*Way Connection parameters.
8. From the **File** menu, click **Save** to save settings, and click **Promote to Run Time** to move the file to the Run Time environment.

Note: *If changes are made to an existing e*Way Connection file, any e*Ways using the revised e*Way Connection must be restarted.*

The CICS e*Way Connection configuration parameters are organized into the following sections:

4.2.1. Connector

This section contains a set of top level parameters:

Type

Description

Specifies the connector type.

Required Values

CICS. The value always defaults to CICS for CICS connections.

Connection Transport

Description

Specifies whether the CICS e*Way will use the **SeeBeyond CICS Listener** or the **IBM CICS Transaction Gateway** as the underlying transport to send requests to and get responses from a CICS region.

Required Values

Select **SeeBeyond CICS Listener** or **Transaction Gateway**. Transaction Gateway is the default.

Connection Establishment Mode

Description

Specifies how the connection to CICS is established and closed.

- **Automatic** indicates that the connection is automatically established when the Collaboration is started, and maintains the connection as needed.
- **OnDemand** indicates that the connection is established on demand as business rules requiring a connection to the external system are performed. The connection is closed once the methods are complete.
- **Manual** indicates that the user will explicitly call the connection open and close methods in the Collaboration as business rules. Properties specified in the configuration file are loaded as default properties.

Required Values

Automatic, **OnDemand**, or **Manual**. Automatic is the default.

Connection Inactivity Timeout

Description

Specifies the timeout in milliseconds for the **Automatic** Connection Establishment Mode. If it is not set, or set to zero, the continuous connection will not timeout due to inactivity. However if the connection goes down, it will automatically attempt to reestablish the connection. If a nonzero value is specified, the connection manager monitors for any inactivity and stops the connection if it reaches the specified value.

Required Values

An integer between 0 and 864000, representing milliseconds (for example, 120000 milliseconds equals 2 minutes). The default value is 50000.

Connection Verification Interval

Description

Specifies the timeout (in milliseconds) for the **Automatic** option for the Connection Establishment Mode parameter.

- If the value is left blank or set to 0 the connection will not timeout (be brought down) due to inactivity. The connection is always kept alive. If the connection goes down, re-establishing connection is attempted automatically.
- If a non-zero value is specified, the connection manager attempts to monitor for inactivity and the connection is ended when the specified timeout value is reached.

Required Values

An integer between 0 and 864000, representing milliseconds (for example, 120000 milliseconds equals 2 minutes). The default value is 10000.

Class

Description

Specifies the class name of the CICS Client connector object.

Required Values

A valid package name. The default is com.stc.eways.cics.CicsClientConnector.

Property.Tag

Description

Specifies the data source identity. This parameter is required by the current EBobConnectorFactory.

Required Values

A valid data source package name.

4.2.2. CICS Gateway

These parameters are specific to the CICS Transaction Gateway (CTG). This section contains the following parameters for CICS Java Gateway setup:

Url

Description

Specifies the remote or local Gateway with which to connect.

Required Values

A valid remote or local Gateway (node name or IP address).

Port

Description

Specifies the TCP/IP port with which to connect, that is, the port where CTG is running.

Required Values

An integer ranging from 1 to 864000. The default value is 2006.

SSL KeyRing Class

Description

Specifies the full classname of the SSL KeyRing class.

Required Values

A valid full classname.

SSL KeyRing Password

Description

Specifies the PASSWORD for the encrypted KeyRing class.

Required Values

A valid password for the SSL KeyRing class.

4.2.3. SeeBeyond CICS Listener

These parameters are specific to the SeeBeyond CICS Listener. This section contains a set of top level parameters:

Host

Description

Specifies the name of the mainframe host with which to connect. This is always CICS.

Required Values

CICS. The value always defaults to CICS for CICS connections. The default is CICS.

Port

Description

Specifies the TCP/IP port where the SeeBeyond CICS Listener is listening. This is the port to which the CICS e*Way will connect.

Required Values

The TCP/IP port to which SeeBeyond CICS Listener is listening. The default is 3001.

SeeBeyond CICS Listener TransId

Description

Specifies the TransId of the SeeBeyond CICS Listener on the mainframe host. This is the CICS Transaction that the SeeBeyond CICS Listener is installed under.

Required Values

The valid TransId of the SeeBeyond Cics Listener

Start Type

Description

Specifies the startup type. This can be either **IC** for CICS interval control or **TD** for CICS transient data. This is the CICS Startup type for the program being executed.

Required Values

Select **IC** or **TD**.

Start Delay

Description

Specifies the hours, minutes and seconds (interval of time) to delay starting the transaction program (TP) on the CICS server for the **IC** Start Type. This field is optional but must specify all 6 digits if used.

Required Values

A 6 digit integer. All 6 digits must be given if this is specified (for example, 000000).

Listener Timeout

Description

Specifies the estimated amount of time (in milliseconds) for the SeeBeyond CICS Listener to wait for the next incoming transaction program request from the CICS e*Way.

Required Values

An integer between 1 and 864000 representing milliseconds (for example, 120000 milliseconds equals 2 minutes). The default value is 5000.

TP Timeout

Description

Specifies the amount of time the CICS e*Way will wait for the SeeBeyond CICS Listener to return results for a current transaction program request.

Required Values

An integer between 1 and 864000 representing milliseconds (for example, 120000 milliseconds equals 2 minutes). The default value is 50000.

Polling Rate

Description

Specifies the polling rate. This is the number of times the SeeBeyond CICS Listener will query the current TCP connection for incoming traffic before issuing an EXEC CICS DELAY for one second.

Required Values

- An integer between 1 and 255 representing . The default value is 5.

Transport Timeout

Description

Specifies the timeout used by both the local and host side for receive on the socket.

Required Values

An integer between 1 and 864000 representing milliseconds (for example, 120000 milliseconds equals 2 minutes). The default value is 5000.

COMMAREA Padding Character

Description

Specifies the EBCDIC code for the character used by the SBYND listener to pad the COMMAREA at the CICS server when the actual length of the payload in the COMMAREA is shorter than the length given by CommAreaLength. The default value is hexadecimal 40 - EBCDIC space.

Required Values

A charactor value coded in Hexadecimal. For example: 40 for Blanks, 00 for Low Values, FF for High Values, and so forth.

SendBufSize

Description

Specifies the Send Buffer Size for the underlying socket.

Required Values

An integer between 1 and 864000 representing bytes (for example, 10240 bytes equals 10 kilobytes). The default value is 4096.

ReceiveBufSize

Description

Specifies the Receive Buffer Size (in bytes) for the underlying socket, this is a hint.

Required Values

An integer between 1 and 864000 representing bytes (for example, 10240 bytes equals 10 kilobytes). The default value is 4096.

NoDelay

Description

Specifies whether the system can delay connections or requests. Generally, **NoDelay/True** is necessary for high-volume and/or critical transactions. In cases of low-volume and/or noncritical transactions, you can use **NoDelay/False**. Required Values

Select **TRUE** or **FALSE**. TRUE is the default.

KeepAlive

Description

Specifies whether to enable socket keep-alive checking. A setting of TRUE enables an implementation specific time period when a probe is sent to peer. The purpose of this option is to detect if the peer host has crashed.

One of three responses is expected:

1. The peer responds with the expected ACK. The application is not notified (since everything is OK). TCP will send another probe following another 2 hours of inactivity.

2. The peer responds with an RST, which tells the local TCP that the peer host has crashed and rebooted. The socket is closed.

3. There is no response from the peer. The socket is closed.

Required Values

Select TRUE or FALSE. TRUE is the default.

4.2.4. CICS Client

This section contains the following parameters for CICS Client setup:

Cics UserId

Description

Specifies the ID of the CICS user. Maximum length is eight characters.

Required Values

A valid CICS user ID, eight characters or less.

Cics Password

Description

Specifies the password for the CICS user. Maximum length is eight characters.

Required Values

A valid password for the user ID, eight characters or less.

ECI call type

Description

Specifies whether the ECI call type is Asynchronous or Synchronous.

- Synchronous Calls will wait for the transaction to complete, then return the contents of the COMMAREA.
- Asynchronous calls will *not* wait for the transaction to complete, so no data is returned.

For further detail, see the IBM publication *"CICS Family: Client/Server Programming"* (document number SC33-1435-03).

Required Values

Asynchronous or Synchronous. Synchronous is the configured default.

CICS Program

Description

Specifies the CICS program to be run on the server. Maximum length is eight characters.

Required Values

A valid CICS program name, eight characters or less.

CICS TransId

Description

Specifies the CICS TransId to be run on the server. Maximum length is four characters.

Required Values

A valid CICS TransId, four characters or less.

COMMAREA length

Description

Specifies the length (in bytes) of the communication area (COMMAREA) passed to the ECI.

Required Values

An integer in the range of 1 to 32659. The configured default is 1000.

ECI extend mode

Description

CTG specific. Specifies whether a logical unit of work is terminated at the end of a call.

- **No** (ECI_NO_EXTEND). IF the input eci_luw_token field is zero, then this is the only call for a logical unit of work. If the input eci_luw_token field is not zero, then this is the last call for the specified logical unit of work. In either case, changes to recoverable resources are committed by a CICS end-of-task syncpoint, and the logical unit of work ends.
- **Yes** (ECI_EXTENDED). If the input eci_luw_token field is zero, then this is the first call for a logical unit of work that is to be continued. If the input eci_luw_token field is not zero, then this call continues the specified logical unit of work. In either case the logical unit of work continues after the called program completes, and changes to recoverable resources remain uncommitted.

Required Values

Yes or No. The configured default is No.

ECI LUW token

Description

CTG specific. Specifies the logical unit of work to which a call belongs. This must be set to zero at the start of a logical work unit. The ECI will update the value on the first or only call of the logical work unit. If the unit of work is to be extended, this value should be used as input to all subsequent calls associated with the same logical work unit.

If the return code is not ECI_NO_ERROR and a call is ending or continuing an existing logical work unit, then this field is used to report the state of the logical work unit. If it is **zero**, the logical work unit has ended and updates have been backed out. If it is **not zero**, the value is the same as the input value. The logical work unit is continuing, and updates are still pending.

Required Values

A valid integer in the range of 0 to 1000. The configured default is 0. This is a required input and output parameter.

Message qualifier

Description

CTG specific. The ECI Message Qualifier identifies each asynchronous call if more than one call is made. This security feature-related ID is only used on the same JavaGateway that created or assigned them.

Required Values

A valid integer in the range of 0 to 1000. This is an optional input parameter.

Async Response Topic

Description

CTG specific. Specifies the default JMS topic for response Events for the asynchronous CICS program call.

Required Values

The valid name of the JMS topic.

Async Call JMS Server Host

Description

CTG specific. Specifies the host where the JMS server for the asynchronous call completion Event publishing and subscribing is running.

Required Values

The valid name of the asynchronous call JMS server host.

Async Call JMS Server Port

Description

CTG specific. Specifies the port where the JMS server for the asynchronous call completion Event publishing and subscribing is listening.

Required Values

The valid port number.

Encoding

Description

Specifies default encoding.

Required Values

The canonical name for any encoding set supported by Sun's Java Runtime Environment 1.1.8 (contained in rt.jar and i18n.jar). Examples are ASCII and Cp500 (EBCDIC). When running the CICS e*Way on z/OS, encoding should be set to "ISO-8859-1".

4.2.5. Tracing

Tracing parameters are used with the CTG implementation only. This section contains a set of top level parameters:

Level

Description

CTG specific. Specifies the level of trace information available. Options are:

- **0 - None**: no CICS Java client application tracing.
- **1 - Standard**: By default, it displays only the first 128 bytes of any data blocks (for example the COMMAREA, or network flows). This trace level is equivalent to the Gateway trace set by the ctgstart -trace option. (Can also set using System property "gateway.T.trace=on".)
- **2 - Full Debug**: By default, it traces out the whole of any data blocks. The trace contains more information about CICS Transaction Gateway than the standard trace level. This trace level is equivalent to the Gateway debug trace set by the ctgstart -x option. (Can also set using System property "gateway.T=on".)
- **3 - Exception Stacks**: It traces most Java exceptions, including exception which are expected during normal operation of the CICS Transaction Gateway. No other tracing is written. This trace level is equivalent to the Gateway stack trace set by the ctgstart -stack option. (Can also set using System property "gateway.T.stack=on".)

Required Values

An integer in the range of 0 to 3.

Filename

Description

CTG specific. Integer-set. Specifies a file location for writing the trace output. This is as an alternative to the default output on stderr. Long filenames must be surrounded by

quotation marks, for example: "trace output file.log". (Can also be set using System property "gateway.T.setTFile=xxx" where xxx is a filename.)

Required Values

A valid output file name.

Truncation Size

Description

CTG specific. Specifies the maximum size of any data blocks that is written in the trace. Specifying 0 will cause no data blocks to be written in the trace. Leave it blank if you do not want to specify truncation size. (Can also be set using System property "gateway.T.setTruncationSize=xxx" where xxx is a number.)

Required Values

An integer in the range of 0 to 864000. The configured default is 100.

Dump Offset

Description

CTG specific. Specifies the offset from which displays of any data blocks will start. If the offset is greater than the total length of data to be displayed, an offset of 0 is used. (Can also be set using System property "gateway.T.setDumpOffset=xxx" where xxx is a number.)

Required Values

An integer in the range of 0 to 864000**.**

Timing

Description

CTG specific. Specifies whether or not to display time-stamps in the trace. (Can also be set using System property "gateway.T.timing=on".)

Required Values

Off or On. The configured default is On.

Chapter 5

Implementation

This chapter includes information pertinent to implementing the Java-enabled CICS e*Way in a production environment. Several sample schemas are provided to demonstrate various implementation scenarios.

The following assumptions are applicable to this implementation: 1) The CICS e*Way has been successfully installed. 2) The executable and the configuration files have been appropriately assigned. 3) All necessary .jar files are accessible.

5.1 Using the Cobol Copybook Converter

The Cobol Copybook Converter is a build tool that takes a Cobol copybook as input and creates an ETD .ssc file. The SSC Wizard feature of the ETD Editor can be used to create Java ETDs. These ETDs can be used to map the contents of the CICS Commarea, to allow parsing and data conversion as needed.

For complete instructions on using the Copybook Convertor, see the ***Cobol Copybook Converter User's Guide.***

5.2 Sample Schemas

A number of sample schemas are provided on the installation CD-ROM or tape in the ..\samples\ewcics directory. When imported into the e*Gate Schema Designer, each sample is nearly complete with all the necessary components created for the sample. Once the component parameters are configured for the specific system, the samples are ready to run. These samples demonstrate the following:

- **The CICSJava_Sample.zip**: allocates a Commarea, issues a call to a program named STCPROGB on the mainframe, and returns a buffer with sample data, demonstrating a simple call to the program. The CICSJava_Sample schema is used in the Implementation chapter as an example for the creation of the e*Way components. These components are complete when the sample is imported, but for the purpose of explaining how the various components are created manually, they are presented from that perspective.

- **The CICSJava_os390.zip**: is the z/OS platform version of the CICSJava_Sample.zip. The sample demonstrates the CICSJava_Sample schema with the addition of handling the z/OS ASCII/EBCDIC encoding.
- **The CICS_Client_Sample.zip**: sends a data transaction to the Commarea, calls a program named QAN3GLR1. The sample demonstrates a simple request/reply, table lookup and returns a name and status or NOT FOUND if the information is unavailable.
- **The CICS_Client_Sample_os390.zip**: is the z/OS platform version of the CICS_Client_Sample.zip. The sample demonstrates the CICS_Client_Sample schema with the addition of handling the z/OS ASCII/EBCDIC encoding.
- **The CICS_Client_SubCollab_Sample.zip**: is similar to the CICS_Client_Sample except that the transaction is done as a sub-routine, demonstrating how Subcollaboration rules (sub-routines) can be called from the main Collaboration. For more information on Subcollaborations see *Subcollaboration Rules* in the *e*Gate Integrator User's Guide*.
- **CICS_Async_Sample_1.zip**: is configured with CICS Transaction Gateway as the underlying connection transport by default. It can be changed to use the SeeBeyond CICS Listener by editing the settings in the e*Way Connection configuration file. The sample simply demonstrates the Call Initiator (the Collaboration that makes the asynchronous call) launching an asynchronous call on the mainframe.
- **CICS_Async_Sample_2.zip**: is also configured for CTG. This sample builds upon the CICS_Async_Sample_1 schema. The sample demonstrates the Call Initiator launching an asynchronous call on the mainframe and proceeding to other business logic. The call is returned to the AsyncCalls pool and the result is harvested by the Call Initiator.
- **CICS_Async_Sample_3.zip**: is also configured for CTG by default. This sample builds upon the CICS_Async_Sample_2 schema. The sample demonstrates the Call Initiator launching an asynchronous call on the mainframe and proceeding to other business logic.The call is returned to the AsyncCalls pool and the result is harvested by the Call Initiator. In addition the Call Initiator publishes the return Event to a topic, making it available to other subscribers.

5.3 Importing the Sample Schemas

To import a sample schema do the following:

1. Start the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the host that you specified during installation, and enter your password.
3. You are then prompted to select a schema. Click **New**. The New Schema dialog box opens. (Schemas can also be imported or opened from the e*Gate File menu by selecting **New Schema** or **Open Schema**.)

4 Enter a name for the new schema, for example, **CICSJava_Sample**, or any name as desired.

5 To import the sample schema select **Create from Export**, and use **Find** to locate and select the sample .zip file on the CD-ROM.

The e*Gate Schema Designer opens to the new schema. You are now ready to make any configuration changes that may be necessary for this sample schema to run on your specific system.

5.3.1. Configuring the Connection Transport for a Sample Schema

To configure a sample schema to use either **SeeBeyond CICS Listener** or **CICS Transaction Gateway** as the Connection Transport, import and open the schema in the e*Gate Schema Designer and do the following:

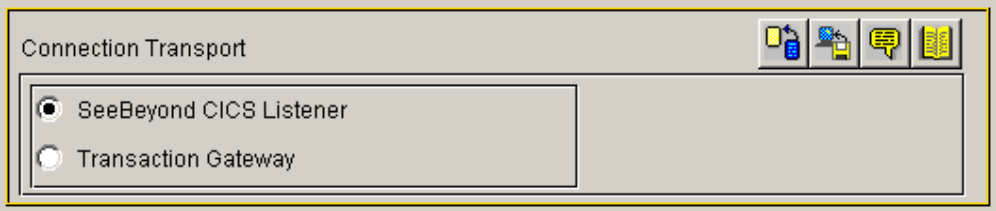
1 From the Navigator pane select the **Components** tab and select the **e*Way Connections folder**. All e*Way connections are now displayed in the Editor pane.

2 Select an e*Way Connection of type **CICS**. Double click the **e*Way Connection** to open the Properties dialog box.

3 Click the **Edit** button under the e*Way Connection Configuration File field. The Configuration Editor appears.

4 From the Connector section, Connection Transport parameter, select the appropriate underlying transport mode, SeeBeyond CICS Listener or Transaction Gateway (see Figure 34).

Figure 34 e*Way Connection Configuration Editor - Connection Transport



After selecting the desired parameters, save the current configuration. Close the **.cfg** file and select **OK** to close the e*Way Properties Window.

5.4 CICS Sample Implementation

During installation, the host and Control Broker are automatically created and configured. The default name of each is the name of the host on which you are installing the e*Gate Schema Designer GUI. To complete the implementation of the Java-enabled CICS e*Way, you will do the following:

- Make sure that the Control Broker is activated.
- In the e*Gate Schema Designer, define and configure the following as necessary:

- Inbound e*Way using **stcewfile.exe**
- Outbound e*Way using **stcewfile.exe**
- The Multi-Mode e*Way component as described in See "Multi-Mode e*Way Configuration" on page 60.
- Event Type Definitions used to package the data to be exchanged with the external system.
- Collaboration Rules to process Events.
- The e*Way Connection as described in **e*Way Connection Configuration** on page 66.
- Collaborations, to be associated with each e*Way component, to apply the required Collaboration Rules.
- The destination to which data will be published prior to being sent to the external system.

5.5 e*Way Components

The following pages explain how the sample e*Way components are created manually, first giving a walk-through of the components for the **CICSJava_Sample** schema, and then defining the components of the other sample schemas. These components are complete when the sample is imported, but are presented her for the purpose of demonstrating their creation and configuration.

5.5.1. Event Types

The CICS e*Way installation includes the file "**GenericBlob.xsc**" which represents a custom CICS Event Type template.

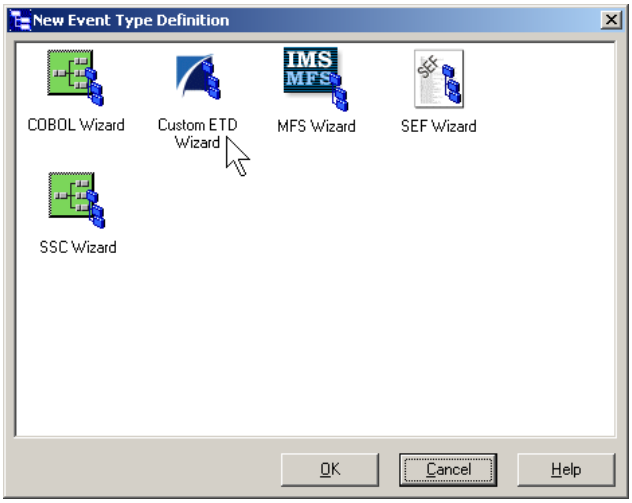
Creating an Event Type Using the Custom ETD Wizard

For the purpose of this example, the following procedure shows how to create an ETD using the Custom ETD Wizard.

1. Highlight the **Event Types** folder on the **Components** tab of the e*Gate Navigator.
2. On the palette, click the **Create a New Event Type** button to create a new **Event Type**.
3. Enter the name of the event type, then click **OK.** (For the purpose of this sample, the first Event Type is defined as "**etd_GenericBlob**.")
4. Double-click the new event type to edit its properties.
5. When the **Properties** window opens, click the **New** button. The ETD Editor opens.
6. Select **New** from the **File** menu on **Task Manager**.

7 The **Event Type Definition Wizard** opens.

Figure 35 The New Event Type Definition Wizard



8 Select the appropriate wizard. (For this Event Type, select Custom ETD Wizard.)

9 Enter a root node name. (For this example type **GenericBlobIn**.)

10 Enter a package name where the ETD Editor can place all the generated Java classes associated with the created ETD. (For this sample, use **com.stc.GenericBlob** as the package name.)

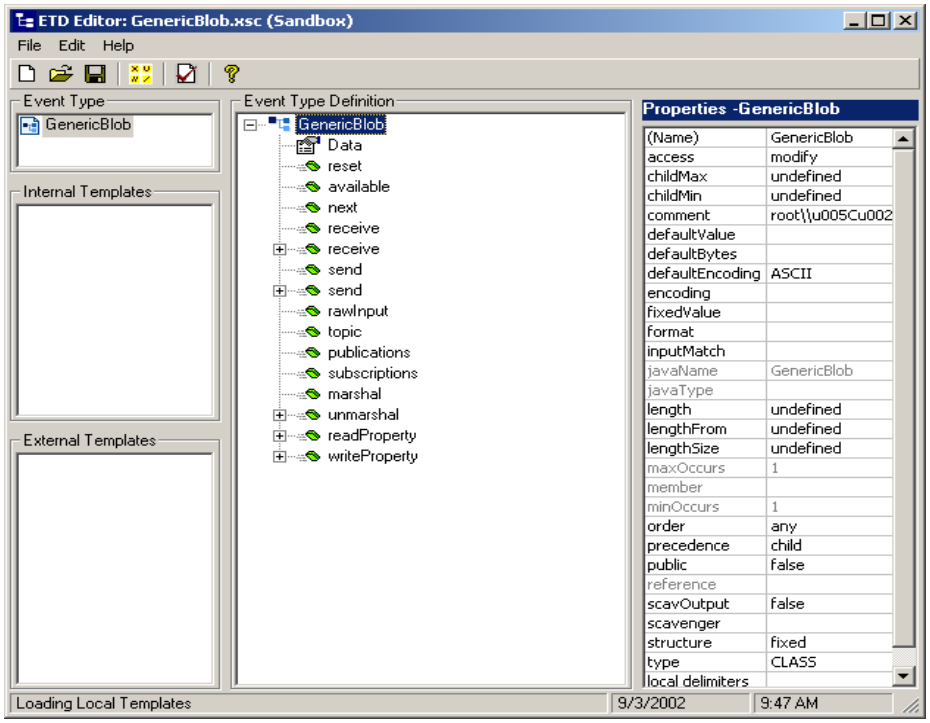
11 Click **OK** and Finish to accept the names and open the **ETD Editor**.

12 Select **GenericBlob** in the Event Type Definition pane. Change the **structure** value under **Properties** from **delim** to **fixed**.

13 Right click the **GenericBlob** root node, and select **Add Field, as Child Node**. A child node, **Field1**, is added. Triple click the **Field1** node and rename it **Data**.

14 Select the **Data** node. Change the **structure** value under **Properties** from **delim** to **fixed**, and the **order** value from **sequence** to **any**.

Figure 36 Event Type Definition Editor - GenericBlob.xsc



15 From the **File** menu, click **Compile and Save**, saving the file as **GenericBlob.xsc**.

16 From the **File** menu, click **Promote to Run Time**.

Creating an Event Type Associated with an Existing ETD

For the purpose of this example, the following procedure shows how to create an **Event Type Definition** (**ETD**) from an existing .xsc file using **cicsclient.xsc** as the input file. The **cicsclient.xsc** comes with the CICS e*Way and is used when creating all Schemas.

1 Select the **Event Types** folder on the **Components** tab of the e*Gate Navigator.

2 On the palette, click the **Create a New Event Type** button.

3 Enter the name of the **Event Type** in the **New Event Type Component** window, then click **OK**. (For this sample, the Event Type is defined as "**etd_CICSClient**.")

4 Double-click the new **Event Type** to open the ETD properties dialog box.

5 When the **Properties** dialog box opens, click the **Find** button. Browse to and select **cicsclient.xsc**.

6 Click **Apply** and **OK** to close the Event Type Properties dialog box.

5.5.2. Creating and Configuring the Component e*Ways

The first components to be created are the following e*Ways.

- **Creating the Inbound e*Way (Feeder)** on page 86
- **Creating the Outbound e*Way (Eater)** on page 87

- **Creating the Multi-Mode e*Way (CICSClient)** on page 88

The following sections provide instructions for creating each e*Way.

Creating the Inbound e*Way (Feeder)

1. Select the Navigator's **Components** tab.
2. Open the host on which you want to create the e*Ways.
3. Select the **Control Broker** that will manage the new e*Ways.
4. On the palette, click the **Create a New e*Way** button.
5. Enter the name of the new e*Way (in this case "**Feeder**"), then click **OK**.
6. Right-click the new e*Way and select **Properties** to edit its properties.
7. The e*Way Properties window opens. Click the **Find** button beneath the **Executable File** field, and select **stcewfile.exe** as the executable file.
8. Under the **Configuration File** field, click the **New** button. The Edit Settings window opens. Select the settings, as displayed in Table 3, for this configuration file.

Table 3 Configuration Parameters for the Inbound e*Way

Parameter	Value
General Settings (unless otherwise stated, leave settings as default)	
AllowIncoming	YES
AllowOutgoing	NO
Outbound Settings	Default
Poller Inbound Settings	
PollDirectory	C:Indata (input file folder)
InputFileExtension	*.fin (input file extension)
PollMilliseconds	1000
Remove EOL	YES
MultipleRecordsPerFile	YES
MaxBytesPerLine	4096
BytesPerLineIsFixed	NO
File Records Per eGate Event	1
Performance Testing	Default

9. From the **File** menu, click **Save**, saving the configuration file as **Feeder.cfg**.
10. From the **File** menu, click **Promote to Run Time**. This closes the **.**cfg file.
11. In the e*Way - Properties window, use the **Startup**, **Advanced**, and **Security** tabs to modify the default settings for each e*Way you configure.
    A. Use the **Startup** tab to specify whether the e*Way starts automatically, or restarts after abnormal termination or due to scheduling, and so forth.

B Use the **Advanced** tab to specify or view the activity and error logging levels, as well as the Event threshold information.

C Use **Security** to view or set privilege assignments.

12 Select **OK** to close the e*Way Properties window.

Creating the Outbound e*Way (Eater)

1 Select the Navigator's **Components** tab.

2 Open the host on which you want to create the e*Ways.

3 Select the **Control Broker** that will manage the new e*Ways.

4 On the palette, click the **Create a New e*Way** button.

5 Enter the name of the new e*Way (in this case "**Eater**"), then click **OK**.

6 Select the new e*Way, right-click and select **Properties** to edit its properties.

7 When the **e*Way Properties** window opens, click the **Find** button beneath the **Executable File** field. Select **stcewfile.exe** as the executable file.

8 Under the **Configuration File** field, click the **New** button. The **Edit Settings** window opens. Select the following settings for this configuration file.

Table 4 Configuration Parameters for the Outbound e*Way

Parameter	Value
General Settings (unless otherwise stated, leave settings as default)	
AllowIncoming	NO
AllowOutgoing	YES
Outbound Settings	
OutputDirectory	C:\DATA
OutputFileName	output%d.dat
MultipleRecordsPerFile	NO
MaxRecordsPerFile	10000
AddEOL	YES
Poller Inbound Settings	Default
Performance Testing	Default

9 From the **File** menu, click **Save**, saving the file as **Eater.cfg**, and click **Promote to Run Time**, to move the file to the run-time environment. This closes the Edit Settings window.

10 In the e*Way - Properties window, use the **Startup**, **Advanced**, and **Security** tabs to modify the default settings for the e*Way.

11 Use **Security** to view or set privilege assignments.

12 Click **OK** to close the e*Way Properties window.

Creating the Multi-Mode e*Way (CICSClient)

1. Select the Navigator's **Components** tab.
2. Open the host on which you want to create the e*Way.
3. Select the **Control Broker** that will manage the new e*Way.
4. On the palette, click the **Create a New e*Way** button.
5. Enter the name of the new e*Way (in this case, "**CICSClient**"), then click **OK**.
6. Right-click the new e*Way and select **Properties** to edit its properties.
7. When the e*Way Properties window opens, click the **Find** button beneath the **Executable File** field, and select **stceway.exe** as the executable file.
8. To edit the JVM Settings, select **New** (or **Edit** if you are editing the existing .cfg file) under Configuration file.

   See **Multi-Mode e*Way Configuration** on page 60 for details on the parameters associated with the Multi-Mode e*Way.
9. From the **File** menu, click **Save** to save the **.**cfg file as **CICSClient.cfg**, and click **Promote to Run Time** to move the file to the run-time environment.
10. In the e*Way Properties window, use the **Startup**, **Advanced**, and **Security** tabs to modify the default settings for each.
    - A Use the **Startup** tab to specify whether the e*Way starts automatically, restarts after abnormal termination or due to scheduling, etc.
    - B Use the **Advanced** tab to specify or view the activity and error logging levels, as well as the Event threshold information.
    - C Use **Security** to view or set privilege assignments.
11. Click **OK** to close e*Way Properties window.

5.5.3. Creating the e*Way Connection

The e*Way Connection configuration file contains the connection information along with the information needed to communicate using CICS.

To create and configure a New e*Way Connection

1. Select the **e*Way Connection** folder on the **Components** tab of the e*Gate Navigator.
2. On the palette, click the **Create a New e*Way Connection** button.
3. Enter the name of the e*Way Connection (for this sample, "**eWc_CICSClient**"), then click **OK.**
4. Double-click the new e*Way Connection to edit its properties.
5. The e*Way Connection Properties window opens. Select **CICS** from the **e*Way Connection Type** drop-down menu.

6 Enter the **Event Type "get" interval** in the dialog box provided. 10000 milliseconds is the configured default. The "get" interval is the intervening period at which, when subscribed to, the e*Way connection is polled.

7 Under e*Way Connection Configuration File, click the **New** button.

8 The e*Way Connection Editor opens. Select the following parameters listed in Table 4. For more information on the CICS e*Way Connection parameters, see **e*Way Connection Configuration** on page 66.

Table 5 e*Way Connection Configuration Parameters

Parameter	Value
connector (unless otherwise stated, leave settings as default)	
type	CICS
class	com.stc.eways.cics.CicsClientConnector
CICS Gateway	
Port	8888
AddEOL	YES
CICS Client	Default
CICS Program	STCPROGB
COMMAREA length	1000
ECI extend mode	No
ECI LUW token	0
Message qualifier	0
Encoding	cp500
Performance Testing	
Level	0
Filename	CICSJava_Trace.txt
Truncation Size	100
Dump Offset	0
Timing	On

9 Save the .cfg file (**eWc_CICSClient.cfg**), and from the **File** menu, click **Promote to Run Time**.

5.5.4. Creating Intelligent Queues

The next step is to create and associate the **IQ Manager** for the CICS e*Way. The IQ Manager governs the exchange of information between components within the e*Gate system, providing non-volatile storage for data as it passes from one component to another. IQs use IQ Services to transport data. IQ Services provide the mechanism for moving Events between IQs, handling the low-level implementation of data exchange (such as system calls to initialize or reorganize a database).

Typically, the IQ Manager is set to type **SeeBeyond JMS** and IQs use the **STC_JMS_IQ** service. This is not an option when running the CICS e*Way on z/OS. The CICS e*Way, when running on z/OS must use the **SeeBeyond Standard IQ Manager** Type and the **STC_Standard IQ Service**.

The CICSJava_Sample schema uses the **SeeBeyond JMS** IQ Manager and the **STC_JMS_IQ** IQ Service.

To create and modify the JMS IQ Manager for the CICS e*Way

1. Select the Navigator's **Components** tab.
2. Open the host on which you want to create the **JMS Queue Server**.
3. Open a **Control Broker**.
4. Click the **Create a New IQ Manager** button. Enter a name for the IQ Manager (for this case, "**localhost_iqmgr**").
5. Right-click the **IQ Manager** and select **Properties**. The **IQ Manager Properties** dialog box opens.
6. From the **IQ Manager Type** field drop-down list box select **SeeBeyond JMS**.
7. Click **New** under the **Configuration File** field to set parameters, or select the Use Default Configuration option to accept default settings. For this schema select the Use Default Configuration option.
8. On the **Start Up** tab select **Start automatically** and **Restart after abnormal termination**. Set the **Number of retries** to **10** and set the **Retry interval every** value to **10 minutes**.
9. Click **OK** to close the **IQ Manager Properties** dialog box.

To create and modify an Intelligent Queue for the CICS e*Way

1. From the Navigator's **Components** pane, open the participating host.
1. Select the **IQ Manager**.
2. On the palette, click the **Create a New IQ** button.
3. Enter the name of the new **IQ** (in this case "**IQ1**"), then click **OK.**
4. Double-click the new **IQ** to edit its properties.
5. On the **General** tab, specify the **Service** and the **Event Type Get Interval**. When using a SeeBeyond JMS Type IQ Manager the IQ Service defaults to STC_JMS_IQ.

   The default **Event Type Get Interval** of 100 Milliseconds is satisfactory for the purposes of this initial implementation.
6. On the **Advanced** tab, make sure that **Simple publish/subscribe** is checked under the **IQ behavior** section.
7. Click **OK** to close the **IQ Properties** window
8. For this schema, repeat steps 1 through 8 to create an additional IQ (for this sample, "**IQ2"**).

Note: *When running the CICS e*Way on z/OS, always select an IQ Manager Type of SeeBeyond Standard. The SeeBeyond JMS IQ Manager is not available for z/OS.*

5.5.5. Creating Collaboration Rules

The next step is to create the Collaboration Rules that will extract and process selected information from the source Event Type defined above, according to its associated Collaboration Service. The **Default Editor** can be set to either **Monk** or **Java**.

From the **Schema Designer Task Bar,** select **Options** and click **Default Editor**. The default should be set to **Java**.

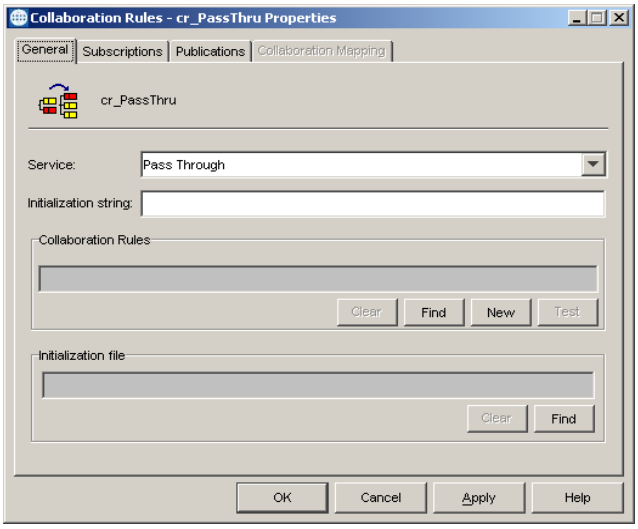
The sample schema calls for the creation of two Collaboration Rules files.

- **cr_PassThru** (Pass Through)
- **cr_CICSClient** (Java)

cr_PassThru (Pass Through)

1. Select the Navigator's **Components** tab in the e*Gate Schema Designer.
2. In the Navigator, select the **Collaboration Rules** folder.
3. On the palette, click the **Create New Collaboration Rules** button.
4. Enter the name of the new Collaboration Rule Component (for this case "**PassThru**"), then click **OK**.
5. Double-click the new Collaboration Rules Component. The **Collaboration Rules Properties** window opens.

Figure 37 Collaboration Rules Properties - Pass Through

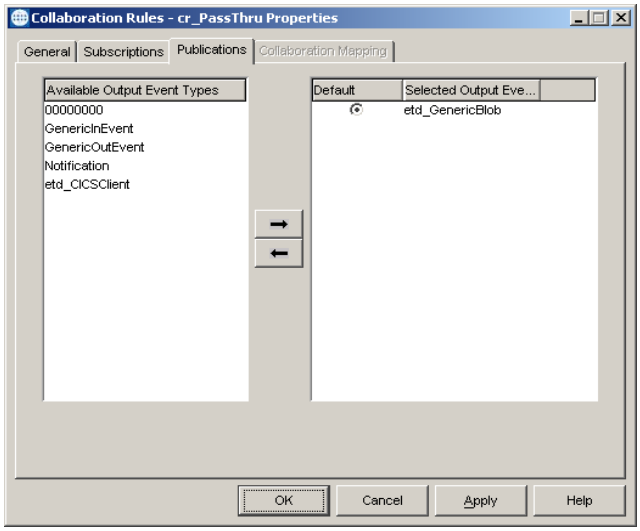


6. The **Service** field defaults to **Pass Through**.
7. Go to the **Subscriptions** tab. Select **etd_GenericBlob** (see **Creating an Event Type Using the Custom ETD Wizard** on page 83 to see how the **etd_GenericBlob** was

created) under **Available Input Event Types**, and click the right arrow to move it to **Selected Input Event Types**. The box under **Triggering Event** should be checked.

8 Go to the **Publications** tab. Select **etd_GenericBlob** under **Available Output Event Types**, and click the right arrow to move it to **Selected Output Event Types**. Make sure that **etd_Blob** is selected as the default.

Figure 38 Collaboration Properties



9 Click **OK** to close the **Collaboration Rules - cr_PassThru Properties** window.

cr_CICSClient (Java)

1 Select the Navigator's **Components** tab in the e*Gate Schema Designer.

2 In the **Navigator**, select the **Collaboration Rules** folder.

3 On the palette, click the **Create New Collaboration Rules** button.

4 Enter the name of the new Collaboration Rule, then click **OK** (for this case, use **CICSClient**).

5 Double-click the new Collaboration Rules Component to edit its properties. The Collaboration Rules Properties window opens.

6 From the **Service** field drop-down box, select **Java**. The **Collaboration Mapping** tab is now enabled, and the **Subscriptions** and **Publications** tabs are disabled.

7 In the **Initialization string** field, enter any required initialization string that the Collaboration Service may require. This field can be left blank.

8 Select the **Collaboration Mapping** tab.

9 Using the **Add Instance** button, create instances to coincide with the Event Types. For this sample, do the following:

10 In the Instance Name column, enter **CICSOut** for the instance name.

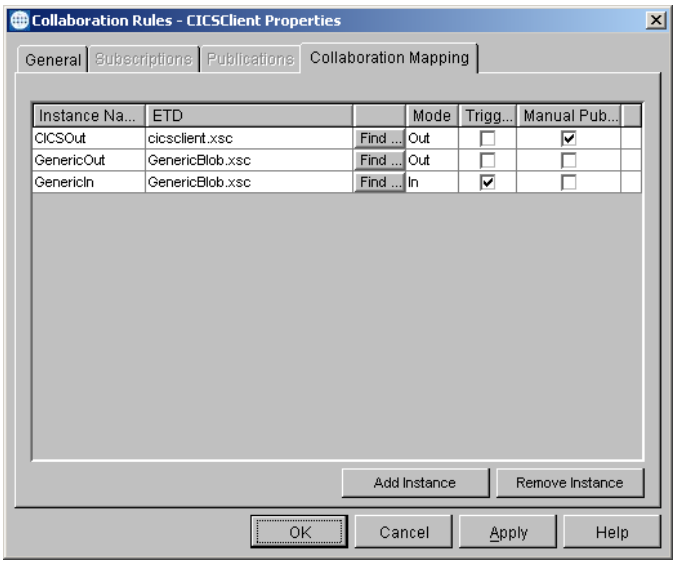
11 Click **Find,** navigate to and double-click **cicsclient.xsc**. This adds **cicsclient.xsc** to the **ETD** column for this instance.

12 In the **Mode** column, select **In** from the drop–down list box. To access the drop-down list box, click the right portion of the **Mode** field for this instance.

13 In the **Trigger** column, make sure that the checkbox is cleared (no trigger).

14 In the **Manual Publish** column, make sure the checkbox is selected.

Figure 39 Collaboration Rules - Collaboration Mapping



15 Repeat steps 9–13 using the following values:

- Instance Name — **GenericOut**
- ETD — **GenericBlob.xsc**
- Mode — **Out**
- Trigger — clear
- Manual Publish - clear

16 Repeat steps 9–13 again using the following values:

- Instance Name — **GenericIn**
- ETD — **GenericBlob.xsc**
- Mode — **In**
- Trigger — select
- Manual Publish - clear

17 Select the **General** tab, under the Collaboration Rule box, select **New**. The **Collaboration Rules Editor** opens.

18 Expand to full size for optimum viewing, expanding the Source and Destination Events as well. The following section describes the setting up the Collaboration rules for **CICSClient** using the Java Collaboration Rules Editor.

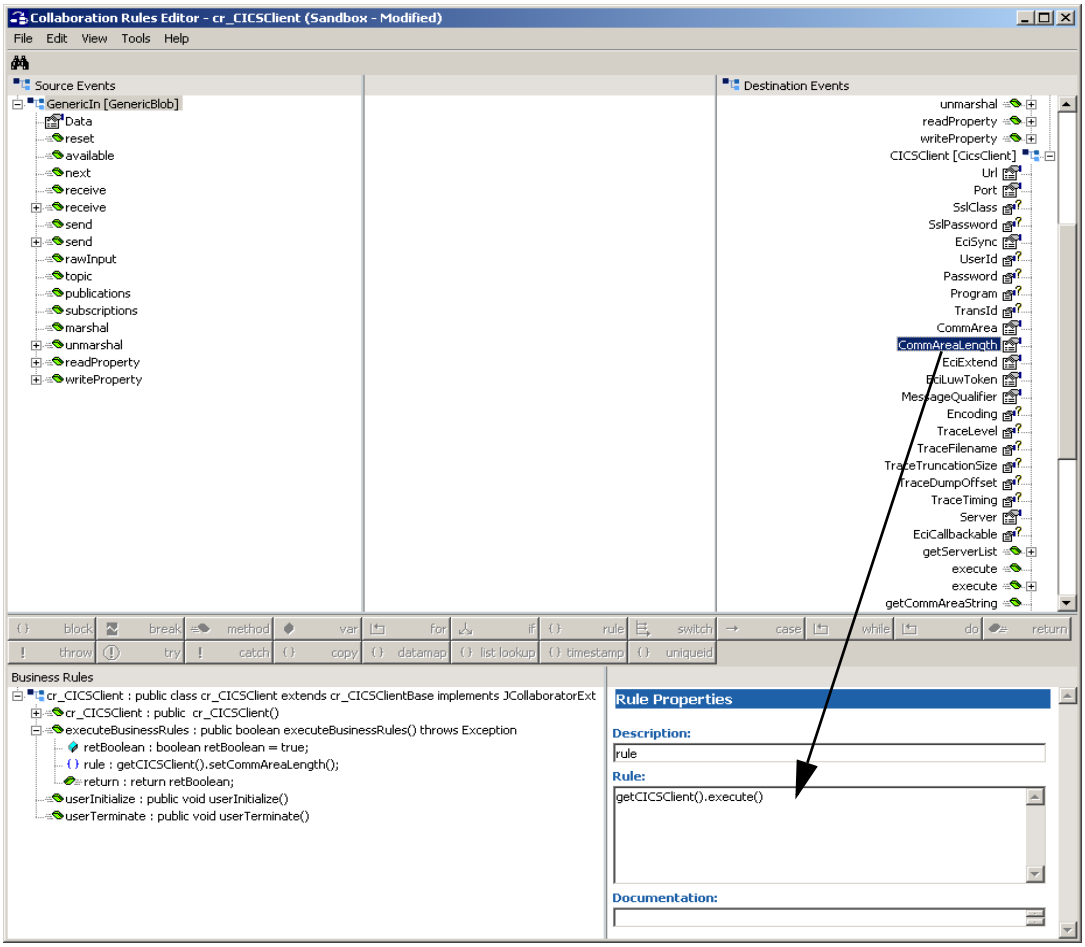
Creating the Collaboration Rules Class

The section is given as an example of how to create the Collaboration Rules Class using the Java Collaboration Rules Editor. The completed Collaboration Rules .xpr file is included with the sample schema on the CD. The following section gives a number of examples that demonstrate how these rules were setup. Refer to the completed class, **CICSClient.class** when completing the Collaboration Rules Properties.

Each **rule** is created by clicking the **rule** button on the Business Rules toolbar or by "dragging and dropping" a node or method from the Source Events pane onto a node or method in the Destination Events pane. For more information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide.*

1. The Java Collaboration Rules Editor opens from the Collaboration Rules Properties dialog box when the Collaboration Rules field, New or Edit button is clicked. Expand to full size for optimum viewing, expanding the Source and Destination Events as well.
2. Select **retBoolean** in the **Business Rules** pane. All of the user–defined business rules are added as part of this method.

Figure 40 The Collaboration Rules Editor



3 The **first rule** under retBoolean is created by dragging **CommAreaLength** from the Destination Events command node into the Rule Properties, Rule field (see Figure 40). Place the Cursor in the last set of parentheses and enter **50** as the parameter to create the following code:

```
getCICSClient().setCommAreaLength(50)
```

4 The **second rule** is created by dragging **Data** from the Source Events command node into the Rule Properties, Rule field to create the following code:

```
getCICSClient().setCommArea()
```

Place the cursor in the last set of parentheses and enter the following:

```
new String (getGenericIn().getData().getBytes("cp500"), "ISO-8859-1").getBytes())
```

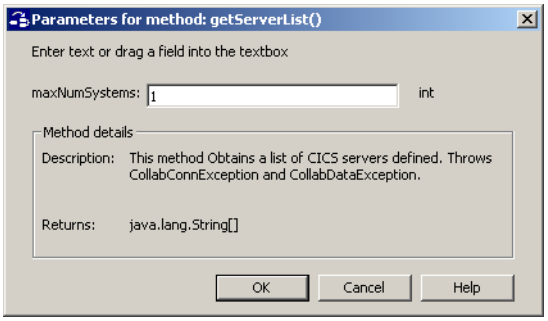
to create the following code:

```
getCICSClient().setCommArea(new String (getGenericIn().getData().getBytes("cp500"), "ISO-8859-
1").getBytes())
```

5 To create **third rule** under retBoolean, drag the **getServerList** method under CICSClient on the Destination Events command node into the Rule Properties, Rule field. When prompted for the maxNumSystems int (see Figure 41) enter **1** and click **OK** to create the following code:

```
getCICSClient().getServerList(1)
```

Figure 41 Parameters for method - getServerList



6 The **forth rule** under retBoolean is created by dragging the **execute** method under CICSClient on the Destination Events command node into the Rule Properties, Rule field.

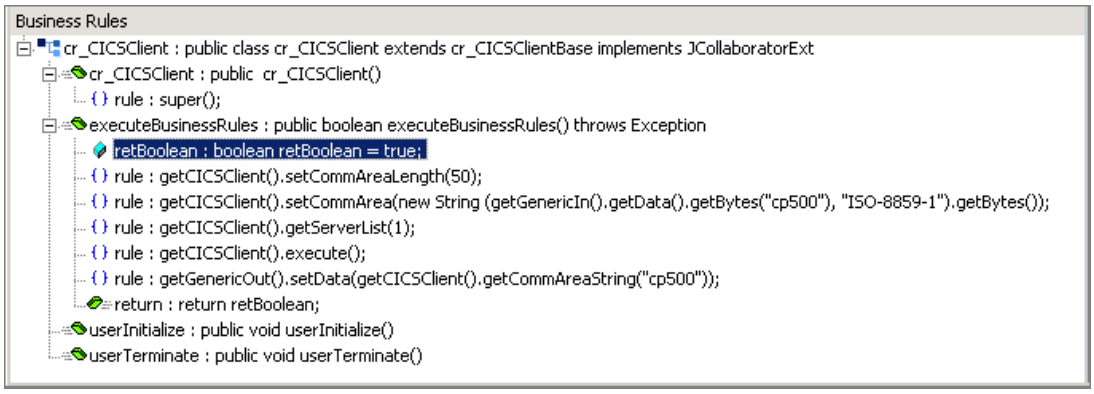
7 To create the **fifth** rule drag **Data** under GenericOut on the Destination Events command node into the Rule Properties, Rule field. Drag the second getCommAreaString method under CICSClient on the Destination Events command node into the last set of parentheses. When prompted for the encoding parameter value, enter **"cp500"** to create the following code:

```
getGenericOut().setData(getCICSClient().getCommAreaString("cp500"))
```

8 From the Tools menu, click **Options**. Add **stccics.jar** to the Classpath and verify that all necessary **.jar** files are included.

9 When all the business logic has been defined (see Figure 42), the code can be compiled by selecting **Compile** from the **File** menu. The **Save** menu opens, provide a name for the .xpr file.

Figure 42 Business Rules - cr_CICSClient



5.5.6. Creating Collaborations

Collaborations are the components that receive and process Event Types and forward the output to other e*Gate components or to an external. Collaborations consist of the Subscriber, which "listens" for Events of a known type (sometimes from a given source) and the Publisher, which distributes the transformed Event to a specified recipient.

To create the CICIS_Multi_Mode Collaboration

1. In the e*Gate Schema Designer, select the Navigator's **Components** tab.
2. Open the host on which you want to create the Collaboration.
3. Select a **Control Broker.**
4. Select an e*Way to assign the Collaboration (for this sample, "**CICSClient**").
5. On the palette, click the **Create a New Collaboration** button.
6. Enter the name of the new Collaboration, then click **OK.** (For the sample, "**col_CICSclient**")
7. Double-click the new **Collaboration** to edit its properties. The **Collaboration Properties** dialog box opens.
8. From the **Collaboration Rules** drop-down list box select the Collaboration Rules file that you created previously (for the sample, "**CICSclient**").
9. In the **Subscriptions** area, click **Add** to define the input Event Types to which this Collaboration will subscribe.
   - A From the **Instance Name** field drop-down list box, select the Instance Name that you previously defined "**GenericIn**."
   - B From the **Event Type** drop-down list box, select the **Event Type** that you previously defined "**etd_Blob**."
   - C From the **Source** drop-down list box, select the source (for this sample "**eWc_CICSClient**").
10. In the **Publications** area, click **Add** to define the output **Event Types** that this Collaboration will publish.

A From the **Instance Name** drop-down list box, select the **Instance Name** that you previously defined "**GenericOut**."

B From the **Event Types** drop-down list box, select the **Event Type** that you previously defined "**etd_Blob**."

C Select the publication destination from the **Destination** drop-down list box. In this case, it should be "**IQ2**."

D The value in the **Priority** column defaults to **5**.

11 In the **Publications** area, click **Add** to add an additional instance.

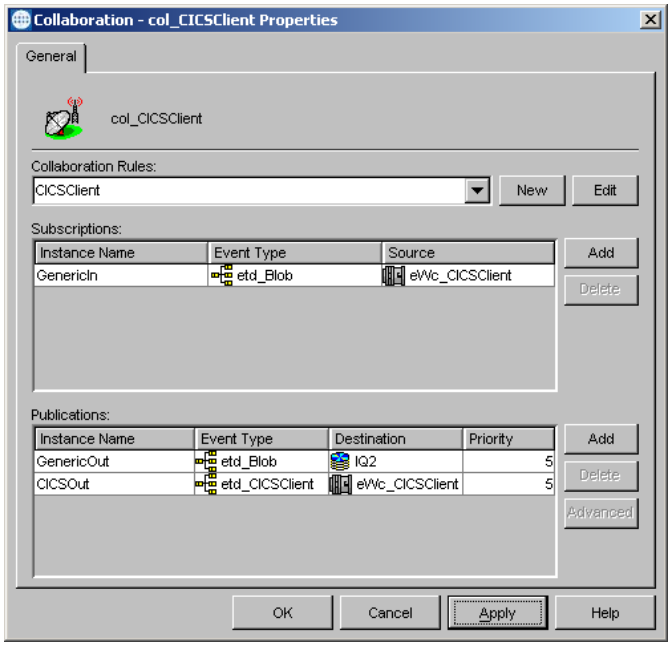
A From the **Instance Name** drop-down list box, select the **Instance Name** that you previously defined "**CICSOut**."

B From the **Event Types** drop-down list box, select the **Event Type** that you previously defined "**etd_CICSClient**."

C Select the publication destination from the **Destination** drop-down list box. In this case, it should be "**eWc_CICSClient**."

D The value in the **Priority** column defaults to **5**.

Figure 43 Collaboration Properties - col_CICSClient



12 Click the **Apply** button and click **OK** to close the Collaboration Properties dialog box.

To create the Inbound e*Way Collaboration

1 In the e*Gate Schema Designer, select the Navigator's **Components** tab.

2 Open the host on which you want to create the Collaboration.

3 Select a **Control Broker.**

4 Select the **Feeder** e*Way to assign its Collaboration.

5 On the palette, click the **Create a New Collaboration** button.

6 Enter the name of the new Collaboration (for the sample, "**col_Feeder**") then click **OK.**

7 Double-click the new Collaboration to edit its properties. The **Collaboration Properties** dialog box opens.

8 From the **Collaboration Rules** drop-down list box, select the Collaboration Rules file that you created previously (for the sample, "**PassThru**").

9 In the **Subscriptions** area, click **Add** to define the input Event Types to which this Collaboration will subscribe.

A From the **Event Type** drop-down list box, select the **Event Type** that you previously defined "**etd_Blob**."

B Select the **Source** from the **Source** drop-down list box. In this case, it should be **<External>**.

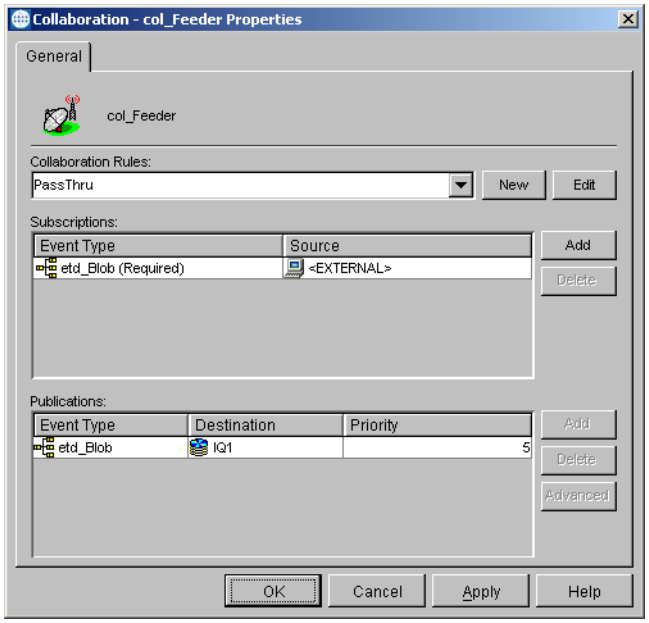
10 In the **Publications** area, click **Add** to define the output **Event Types** that this Collaboration will publish.

A From the **Event Types** list, select the **Event Type** that you previously defined "**etd_Blob**."

B Select the publication destination from the **Destination** list. In this case, it should be "**IQ1**."

C The value in the **Priority** column defaults to **5**.

Figure 44 Collaboration Properties_col_Feeder

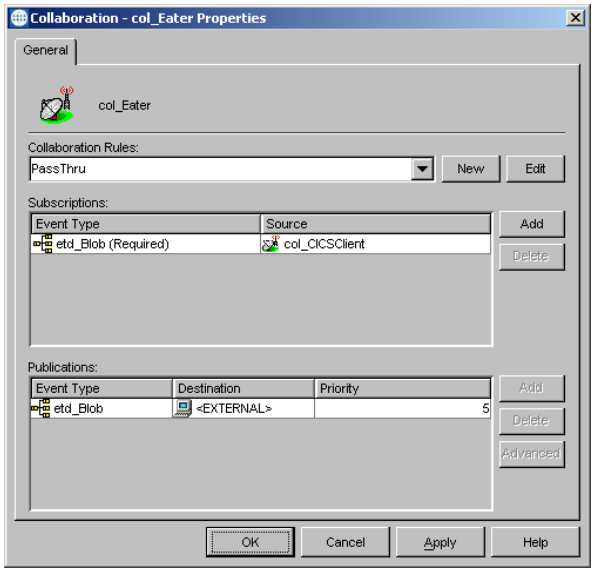


11 Click the **Apply** button and click **OK** to close the Collaboration Properties dialog box.

To create the Outbound e*Way Collaboration

1. In the e*Gate Schema Designer, select the Navigator's **Components** tab.
2. Open the host on which you want to create the Collaboration.
3. Select a **Control Broker.**
4. Select the **Eater** e*Way to assign its Collaboration.
5. On the palette, click the **Create a New Collaboration** button.
6. Enter the name of the new Collaboration (for this sample, "**col_Eater**"), then click **OK**.
7. Double-click the new **Collaboration** to edit its properties.
8. From the **Collaboration Rules** drop-down list box, select the Collaboration Rules file that you created previously (for the sample, "**PassThru**").
9. In the **Subscriptions** area, click **Add** to define the input Event Types to which this Collaboration will subscribe.
   A. From the **Event Type** drop-down list box, select the **Event Type** that you previously defined "**etd_Blob**."
   B. Select the **Source** from the **Source** list. In this case, it should be "**col_CICSClient**."
10. In the **Publications** area, click **Add** to define the output **Event Types** that this Collaboration will publish.
    A. From the **Event Types** list box, select the **Event Type** that you previously defined "**etd_Blob**."
    B. Select the publication destination from the **Destination** list. In this case, it should be **<External>**.
    C. The value in the **Priority** column defaults to **5**.

Figure 45 Collaboration Properties_col_Feeder



11 Click the **Apply** button and click **OK** to close the Collaboration Properties dialog box.

5.6 CICS Sample Schemas

The following sections defines the components of the remaining sample schemas. It is assumed that the reader has a basic understanding of the e*Way components and how they are created. For more information on creating and configuring e*Way components see the *e*Gate Integrator User's Guide*. The following sections describe the various CICS sample schemas:

- **The CICSJava_Sample Schema** on page 80
- **The CICS_Client_Sample Schema** on page 100
- **The CICSJava_os390 and CICS_Client_Sample_os390 Schemas for z/OS** on page 109
- **The CICS_Client_SubCollab_Sample Schema** on page 110
- **Asynchronous Call Handling Samples** on page 114

5.7 The CICS_Client_Sample Schema

The CICS_Client_Sample demonstrates a simple request/reply, table lookup, and returns a name and status or NOT FOUND if information is unavailable. The sample sends a data transaction to the Commarea and calls a program named QAN3GLR1.

Note: *The components of the sample schema are created when the schema is imported and only require changes to the configuration parameters of the e*Ways and e*Way Connections for your specific system. The following section describes how the sample components are created manually.*

5.7.1 Creating the e*Ways

The **CICS_Client_Sample** uses three e*Ways, Feeder (Inbound - stcewfile.exe), Eater (outbound - stcewfile.exe) and CICSClient (Multi-mode - stceway.exe).

Configuring the File e*Ways

The File e*Ways, **Feeder** and **Eater** use the executable file "**stcewfile**", set in the e*Way properties. The Configuration file for the e*Way is set as displayed in Table 6and Table 7.

Table 6 Feeder e*Way Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	YES
AllowOutgoing	NO
Outbound (send) settings - Set as directed, otherwise leave as default.	
Poller (inbound) settings - Set as directed, otherwise leave as default.	
PollDirectory	C:\INDATA\CICS_Sample
InputFileMask	*.fin
PollMilliseconds	1000
RemoveEOL	YES
MultipleRecordsPerFile	YES
MaxBytesPerLine	4096
BytesPerLineIsFixed	NO
File Records Per eGate Event	1
Performance Testing - Set as directed, otherwise leave as default.	

Table 7 Eater e*Way Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	NO
AllowOutgoing	YES
Outbound (send) settings - Set as directed, otherwise leave as default.	
OutputDirectory	C:\DATA\CICS_Sample
OutputFileName	CICS_output%d.dat
MultipleRecordsPerFile	YES
MaxRecordsPerFile	10000
AddEOL	YES
Poller (inbound) settings - Set as directed, otherwise leave as default.	
Performance Testing - Set as directed, otherwise leave as default.	

Configuring the Multi-Mode e*Way

The Multi-Mode e*Way, **CICSClient**, uses the executable file "**stceway**", set in the e*Way's properties.

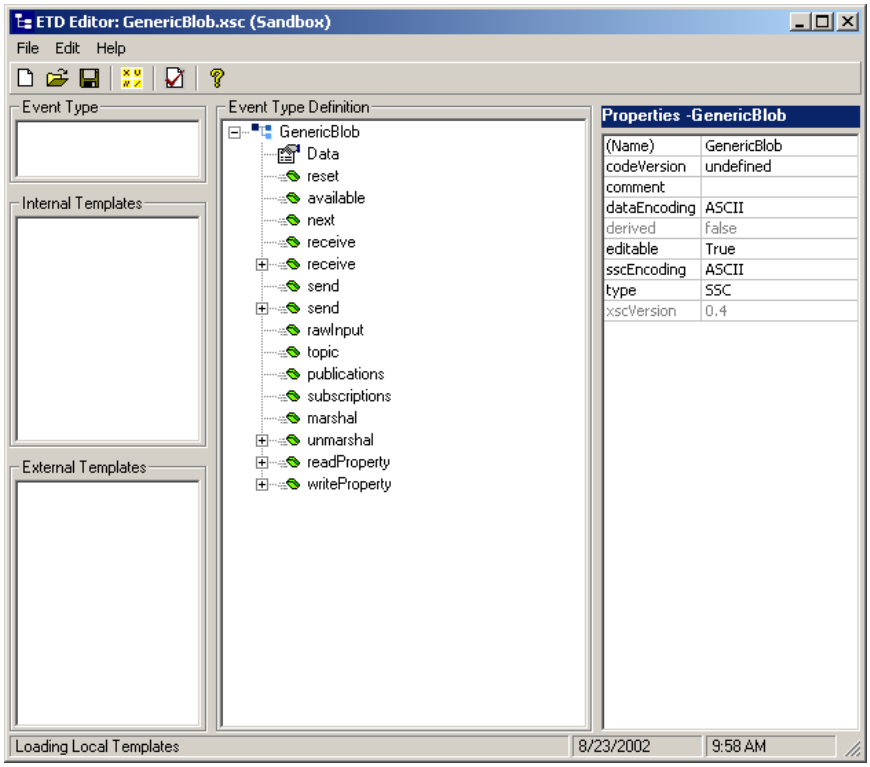
For the purposes of this sample the configuration file for the Multi-Mode e*Way can be saved as default.

For more information on the Multi-Mode e*Way configuration settings see the *e*Gate Integrator User's Guide*.

5.7.2 Creating the ETDs

The **CICS_Client_Sample** uses two Event Types, **etd_CICSClient** using **cicsclient.xsc** (see **The CICSClient ETD** on page 46), and **etd_GenericBlob** (GenericBlob.xsc) as seen in Figure 46.

Figure 46 ETD Editor - etd_GenericBlob



5.7.3 Configuring the IQs

Open the IQ Manager Properties and select **SeeBeyond Standard** as the IQ Manager Type. Create two STC_Standard IQs, IQ1 and IQ2, both set for Simple publish and subscribe (see **Creating Intelligent Queues** on page 89 for more information).

5.7.4 Creating the e*Way Connections

One e*Way Connection, **eWc_CICSClient** is created for the CICS_Client_Sample. The e*Way Connection Type, viewed in the e*Way Connection Properties, is **CICS**. The default **Event Type "get" interval** setting of 100 milliseconds is sufficient for this sample. To set the Configuration parameters for the **eWc_CICSClient** e*Way Connection (see Table 8), click on the **New/Edit** button under the e*Way Connection Configuration File field. Edit the settings specific to your system (see **Creating the e*Way Connections** on page 102 and **e*Way Connection Configuration** on page 66). From the **File** menu, click **Save**, and **Promote to Run Time**.

Table 8 eWc_CICSClient e*Way Connection Parameters

e*Way Configuration Parameters	Feeder
connector Settings - Set as directed, otherwise leave as default.	
type	CICS
Connection Transport	SeeBeyond CICS Listener
Connection Establishment Mode	Automatic
Connection Inactivity Mode	3000
Connection Verification Interval	10000
class	
Property.Tag	
CICS Gateway - Set as directed, otherwise leave as default.	
URL	
Port	
SSL KeyRingClass	
SSL KeyRing Password	
SeeBeyond CICS Listener - Set as directed, otherwise leave as default.	
Host	
Port	3009
SeeBeyond CICS Listener TransId	STCL
Start Type	IC
Start Delay	000000
Listener Timeout	5000
TP Timeout	50000
Polling Rate	25
Transport Timeout	5000
COMMAREA Padding Character	40
SendBufSize	4096
ReceiveBufSize	4096
NoDelay	TRUE
KeepAlive	TRUE
CICS Client - Set as directed, otherwise leave as default.	
CICS UserId	

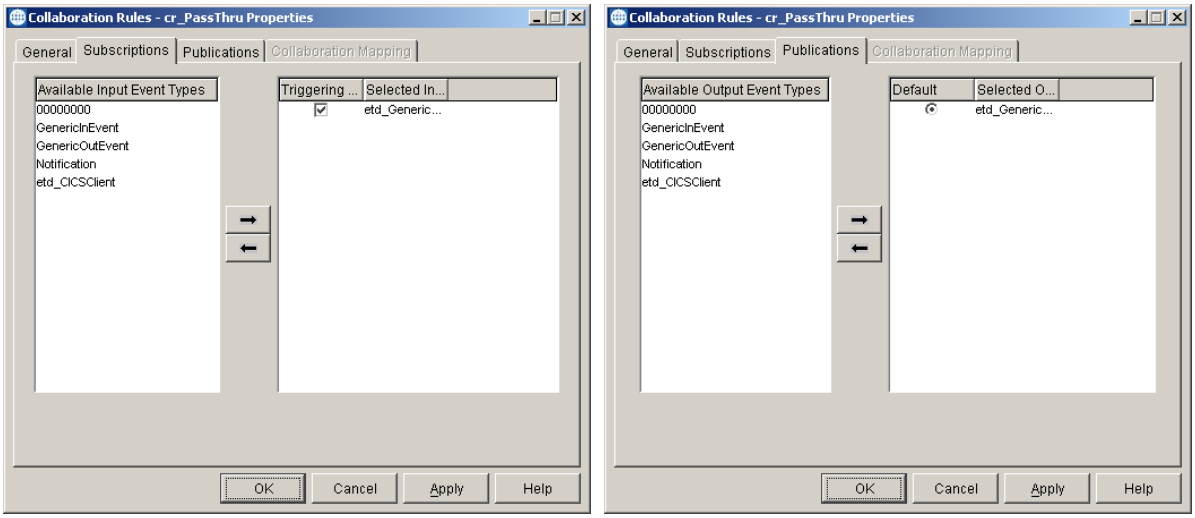
e*Way Configuration Parameters	Feeder
CICS Password	
ECI call type	Synchronous
CICS Program	QAN3GLR1
CICS TransId	
COMMAREA Length	50
ECI extend mode	No
ECI LUW token	0
Message qualifier	0
Encoding	cp500

5.7.5 Creating the Collaboration Rules

The **CICS_Client_Sample** includes the **cr_PassThru** (Pass Through) and **crCICSClient** (Java) Collaboration Rules. Create the Collaboration Rules as displayed. Business logic for the Java Collaboration Rules is defined using the Collaboration Rules Editor. See **Creating Collaboration Rules** on page 91 for more information.

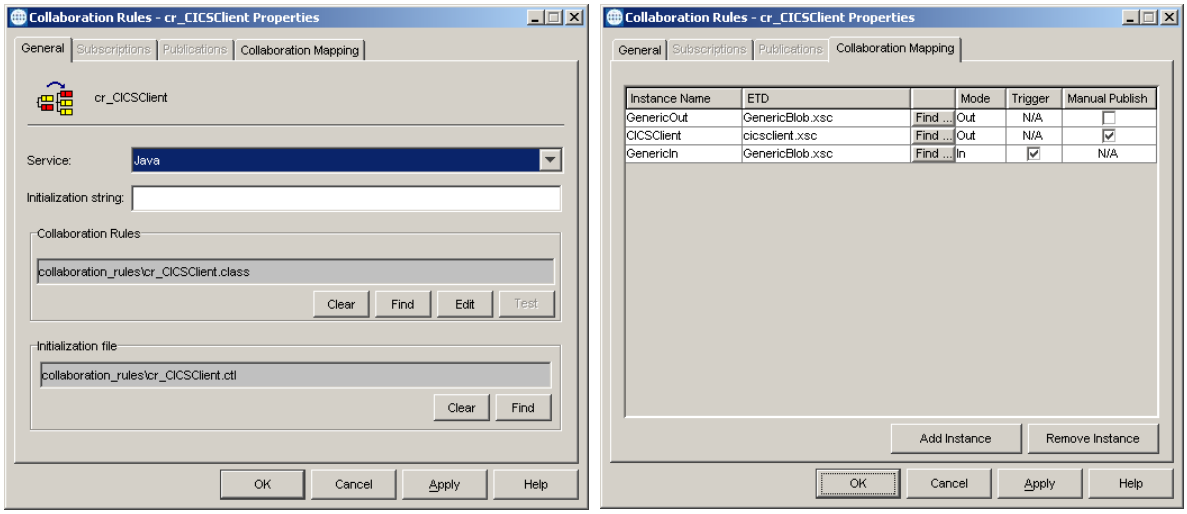
The **cr_PassThru** Collaboration Rules Properties dialog box Subscriptions and Publications tabs appear as they do in Figure 47 when complete.

Figure 47 Collaboration Rules Properties - cr_PassThru



The **cr_CICSClient** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs appear as they do in Figure 48 when complete.

Figure 48 Collaboration Rules Properties - cr_CICSClient



From the General tab of the **cr_CICSClient** Collaboration Rules Properties dialog box, click **Edit** or **New** under the Collaboration Rules field. The Collaboration Rules Editor opens.

5.7.6 Business Rules for the cr_CICSClient.class

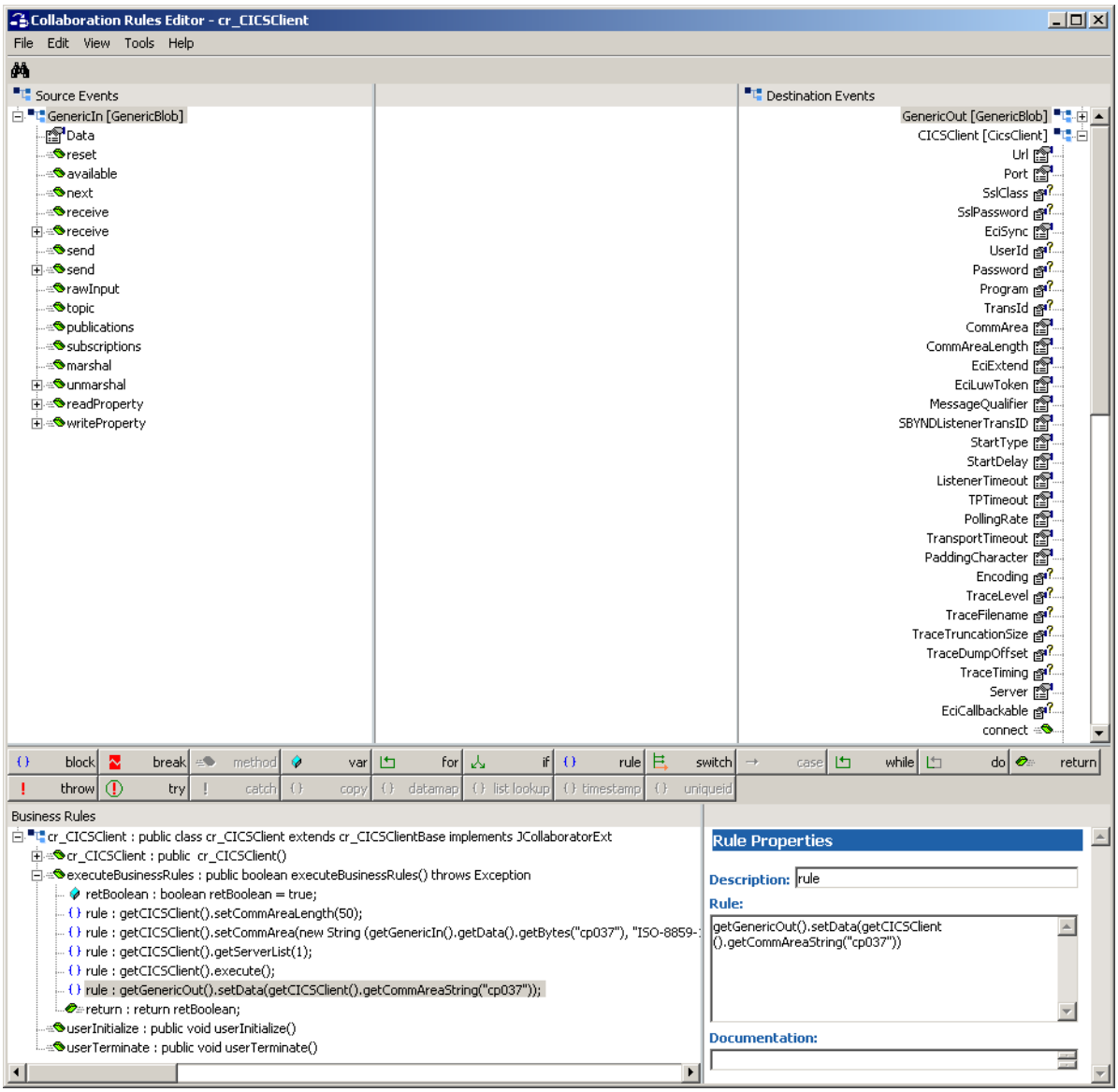
The business logic of the Collaboration is defined using the Collaboration Rules Editor (see Figure 50). A Java Collaboration Rule is created by designating one or more source Events and one or more destination Events and then setting up rules governing the relationship between fields in the Event instances.

Each **rule** is created by clicking the **rule** button on the Business Rules toolbar or by "dragging and dropping" a node or method from the Source Events pane onto a node or method in the Destination Events pane. For more information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide.*

The **cr_CICSClient** Collaboration Rules (see Figure 49) are created as follows:

Figure 49 cr_CICSClient.class Business Rules

Figure 50 Collaboration Rules Editor - cr_CICSClient.class



1 The first **rule**, under retBoolean in the Business Rules window (see Figure 50), is created by "dragging and dropping" **CommAreaLength** under CICSClient on the Destination Events command node into the Rule Properties, Rule field. Place the cursor in the last set of parentheses and type **50** to create the following code:

```
getCICSClient().setCommAreaLength(50)
```

2 To create the second **rule** drag **CommArea** under CICSClient on the Destination Events command node, into the Rule Properties, Rule field. Place the cursor in the last set of parentheses and type in the following:

```
new String (getGenericIn().getData().getBytes("cp037"), "ISO-8859-1").getBytes("ISO-8859-1")
```

The completed code for the second rule should appear as follows:

```
getCICSClient().setCommArea(new String (getGenericIn().getData().getBytes("cp037"), "ISO-8859-
1").getBytes("ISO-8859-1"))
```

3 For the third **rule**, drag the **getServerList** method under CICSClient, on the Destination Events command node into the Rule Properties, Rule window. When prompted for a **MaxNumSystems Int** enter "**1**".

4 For the forth **rule**, drag the **execute** method under CICSClient, on the Destination Events command node into the Rule Properties, Rule window.

5 To create the fifth **rule** drag **Data,** under GenericOut on the Destination Events command node, into the Rule Properties, Rule field. Drag the **getCommAreaString** method with only the encoding parameter into the last set of parentheses. When prompted for an encoding parameter value, type "**cp037**" and click **OK**, to create the following code:

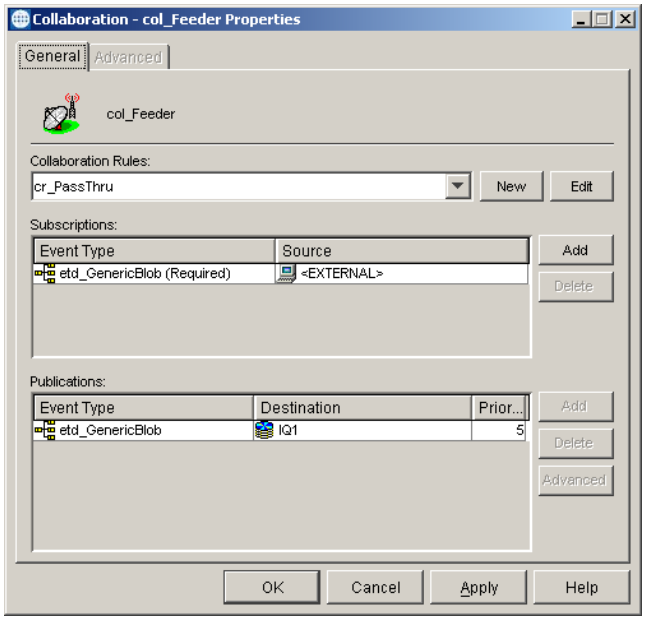
```
getGenericOut().setData(getCICSClient().getCommAreaString("cp037"))
```

6 When the business logic is complete, select **Save** and **Compile** from the **File** menu. If the file compiles successfully, select **Promote** from the **File** menu to promote the file to the run-time environment.

5.7.7 Creating the Collaborations

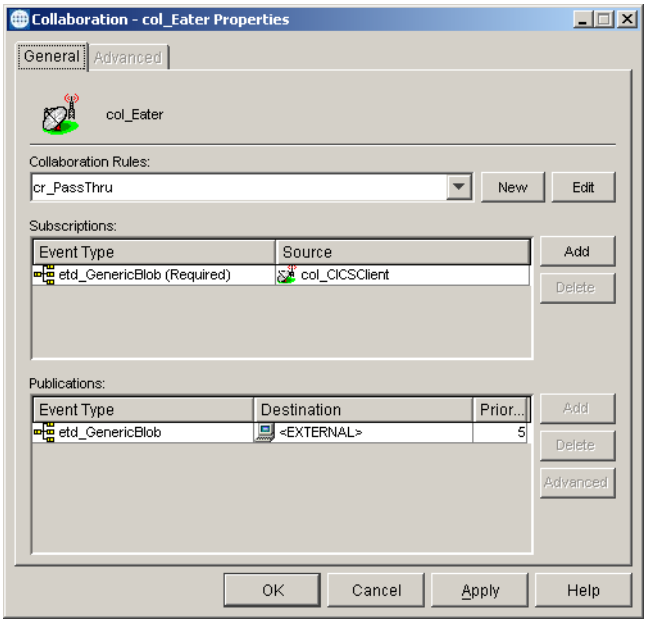
The **Feeder** e*Way Collaboration, named **col_Feeder**, appears as it is displayed in Figure 51 when complete.

Figure 51 Collaboration Properties - col_Feeder



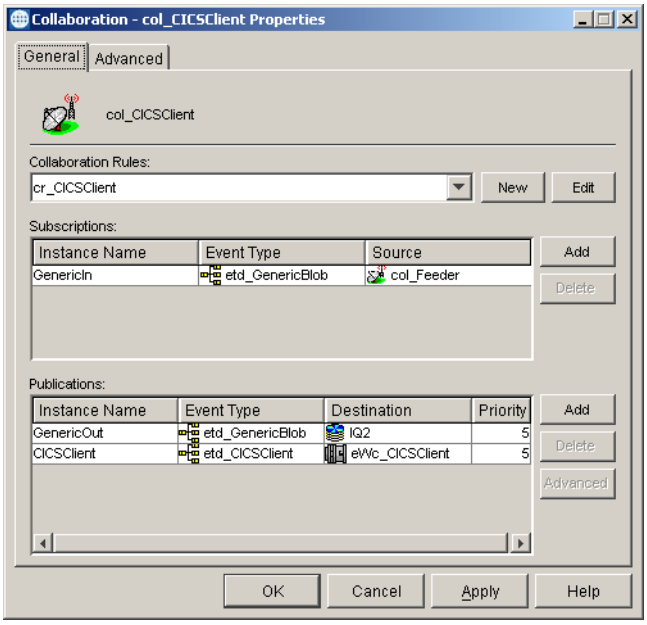
The Collaboration for the **Eater** e*Way, named **col_Eater,** appears as it is displayed in Figure 52 when complete.

Figure 52 Collaboration Properties - col_Eater



The **CICSClient** e*Way Collaboration, named **col_CICSClient**, appears as it is displayed in Figure 53 when complete.

Figure 53 Collaboration Properties - col_CICSClient



5.8 The CICSJava_os390 and CICS_Client_Sample_os390 Schemas for z/OS

The **CICSJava_os390** and **CICS_Client_Sample_os390** schemas are nearly identical to the non-z/OS platform versions, with a few exceptions.

1. Both the CICS Java_os390 and the CICS_Client_Sample_os390 schemas use the SeeBeyond Standard IQ Manager rather than the SeeBeyond JMS IQ Manager. Therefore the IQs for the z/OS samples are STC_Standard rather than STC_JMS_IQ. The CICS e*Way on z/OS must use the SeeBeyond Standard IQ Manager type. (See **Creating Intelligent Queues** on page 89 for more information regarding IQs.)
2. For the **CICS_Client_Sample_os390** has two additional Business Rules have been added in the z/OS version of the Collaboration (see Figure 54) to provide inbound and outbound encoding for z/OS.

   For Java Collaborations to compile properly on z/OS, the following actions must be performed in the userInitialize section in the Business Rules pane of the Java Collaboration. Perform these initializations once and only once.

   Converting incoming EBCDIC data to ASCII

   To convert incoming EBCDIC data (Java codepage "cp037") to ASCII (Java codepage "ISO-8859-1"), code the jCollabController in the userInitialize section as:

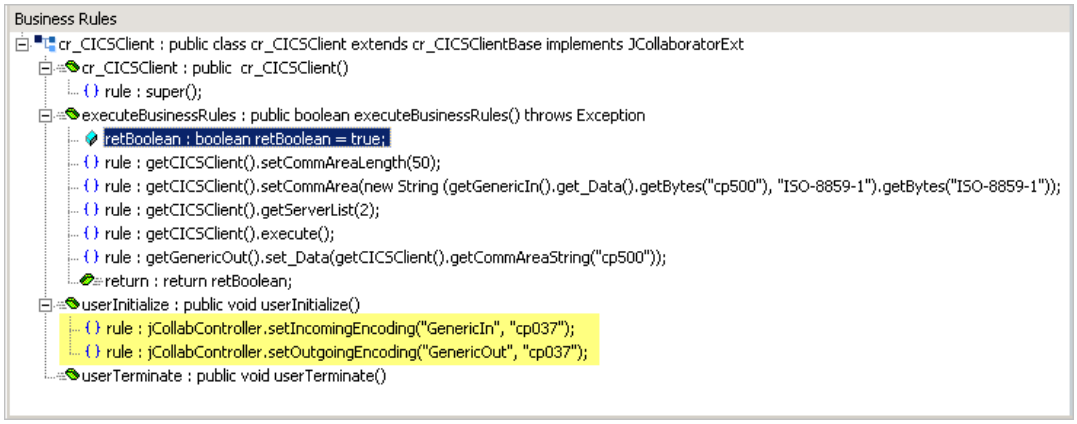
   ```
   jCollabController.setIncomingEncoding("In1","cp037");
   ```

   Converting outgoing ASCII data to EBCDIC

   To convert outgoing ASCII data (Java codepage "ISO-8859-1") to EBCDIC (Java codepage "cp037"), code the jCollabController in the userInitialize section as:

   ```
   jCollabController.setOutgoingEncoding("In2","cp037");
   ```

Figure 54 Business Rules - CICS_Client_Sample_os390



For further information on Java Collaborations on z/OS, see the *e*Gate Integrators Installation Guide.*

5.9 The CICS_Client_SubCollab_Sample Schema

Depending on how it is used, every Collaboration Rules file is either a Root Collaboration Rule, invoked by e*Gate itself, or a Subcollaboration Rule, invoked by another Collaboration Rule.

A Collaboration Rule, when used as a Root Collaboration Rule, is like a main program. When used as a Subcollaboration Rule, it is like a subroutine. A Subcollaboration Rule allows you to reuse a valuable piece of work in another context without having to reinvent it or reconstruct it from scratch.Typically, a Subcollaboration Rule takes care of details or special-purpose parsings and transformations, allowing the parent Collaboration Rule to be simpler and more general.

Subcollaboration Rules can nest to indefinite depth, limited only by system resources (memory, stack, and so forth). A Subcollaboration Rule is invoked programmatically, whereas a Root Collaboration Rule, like a main program, is called by the e*Way itself.

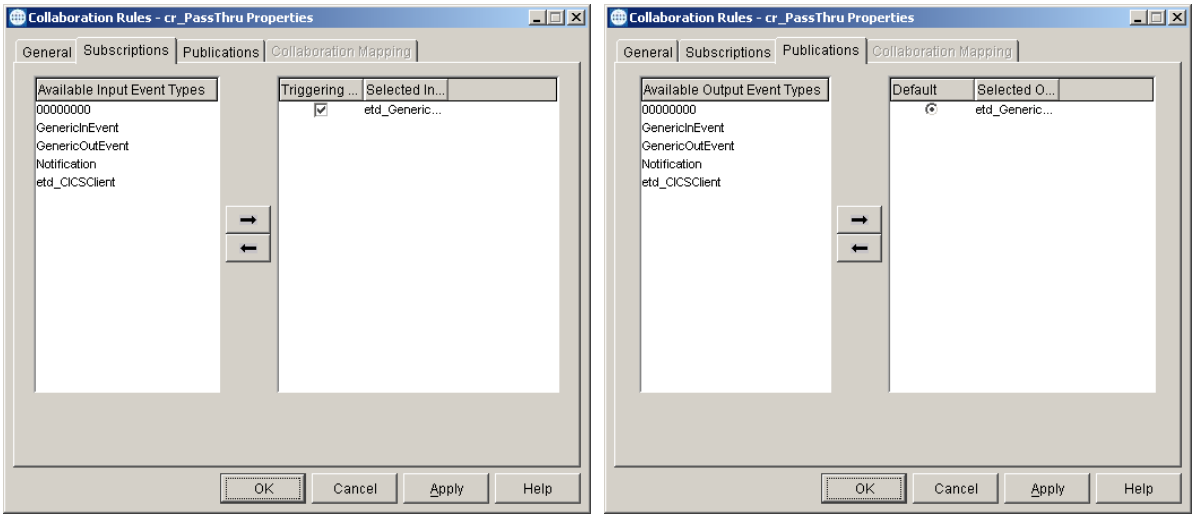
The CICS_Client_SubCollab_Sample executes the same operations as the CICS_Client_Sample, except that the Business Rules are carried out as a Subcollaboration. The components of the CICS_Client_SubCollab_Sample are identical to the CICS_Client_Sample presented previously, with the exception of the components presented in the following sections. For more information on creating and using Subcollaborations see the *Subcollaboration Rules* section of the *e*Gate Integrator User's Guide.*

5.9.1 Creating the Collaboration Rules

The **CICS_Client_SubCollab_Sample** includes the **cr_PassThru** (Pass Through), **crCICSClient** (Java) and **crCICSClient_Tran (Java)** Collaboration Rules. Create the Collaboration Rules as displayed. Business logic for the Java Collaboration Rules is defined using the Collaboration Rules Editor. See **"Creating Collaboration Rules" on page 91** for more information.

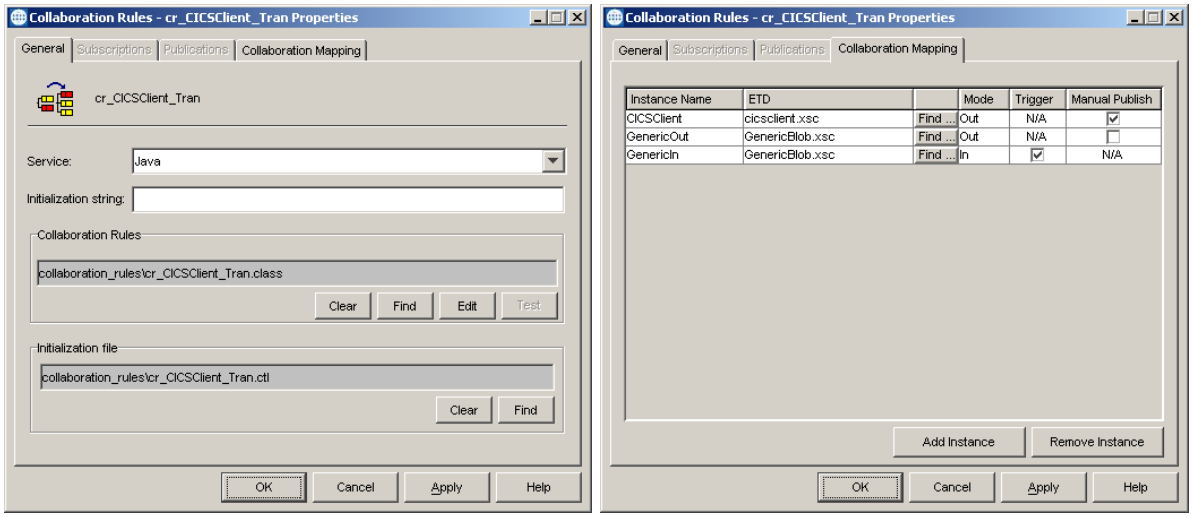
The **cr_PassThru** Collaboration Rules Properties dialog box Subscriptions and Publications tabs appear as they do in Figure 55 when complete.

Figure 55 Collaboration Rules Properties - cr_PassThru



The **cr_CICSClient** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs appear as they do in Figure 56 when complete.

Figure 56 Collaboration Rules Properties - cr_CICSClient_Tran

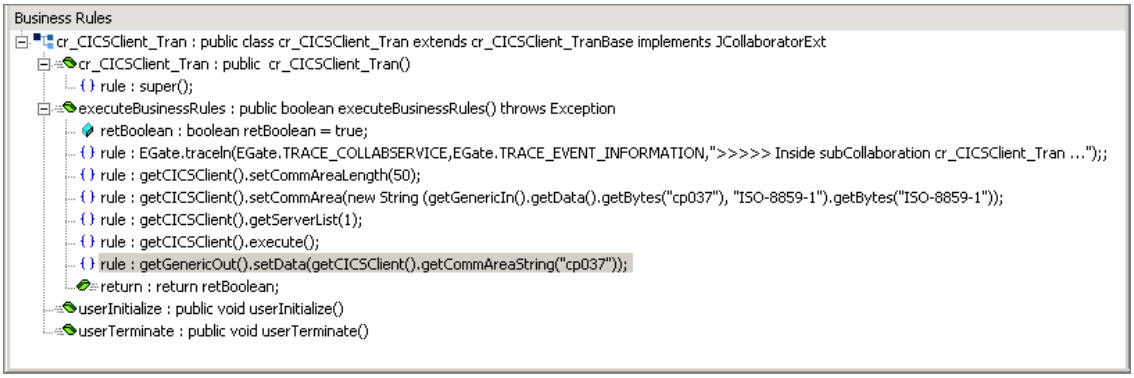


From the General tab of the cr_CICSClient_Tran Collaboration Rules Properties dialog box, click **Edit** or **New** under the Collaboration Rules field. The Collaboration Rules Editor opens.

5.9.2 Creating the Business Rules

The Business Logic of the Collaboration is defined using the Collaboration Rules Editor. To create and use cr_CICSClient_Tran as a Subcollaboration, create cr_CICSClient_Tran.class using the Collaboration Rules Editor. cr_CICSClient_Tran.class is similar to cr_CICSClient.class created for the CICS_Client_Sample (see **Business Rules for the cr_CICSClient.class** on page 105).

Figure 57 cr_CICSClient_Tran.class Business Rules



Creating the cr_CICSClient.class Collaboration Rules

The **cr_CICSClient_Tran** Collaboration Rule is called from the parent Collaboration Rule file, **cr_CICSClient.class** as a Subcollaboration (see **Figure 58 on page 113**). The **cr_CICSClient.class** Collaboration Rules are created as follows:

Each **rule** is created by clicking the **rule** button on the Business Rules toolbar or by "dragging and dropping" a node or method from the Source Events pane onto a node or method in the Destination Events pane. For more information on using the Java Collaboration Rules Editor and creating Subcollaborations, see the *e*Gate Integrator User's Guide.*

1. Start the Collaboration Rules Editor for cr_CICSClient by clicking the **Edit** or **New button** under the Collaboration Rules field in the cr_CICSClient Collaboration Rules Properties dialog box. The Collaboration Rules Editor opens.
2. Select the **cr_CICSClient** node and click the **var** button on the Business Rules toolbar to add a variable. In the Variable Properties pane, select **JSubCollabMapInfo** as the Type. Enter **SubCollabMapInfo** as the Name, and **null** as the Initial Value. This variable keeps track of the mapping information of the Subcollaboration Rule.
3. Select **retBoolean** in the Business Rules pane and click the **rule** button on the toolbar. The first **rule**, under retBoolean in the Business Rules window, is a trace statement used for debugging purposes. To create the trace statement type the following in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE,EGate.TRACE_EVENT_INFORMATION,">>>>> Inside Main
Collaboration cr_CICSClient ...")
```

4. To create the second rule, **Invoke subCollaboration**, type the following code in the Rule Properties, Rule field:

```
this.subCICSTranMethod()
```

   Type **Invoke subCollaboration** into the Rule Properties Description field. This rule is used to call the method that is defined in step 6.

5. Within the **userInitialize()** method, add a **rule** and type the following code into the Rule Properties, Rule field:

```
SubCollabMapInfo = this.jCollabController.createSubCollabMapInfo("cr_CICSClient_Tran", this)
```

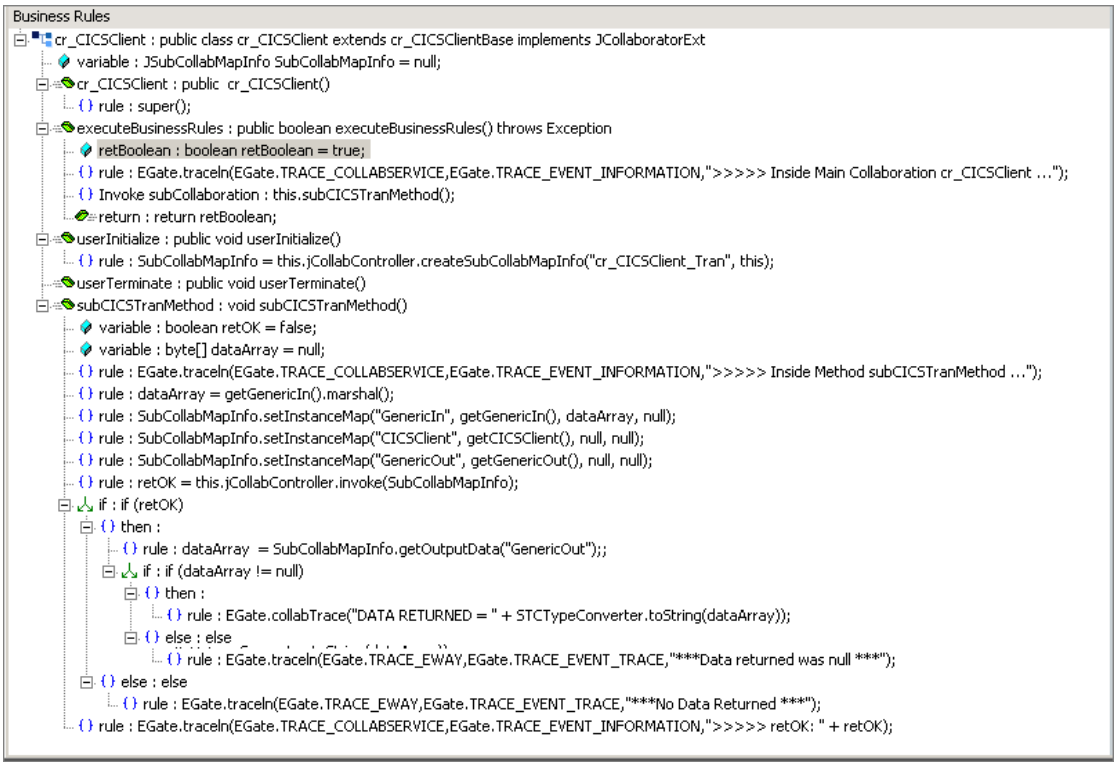
This creates the mapping information for the Subcollaboration Rule and assigns it to the variable added in step 2.

6 Select the **userTerminate( )** method in the Business Rules pane and click the **method** button to add a new method. In the Method Properties, enter **subCICSTranMethod** as the description and name, and **void** as the return type.

7 Add a variable of type **boolean**, name **retOK**, and initial value **false**. This variable checks on the Subcollaboration Rule's success.

8 Add a variable of type **byte** (Array selected), name **dataArray**, and initial value **null**. This variable harvests the output from the Subcollaboration Rule.

Figure 58 CICS_Client_SubCollab_Sample cr_CICSClient.class Business Rules



9 The next **rule** is a trace statement, created by typing the following code into the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE,EGate.TRACE_EVENT_INFORMATION,">>>>> Inside Method
subCICSTranMethod ...")
```

10 The next **rule** is created by typing the following into the Rule Properties, Rule field:

```
dataArray =
```

and dragging **marshal()**, under GenericIn on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
dataArray = getGenericIn().marshal()
```

11 The next **rule** calls **setInstanceMap**() for the Subcollaboration Rule's inbound Event Type instance to populate the instance. To create the rule, type the following code into the Rule Properties, Rule field:

```
SubCollabMapInfo.setInstanceMap("GenericIn", getGenericIn(), dataArray, null)
```

12 The next two **rules** call **setInstanceMap()** for each of the Subcollaboration Rule's harvestable outbound Event Type instances**,** to harvest the instance.

A To create the first of these **rules**, type the following code into the Rule Properties, Rule field:

```
SubCollabMapInfo.setInstanceMap("CICSClient", getCICSClient(), null, null)
```

B To create the second of these **rules**, type the following code into the Rule Properties, Rule field:

```
SubCollabMapInfo.setInstanceMap("GenericOut", getGenericOut(), null, null)
```

13 The next **rule** assigns a value to the variable defined in step 7. Type the following code into the Rule Properties, Rule field:

```
retOK = this.jCollabController.invoke(SubCollabMapInfo)
```

14 To create the **if** expression, click the **if** button. Type **retOK** in the if Properties, Condition field.

15 To create the next **rule**, select the **then** expression and click the **rule** button. Type the following code into the Rule Properties, Rule field:

```
dataArray  = SubCollabMapInfo.getOutputData("GenericOut");
```

16 Create another **if** expression. Type **dataArray != null** in the if Properties, Condition field.

17 To create the next **rule**, select the next **then** expression and click the **rule** button. Type the following trace statement into the Rule Properties, Rule field:

```
EGate.collabTrace("DATA RETURNED = " + STCTypeConverter.toString(dataArray));

getGenericOut().setData(STCTypeConverter.toString(dataArray));
```

18 To create the next **rule**, select the **else** expression, click the **rule** button, and type the following trace statement into the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_EWAY,EGate.TRACE_EVENT_TRACE,"***Data returned was null ***")
```

19 To create the next **rule**, select the first **if** expressions **else** expression, click the **rule** button and type the following trace statement into the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_EWAY,EGate.TRACE_EVENT_TRACE,"***No Data Returned ***")
```

20 The last **rule** is created by selecting the first **if** expression and clicking the **rule** button. Type the following trace statement into the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE,EGate.TRACE_EVENT_INFORMATION,">>>>> retOK: " + retOK)
```

21 When the business logic is complete, select **Save** from the File menu, an then select **Compile** from the File menu. If the file compiles successfully, select **Promote** from the **File** menu to promote the file to the run-time environment.

5.10 Asynchronous Call Handling Samples

The **CICS_Async_Sample_1**, **CICS_Async_Sample_2**, and **CICS_Async_Sample_3** sample schemas demonstrate asynchronous call handling. The samples increment upon each other, with sample 2 building upon sample 1 and sample 3 building upon sample 2. The components of the CICS_Async_Sample_3 are presented in the following

sections and can be referred to for the components of samples 1 and 2. These sample schemas are configured with CICS Transaction Gateway as the underlying connection transport by default. To use the samples with SeeBeyond CICS Listener as the underlying transport the e*Way Connection configuration parameters must be edited for the sample (see **Configuring the Connection Transport for a Sample Schema** on page 82).

- **CICS_Async_Sample_1**: demonstrates the Call Initiator (the Collaboration that makes the asynchronous call) launching an asynchronous call on the mainframe and proceeding to other business logic.
- **CICS_Async_Sample_2**: demonstrates the Call Initiator launching an asynchronous call on the mainframe and proceeding to other business logic. The call is returned to the AsyncCalls pool and the result is harvested by the Call Initiator.
- **CICS_Async_Sample_3**: demonstrates the Call Initiator launching an asynchronous call on the mainframe and proceeding to other business logic.The call is returned to the AsyncCalls pool and the result is harvested by the Call Initiator. In addition the Call Initiator publishes the return Event to a topic, making it available to other subscribers.

5.10.1. The CICS_Async_Sample Schemas

The CICS_Async_Sample_1 Schema

In the **CICS_Async_Sample_1** schema, the inbound (**feeder**) e*Way picks up either a data message or command message from a local folder and publishes it to a JMS IQ (**IQ1**). The **CICSClient e*Way** subscribes to **IQ1** and tests whether the message is a data message or command message. If it is a **command** message then the command is followed. If it is a **data** message, it launches an asynchronous call to a remote CICS program. This program (**EC01**, a time stamp program on the mainframe) has been configured in the e*Way Connection configuration file. The **CICSClient e*Way** also sends a string message to the JMS IQ (**IQ2**), saying that an asynchronous call has been launched. This message is picked up by the outbound (**eater**) e*Way and published to a local file.

The CICS_Async_Sample_2 Schema

The **CICS_Async_Sample_2** schema builds upon the first sample. The incoming Event is picked up by the incoming (**feeder**) e*Way and published to **IQ1**. The message (incoming Event) triggers the Collaboration and is converted into a string called **message**. If the message is a **command** message it is effected. If it is a **data** message it is launched to the CICS program on the remote host.

These are manually driven schemas. The user chooses which message to send. Three sample data messages are provided in the samples file (**Async_SampleData.zip**) on the installation CD-ROM: **cics_cleanup** (the CLEANUP command message), **cics_harvest** (the HARVEST command message), **and cicstran1** (the data message).

The command messages:

- **CLEANUP**: If the incoming message is CLEANUP, a REWIND is done and resetList() is called. This scans the elements in the asynchronous call pool, five at a

time, for results. If something is found in the pool, the call object is checked to see if it has completed. After all elements in the pool are checked the pool is wiped out.

- **HARVEST**: If the incoming message is HARVEST, a REWIND is done, resetList() is called, and the pool is scanned. All information is printed out, collected, put in a string, and published to the output file.

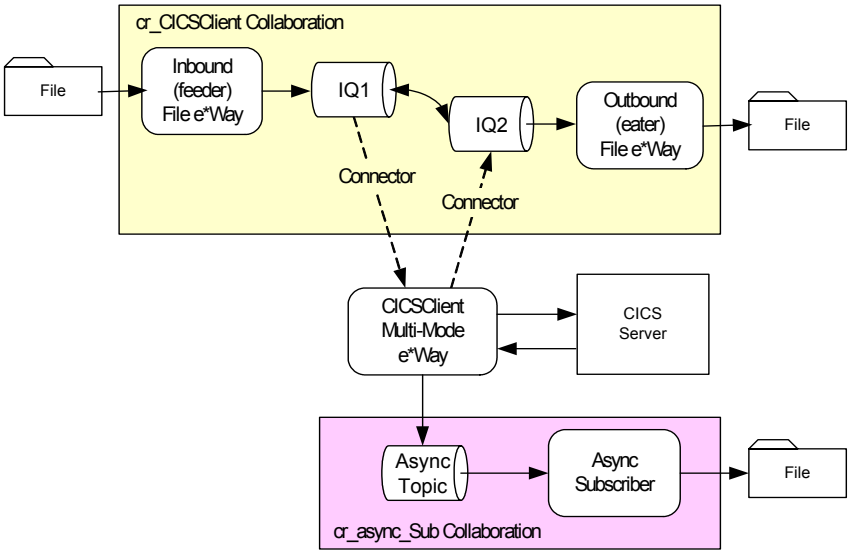
When the incoming call is a **data** message, the Commarea is set to 18 white spaces, and **EC01** is executed. An asynchronous call is launched to the CICS program on the remote host (), and a string message stating that an asynchronous call has been launched, is sent to **IQ2**, where is picked up by the outbound (**eater**) e*Way and published to a local file. **EciCallbackable** is set. The **EciCallbackable** node is used to hold an **AsyncCallHandler** object for the subsequent asynchronous call. Each asynchronous call has its own instance of an **AsyncCallHandler** object. When the call is returned it is placed in an asynchronous call pool, **AsyncCalls**, that holds and lists all of the outstanding calls. For the **CICS_Async_Sample_2** schema, the handler is registered but no topic is given and the return message is not published. It is only place in the local asynchronous call pool. The output folder receives "asynchronous call sent" messages as well as a long string consisting of all the returned messages.

Note: *When the CICS Transaction Gateway is used as the underlying connection transport, all of the Trace parameters in the e*Way Connection configuration are available for monitoring performance and for debugging.*

The CICS_Async_Sample_3 Schema

The **CICS_Async_Sample_3** schema does everything that samples 1 and 2 do. In addition, the **CICSClient e*Way** publishes the asynchronous call results to the topic **async_topic**. The **async_subscriber e*Way** picks up these messages and publishes them to a local file.

Figure 59 CICS_Async_Sample_3 Schema



5.10.2 Creating the e*Ways

The **CICS_Async_Sample_3** uses four e*Ways: **feeder** (Inbound - stcewfile.exe), **eater** (outbound - stcewfile.exe), **async_subscriber** (outbound - stcewfile.exe), and **CICSClient** (Multi-mode - stceway.exe).

Configuring the File e*Ways

The File e*Ways, **feeder**, **eater**, and **async_subscriber**, use the executable file "**stcewfile**", set in the e*Way properties. The Configuration files for the e*Ways are set as displayed in Table 9, Table 10, and Table 11.

Table 9 feeder e*Way Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	YES
AllowOutgoing	NO
Outbound (send) settings - Set as directed, otherwise leave as default.	
Poller (inbound) settings - Set as directed, otherwise leave as default.	
PollDirectory	C:\WORK_AREA\CICS_TEST
InputFileMask	*.fin
PollMilliseconds	1000
RemoveEOL	YES
MultipleRecordsPerFile	YES
MaxBytesPerLine	4096
BytesPerLineIsFixed	NO
File Records Per eGate Event	1
Performance Testing - Set as directed, otherwise leave as default.	

Table 10 eater e*Way Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	NO
AllowOutgoing	YES
Outbound (send) settings - Set as directed, otherwise leave as default.	
OutputDirectory	C:\WORK_AREA\CICS_TEST
OutputFileName	CICSAsyncTestoutput%d.dat
MultipleRecordsPerFile	YES
MaxRecordsPerFile	10000
AddEOL	YES
Poller (inbound) settings - Set as directed, otherwise leave as default.	

e*Way Configuration Parameters	Feeder
Performance Testing - Set as directed, otherwise leave as default.	

Table 11 async_subscriber e*Way Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
AllowIncoming	NO
AllowOutgoing	YES
Outbound (send) settings - Set as directed, otherwise leave as default.	
OutputDirectory	C:\WORK_AREA\CICS_TEST
OutputFileName	AsyncCallCompletionoutput%d.dat
MultipleRecordsPerFile	YES
MaxRecordsPerFile	10000
AddEOL	YES
Poller (inbound) settings - Set as directed, otherwise leave as default.	
Performance Testing - Set as directed, otherwise leave as default.	

Configuring the Multi-Mode e*Way

The Multi-Mode e*Way, **CICSClient**, uses the executable file "**stceway**", set in the e*Way's properties. For the purposes of this sample the configuration file for the Multi-Mode e*Way can be saved as default. For more information on the Multi-Mode e*Way configuration settings see the *e*Gate Integrator User's Guide*.

5.10.3 Creating the ETDs

The **CICS_Async_Sample_3** uses three Event Types (ETDs), **etd_CICS** using **cicsclient.xsc** (see **The CICSClient ETD** on page 46), **etd_Data** (BlobData.xsc) as seen in Figure 60, and **ASYNCRESPONSETOPIC** (AsyncEvent.xsc) as seen in Figure 61.

Figure 60 ETD Editor - etd_Data (BlobData.xsc)

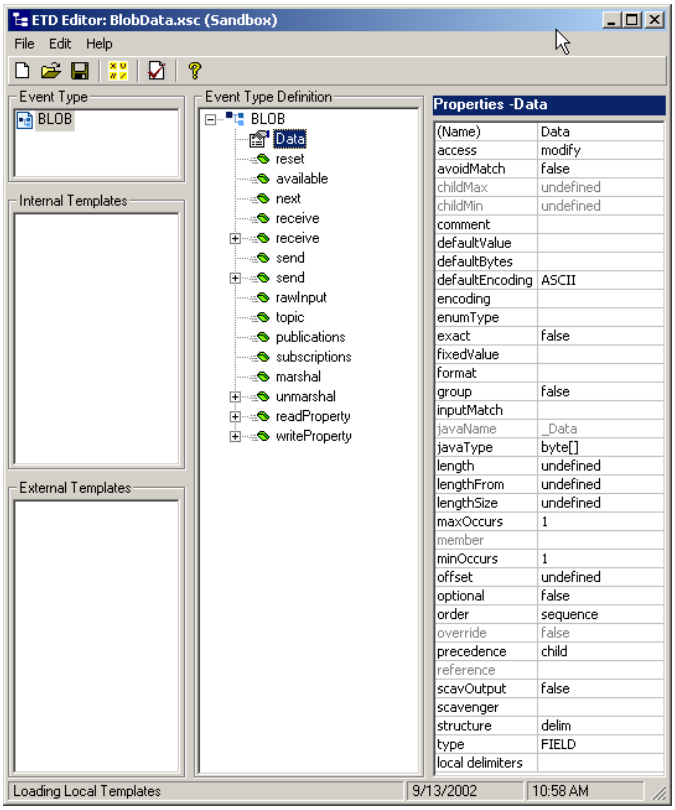
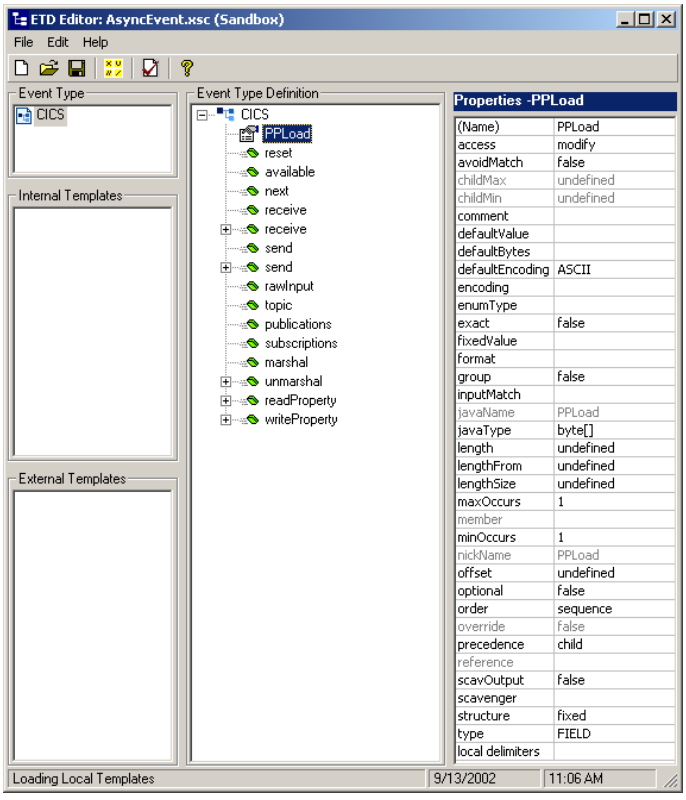


Figure 61 ETD Editor - etd_Data (BlobData.xsc)



The Event Type Definitions (ETDs) **BlobData.xsc** and **AsyncEvent.xsc**, are created using the Custom ETD Wizard. A field (child node to the root node) has been added in both ETDs, with the Properties parameter **javaType** set to **byte[]**.

5.10.4 Creating the IQs

Open the IQ Manager Properties and select **SeeBeyond JMS** as the IQ Manager Type. Create two **STC _JMS_ IQ** service IQs, **IQ1** and **IQ2**, both set for simple publish and subscribe (see **Creating Intelligent Queues** on page 89 for more information).

5.10.5. Creating the async_topic (IQ Manager)

Topics are similar to queues, the difference being that all subscribers to a topic will receive the same message when the message is published. Only one subscriber to a queue receives a message when it is published. In the CICS_Async_Sample_3 schema, the **Call Initiator** publishes the return Event to the topic, **async_topic**.

To creating the async_topic

1. From the Navigator's Components tab, select the **Control Broker** and click the **Create a New IQ Manager** button on the pallet.
2. Enter the name of the topic, for this sample "**async_topic**", and click **OK**. The new topic is added to the Editor pane.
3. Right-click the new topic to open its Properties dialog box. For this sample the default properties can be saved. The IQ Manager Type is **SeeBeyond JMS**.

5.10.6 Creating the e*Way Connections

The CICS_Async_Sample schema contains two e*Way Connections: async_topic_eWc of type SeeBeyond JMS, and conn_CICS_client of type CICS. The default **Event Type "get" interval** setting of 10000 milliseconds is sufficient for this sample. Edit any settings specific to your system. (see **Creating the e*Way Connections** on page 102 and **e*Way Connection Configuration** on page 66).

async_topic_eWc e*Way Connection (SeeBeyond JMS) configuration parameters are set as displayed in Table 12.

Table 12 async_topic_eWc.cfg Parameters

e*Way Configuration Parameters	Feeder
General Settings - Set as directed, otherwise leave as default.	
Connection Type	Topic
Transaction Type	Internal
Delivery Mode	Persistent
Connection Inactivity Mode	3000
Maximum Number of Bytes to read	5000
Default Outgoing Message Type	Bytes

e*Way Configuration Parameters	Feeder
Factory Class Name	com.stc.common.collabService. SBYNJMSFactory
Message Service - Set as directed, otherwise leave as default.	
Server Name	async_topic
Host Name	localhost
Port Number	24053
Maximum Message Cache Size	100

For more information on the SeeBeyond JMS e*Way Connection configuration parameters see the *SeeBeyond JMS Intelligent Queue User's Guide*.

conn_CICS_client e*Way Connection (CICS) configuration parameters are set as displayed in Table 13.

Table 13 conn_CICS_client.cfg Parameters

e*Way Configuration Parameters	Feeder
connector Settings - Set as directed, otherwise leave as default.	
type	CICS
Connection Transport	SeeBeyond CICS Listener
Connection Establishment Mode	Automatic
Connection Inactivity Mode	50000
Connection Verification Interval	10000
class	com.stc.eways.cics.CicsClient Connector
CICS Gateway - Set as directed, otherwise leave as default.	
URL	local
Port	2006
SSL KeyRing Class	
SSL KeyRing Password	
SeeBeyond CICS Listener - Set as directed, otherwise leave as default.	
Host	local
Port	3009
SeeBeyond CICS Listener TransId	STCL
Start Type	IC
Start Delay	000000
Listener Timeout	5000
TP Timeout	50000
Polling Rate	25
Transport Timeout	5000
COMMAREA Padding Character	40
SendBufSize	4096

e*Way Configuration Parameters	Feeder
ReceiveBufSize	4096
NoDelay	TRUE
KeepAlive	TRUE
CICS Client - Set as directed, otherwise leave as default.	
CICS UserId	
CICS Password	
ECI call type	Asynchronous
CICS Program	EC01
CICS TransId	
COMMAREA Length	1000
ECI extend mode	No
ECI LUW token	0
Message qualifier	0
Async Response Topic	ASYNCRESPONSETOPIC
Async Call JMS Server Host	localhost
Async Call JMS Server Port	24053
Encoding	cp500

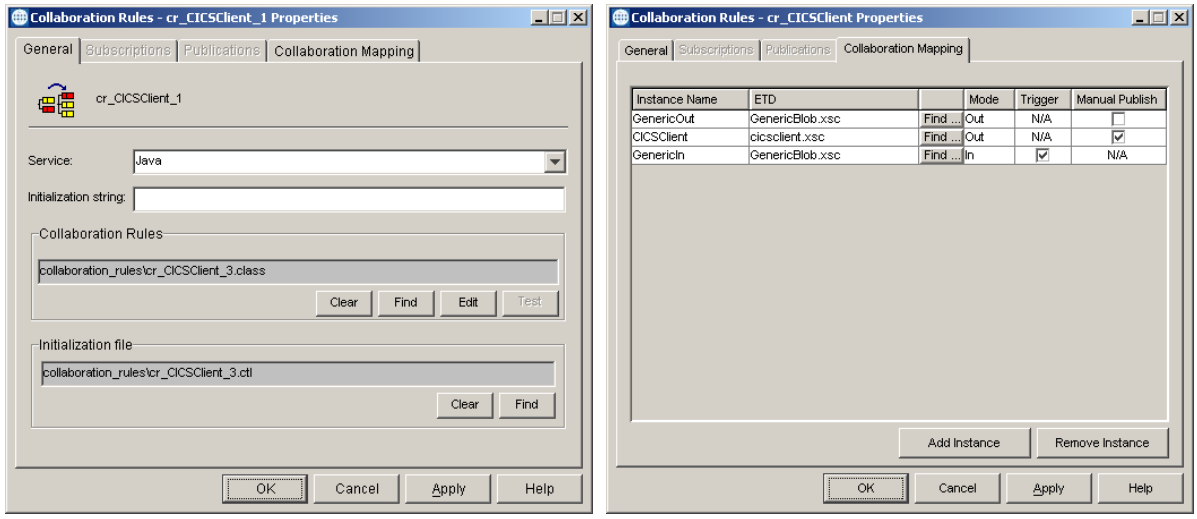
For more information of CICS e*Way Connection configuration parameters see **Creating the e*Way Connections** on page 102 and **e*Way Connection Configuration** on page 66.

5.10.7 Creating the Collaboration Rules

The **CICS_Async_Sample_3** includes four Java Collaboration Rules: **cr_CICSClient_1**, **cr_async_sub**, **cr_eater_1**, and **cr_feeder_1**. Create the Collaboration Rules as displayed. Business logic for the Java Collaboration Rules is defined using the Collaboration Rules Editor. See **Creating Collaboration Rules** on page 91 for more information.

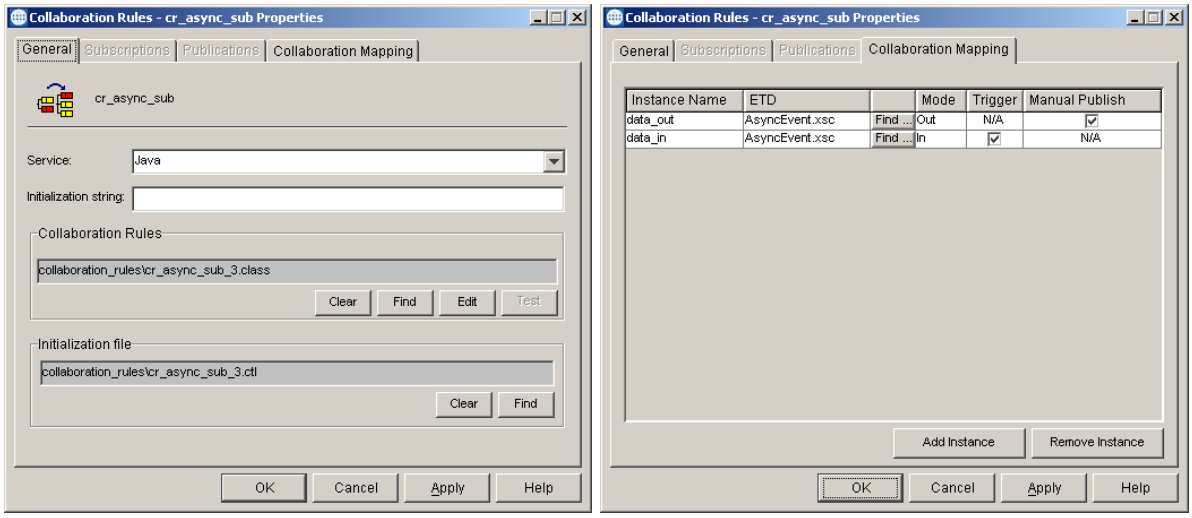
The **cr_CICSClient_1** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs are defined as they appear in Figure 62.

Figure 62 Collaboration Rules Properties - **cr_CICSClient_1**



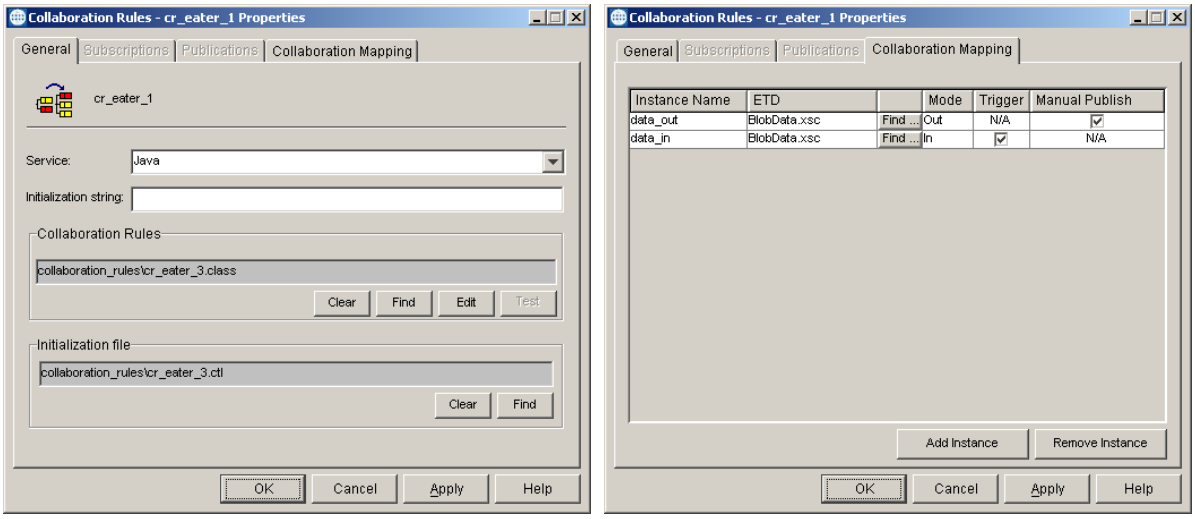
The **cr_async_sub** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs are defined as they appear in Figure 63.

Figure 63 Collaboration Rules Properties - **cr_async_sub**



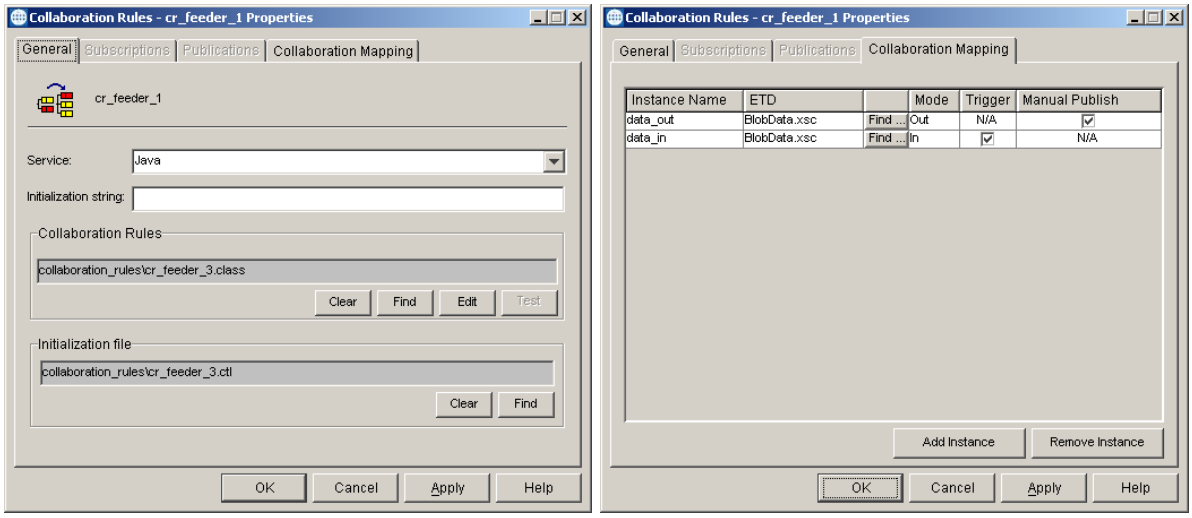
The **cr_eater_1** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs are defined as they appear in Figure 64.

Figure 64 Collaboration Rules Properties - **cr_eater_1**



The **cr_feeder_1** Collaboration Rules Properties dialog box General and Collaboration Mapping tabs are defined as they appear in Figure 65.

Figure 65 Collaboration Rules Properties - **cr_feeder_1**



From the General tab of the Collaboration Rules Properties dialog box, click **Edit** or **New** under the Collaboration Rules field to open the Collaboration Rules Editor.

5.10.8 Collaboration Rules Editor

The Business Rules for the Collaboration are defined using the Collaboration Rules Editor. The Java Collaboration Rules Editor is the graphical user interface (GUI) for creating and modifying Java Collaboration Rules. A Java Collaboration Rule is created by designating one or more source Events and one or more destination Events and then setting up rules governing the relationship between fields in the Event instances. The **CICS_Async_Sample_3** schema contains four Java Collaboration Rules. The Collaboration Rules files or .class files are cr_CICSClient_3.class, cr_eater_3.class,

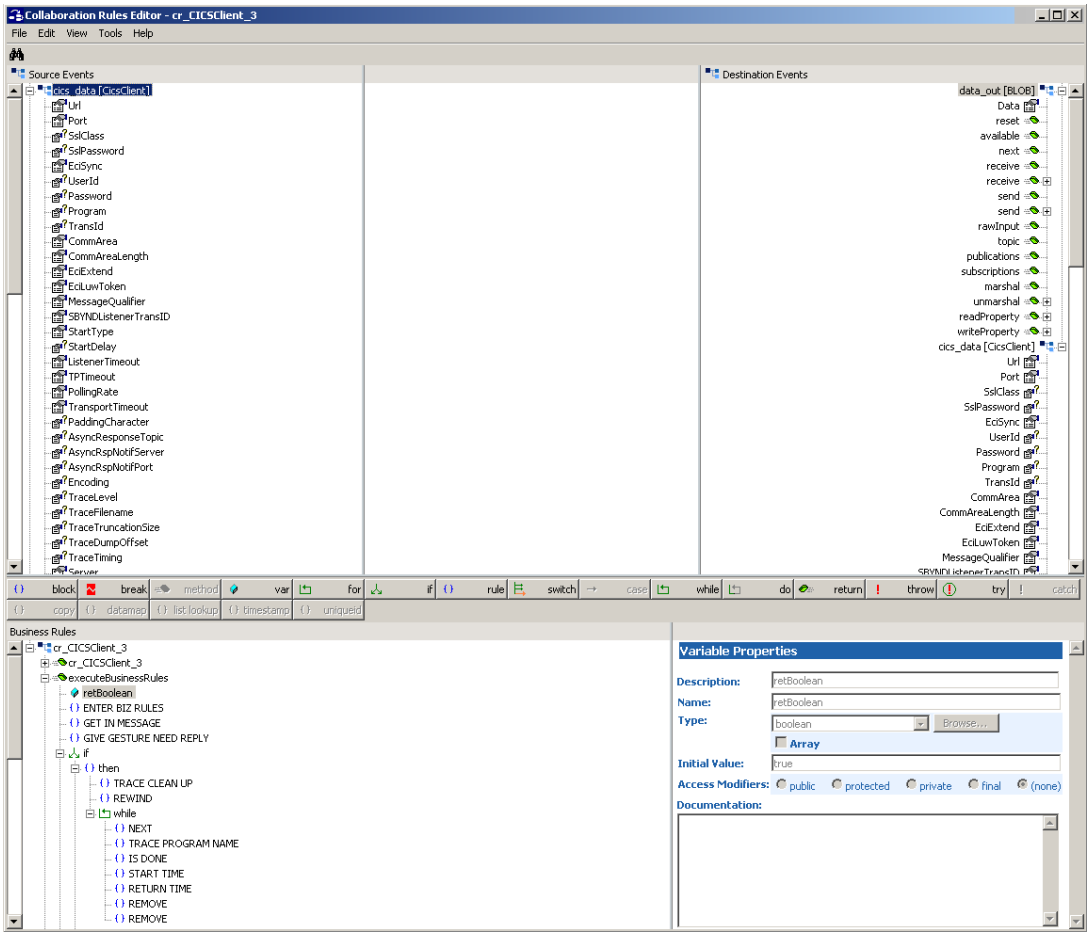
cr_feeder_3.class, and cr_async_sub.class. The cr_CICSClient_3.class Collaboration Rules are presented in detail in the following section. The other Collaboration Rules are presented as they appear in the schema.

The cr_CICSClient_3.class Collaboration Rules

The **cr_CICSClient_1** Collaboration Rules use the **cr_CICSClient_3.class** Collaboration Rules file (see Figure 67). The Business Rules for **cr_CICSClient_3.class** (see Figure 67 to see the completed Business Rules) are created as follows:

Each **rule** is created by clicking the **rule** button on the Editor's Business Rules toolbar or by "dragging and dropping" a node or method from the Source Events pane onto a node or method in the Destination Events pane. When creating a Business Rule "dragging and dropping" a node or method into the properties pane can be used as a shortcut to writing code. For more information on using the Java Collaboration Rules Editor, see the *e*Gate Integrator User's Guide.*

Figure 66 Collaboration Rules Editor - cr_CICSClient_3.class



1 The first **rule** under retBoolean in the Business Rules window, **ENTER BIZ RULES**, is an available trace statement used for performance testing and debugging. It is created by typing the following into the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "ENTER BIZ RULE
ASYNC===")
```

Rule descriptions are created by typing the description in the Rule Properties, Description field.

2 The **GET IN MESSAGE** rule is created by typing **String msg = new String()** in the Rule Properties, Rule field. Drag **Data**, under data_in on the Source Events command node, into the last set of parentheses to create the following code:

```
String msg = new String(getdata_in().get_Data())
```

3 To create the **GIVE GESTURE NEED REPLY** rule, drag and drop **EciCallbackable** under CICSClient on the Destination Events command node, into the Rule Properties, Rule field. Drag the **createAsyncCallHandler** method under CICSClient, on the Destination Events command node, into the last set of parentheses to create the following code:

```
getcics_data().setEciCallbackable(getcics_data().createAsyncCallHandler())
```

4 Click the **if** button on the Business Rules toolbar to create an **if** statement. Type the following in the If Properties, Condition field:

```
msg.equals("CLEANUP")
```

5 Create the **TRACE CLEAN UP** rule under the **then** statement by typing the following code in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "CLEAN UP ASYNC
POOL====")
```

6 To create the **REWIND** rule, drag the **resetList** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
getcics_data().getAsyncCalls().resetList()
```

7 Click the **while** button on the toolbar to create a **while** loop. Drag the **hasNext** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the **while** Properties, Condition field to create the following code:

```
getcics_data().getAsyncCalls().hasNext()
```

8 The **Next** rule, under the **while** loop, is created by dragging the **next** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
getcics_data().getAsyncCalls().next()
```

9 The **TRACE PROGRAM NAME** rule, under the **while** loop, is a trace statement created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "PROGNAME=" +
getcics_data().getAsyncCalls().getProgram())
```

10 The **IS DONE** rule, under the **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "IS DONE=" +
getcics_data().getAsyncCalls().isDone())
```

11 The **START TIME** rule, under the **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "START TIME=" +
getcics_data().getAsyncCalls().getStartTime())
```

12 The **RETURN TIME** rule, under the **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "RETURN TIME=" +
getcics_data().getAsyncCalls().getReturnTime())
```

13 The first **REMOVE** rule, under the **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "REMOVE ASYNC CALL
FROM POOL===")
```

14 The second **REMOVE** rule under the **while** loop, is created by dragging the **remove** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
getcics_data().getAsyncCalls().remove()
```

15 Create a second **if** statement under the else statement and type the following in the If Properties, Condition field:

```
msg.equals("HARVEST")
```

16 Create the **TRACE HARVEST** rule under the second **then** statement by typing the following code in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "HARVEST   ASYNC
POOL====")
```

17 Create the **HARVEST STR** rule under the second **then** statement by typing the following code in the Rule Properties, Rule field:

```
String harvestStr = ""
```

18 Create the **Rewind** rule under the second **then** statement by dragging the **resetList** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
getcics_data().getAsyncCalls().resetList()
```

19 Click the **while** button on the toolbar to create a second **while** loop. Drag the **hasNext** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the **while** Properties, Condition field to create the following code:

```
getcics_data().getAsyncCalls().hasNext()
```

20 The **Next** rule, under the second **while** loop, is created by dragging the **next** method under cics_data [CicsClient], AsyncCalls on the Source Events command node, into the Rule Properties, Rule field to create the following code:

```
getcics_data().getAsyncCalls().next()
```

21 The **TRACE PROGRAM NAME** rule, under the second **while** loop, is a trace statement created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "PROGNAME=" +
getcics_data().getAsyncCalls().getProgram())
```

22 The **IS DONE** rule, under the second **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "IS DONE=" +
getcics_data().getAsyncCalls().isDone())
```

23 The **START TIME** rule, under the second **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "START TIME=" +
getcics_data().getAsyncCalls().getStartTime())
```

24 The **RETURN TIME** rule, under the second **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "RETURN TIME=" +
getcics_data().getAsyncCalls().getReturnTime())
```

25 The **COLLECT** rule, under the second **while** loop, is created by typing the following code in the Rules Properties, Rule field:

```
harvestStr += "Topic:" + getcics_data().getAsyncCalls().getTopic() + " PROGRAM:" +
getcics_data().getAsyncCalls().getProgram() + " START TIME:" +
getcics_data().getAsyncCalls().getStartTime() + " RETURN TIME: " +
getcics_data().getAsyncCalls().getReturnTime() + " IS DONE :" +
getcics_data().getAsyncCalls().isDone()
```

26 The **PUB HARVEST STR** rule, under the second **then** statement, is created by dragging Data, under data_out on the Destination Events command node, into the Rule Properties, Rule field. Place the cursor into the last set of parentheses and type **harvestStr.getBytes()** to create the following code:

```
getdata_out().set_Data(harvestStr.getBytes())
```

27 The **SEND** rule, under the second **then** statement, is created by dragging the **send** method, under data_out on the Destination Events command node, into the Rule Properties, Rule field to create the following code:

```
getdata_out().send()
```

28 The **TRACE INVOKE** rule, under the second **else** statement, is created by typing the following code in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "INVOKE  ASYNC CALL
====")
```

29 The **SET COMM LEN** rule, under the second **else** statement, is created by dragging **CommAreaLength**, under data_out on the Destination Events command node, into the Rule Properties, Rule field. Place the cursor into the last set of parentheses and type **18** as the property, to create the following code:

```
getcics_data().setCommAreaLength(18)
```

30 The **SET COMM AREA** rule, under the second **else** statement, is created by dragging **CommArea**, under data_out on the Destination Events command node, into the Rule Properties, Rule field. Place the cursor into the last set of parentheses and type **" ".getBytes("cp500")** with 18 spaces between the quotation marks, to create the following code:

```
getcics_data().setCommArea("                  ".getBytes("cp500"))
```

31 The **GET SERVERS** rule, under the second **else** statement, is created by typing **String[] servers** = in the Rule Properties, Rule field. Drag the **getServerList** method, under cics_data on the Source Events command node to the end of the statement in the Rule Properties, Rule field. When prompted for the maxNumSystem parameter, enter 1 and click OK. The following code is created:

```
String[] servers = getcics_data().getServerList(1)
```

32 The **EXEC PROGRAM** rule, under the second **else** statement, is created by dragging the first **execute** method, under cics_data on the Source Events command node, into the Rule Properties, Rule field to create following code:

```
getcics_data().execute()
```

33 The **TRACE COMM AREA** rule, under the second **else** statement, is created by typing the following in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "AFTER EXECUTE()
====COMM AREA=" + getcics_data().getEncodedCommAreaString())
```

34 The **COPY TO BUF** rule, under the second **else** statement, is created by dragging **Data**, under data_out on the Destination Events command node, into the Rule

Properties, Rule field. Place the cursor into the last set of parentheses and type **"PROGRAM ASYNC CALLED===".getBytes()** to create following code:

```
getdata_out().set_Data("PROGRAM ASYNC CALLED===".getBytes())
```

35 The **SEND** rule, under the second **else** statement, is created by dragging the **send** method, under data_out on the Destination Events command node, into the Rule Properties, Rule field to create the following code:

```
getdata_out().send()
```

36 The **EXIT BIZ RULES** rule is created by typing the following in the Rule Properties, Rule field:

```
EGate.traceln(EGate.TRACE_COLLABSERVICE_VERBOSE, EGate.TRACE_EVENT_TRACE, "EXIT BIZ RULE====")
```

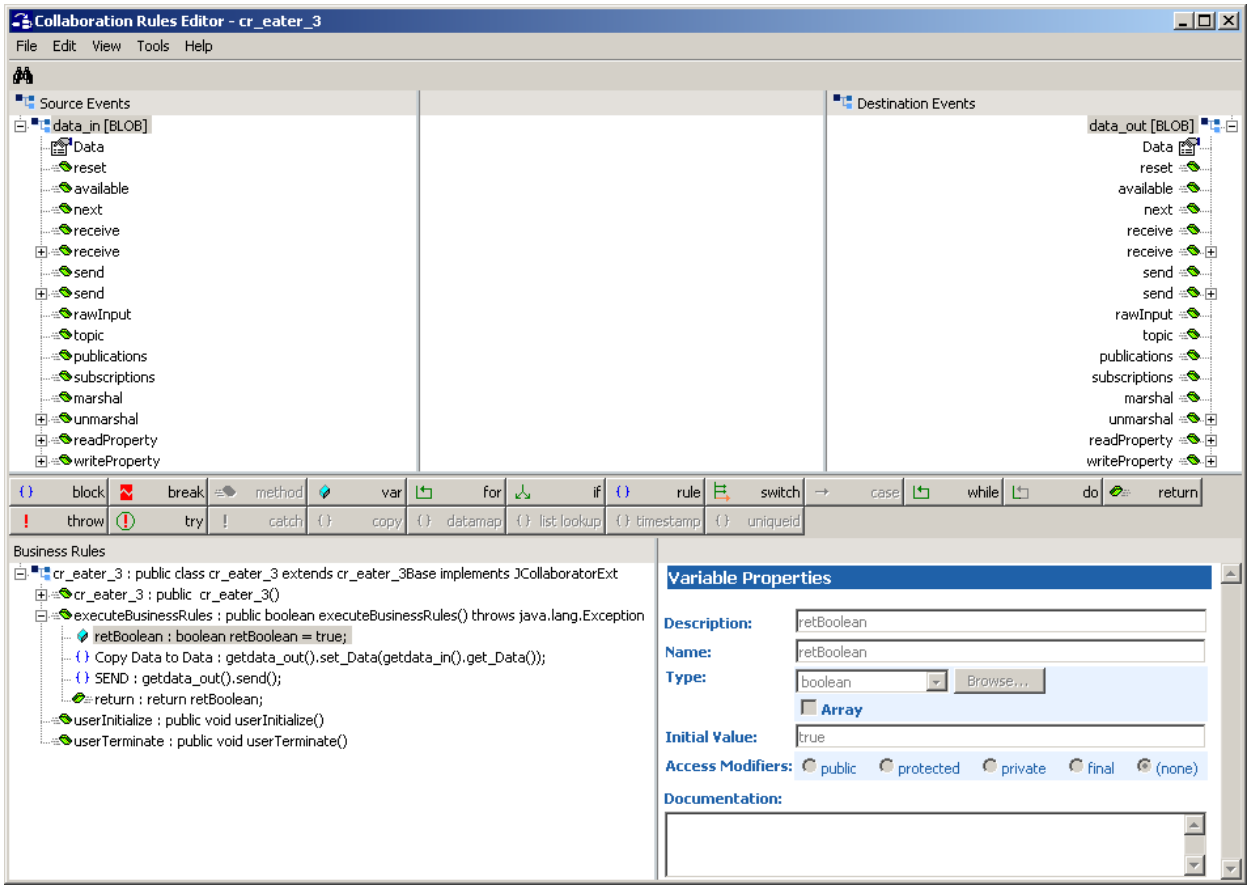
Figure 67 cr_CICSClient_3.class Business Rules

37 When the business logic is complete, select **Save** from the File menu, an then select **Compile** from the File menu. If the file compiles successfully, select **Promote** from the File menu to promote the file to the run-time environment.

The cr_eater_3.class Collaboration Rules

The cr_eater_1 Collaboration Rules, cr_eater_3.class file is displayed in Figure 68.

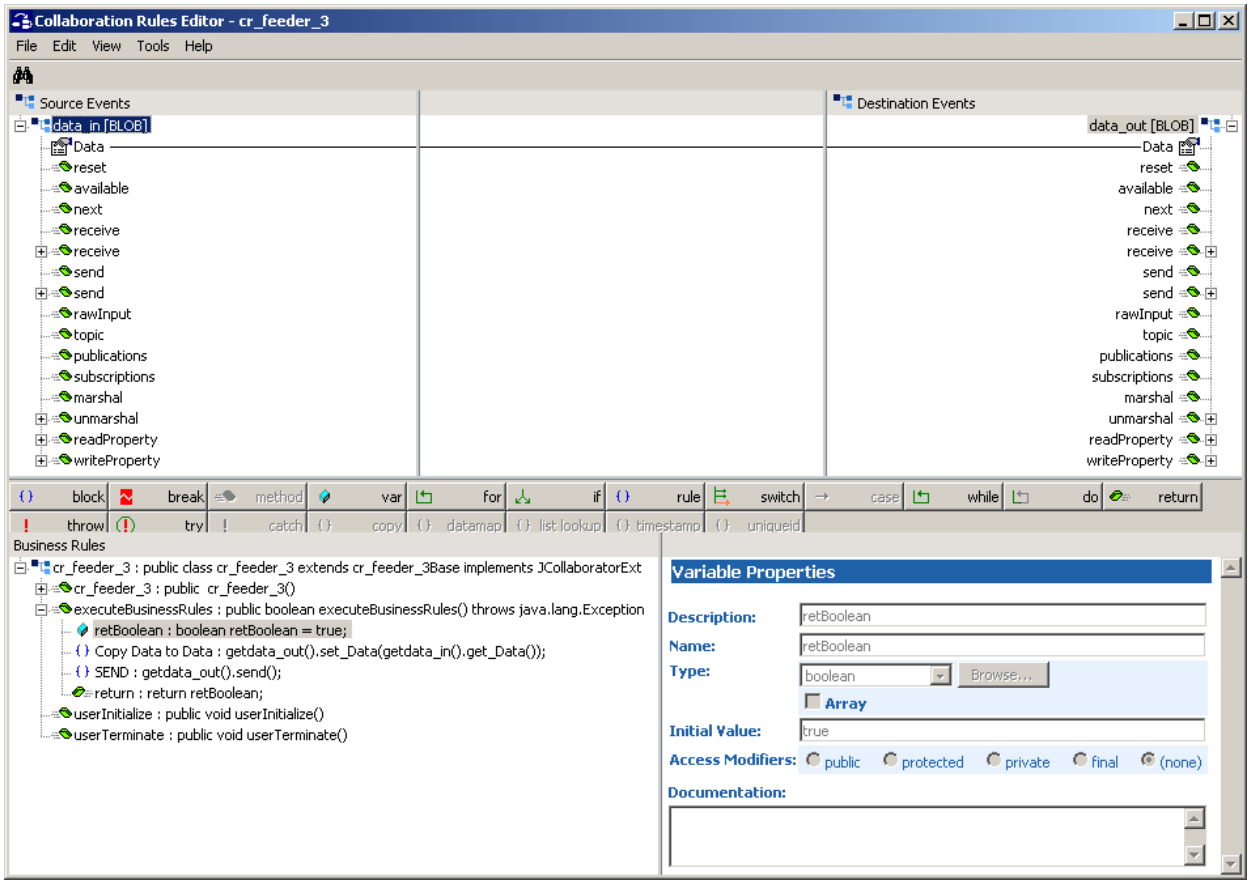
Figure 68 Collaboration Rules Editor - cr_eater_3.class



The cr_feeder_3.class Collaboration Rules

The cr_feeder_1 Collaboration Rules, cr_feeder_3.class file is displayed in Figure 69.

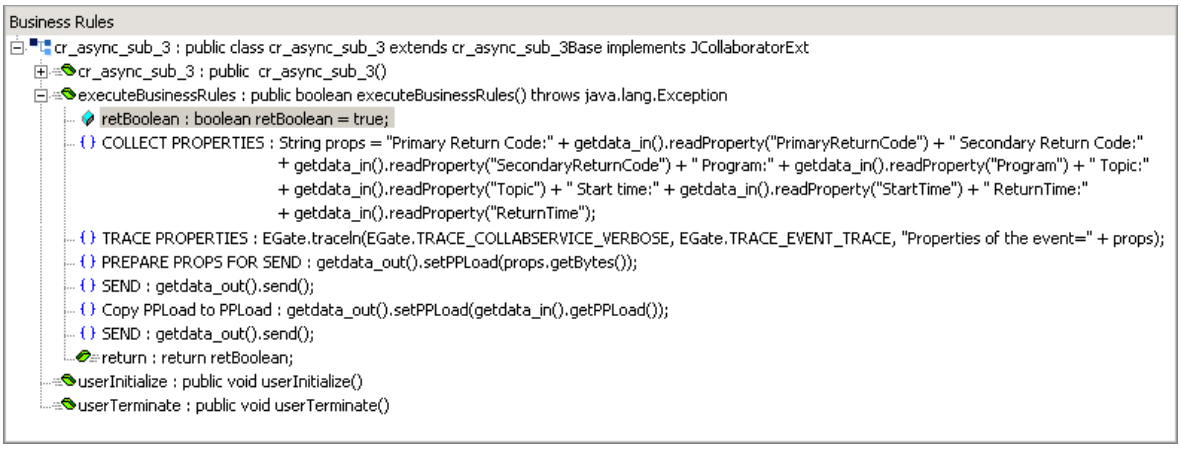
Figure 69 Collaboration Rules Editor - cr_feeder_3.class



The cr_async_sub_3.class Collaboration Rules

The **cr_async_sub_1** Collaboration Rules, **cr_async_sub_3.class** Business Rules are displayed in Figure 70.

Figure 70 Business Rules - cr_async_sub_3.class

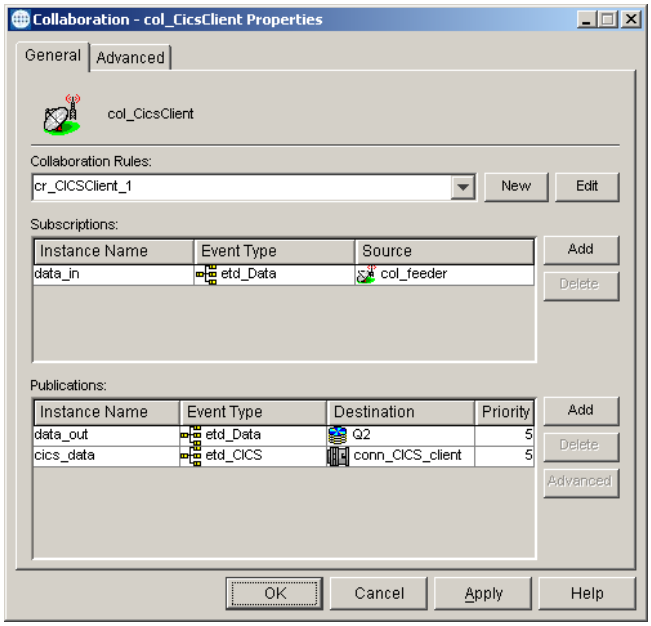


5.10.9 Creating the Collaborations

The CICS_Async_Sample_3 schema contains 4 Java Collaborations: **col_CicsClient**, **col_eater**, **col_feeder**, and **col_async_sub**

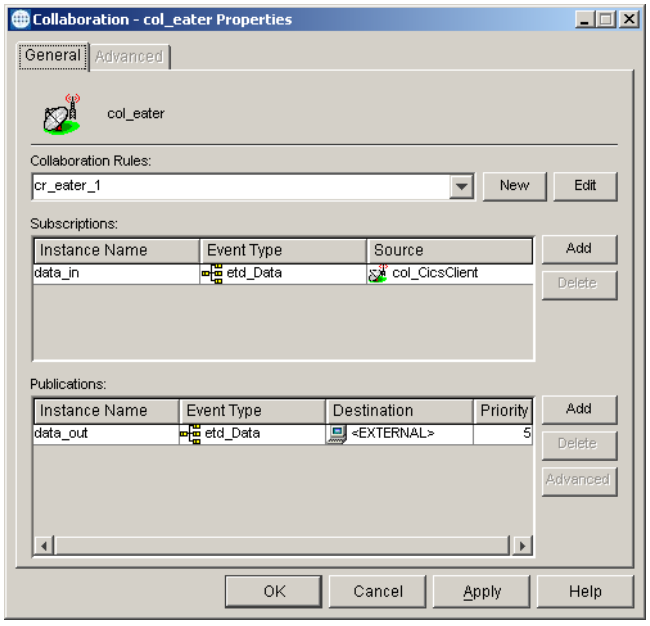
The **CICSClient** e*Way Collaboration, named **col_CicsClient**, appears as it is displayed in Figure 73 when complete.

Figure 71 Collaboration Properties - col_async_sub



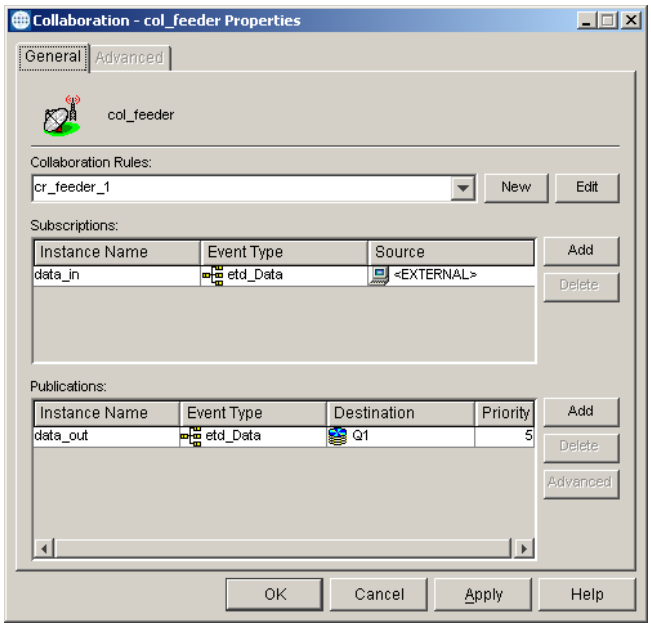
The outbound (**eater**) e*Way Collaboration, named **col_eater**, appears as it is displayed in Figure 72 when complete.

Figure 72 Collaboration Properties - col_eater



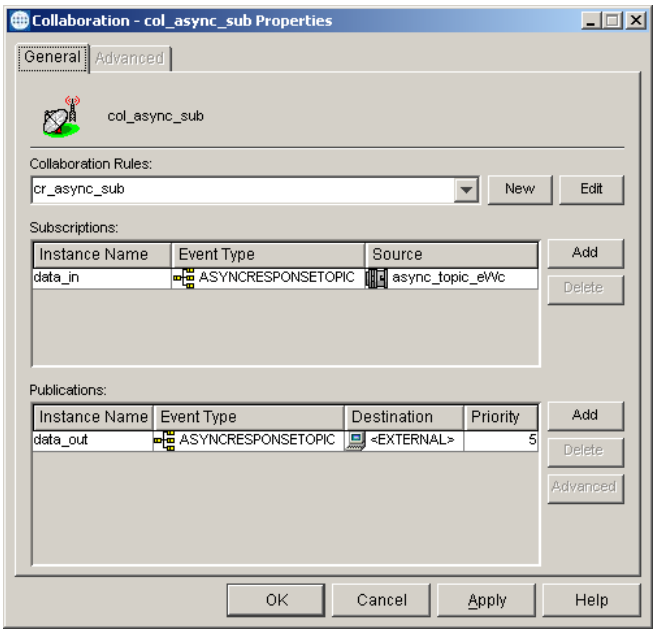
The inbound (**feeder**) e*Way Collaboration, named **col_feeder**, appears as it is displayed in Figure 73 when complete.

Figure 73 Collaboration Properties - col_feeder



The Collaboration for the **async_subscriber** e*Way, named **col_async_sub,** appears as it is displayed in Figure 74 when complete.

Figure 74 Collaboration Properties - col_async_sub



5.11 Executing the Schemas

To execute the a sample schema, do the following:

1. Go to the command line prompt, and enter the following:

   ```
   stccb -rh hostname -rs samplename -un username -up user password
   -ln hostname_cb
   ```

   Substitute ***hostname***, *samplename*, ***username*** and ***user password*** as appropriate.

2. Start the Schema Manager. Specify the server that contains the Control Broker you started in Step 1 above.
3. Select the sample schema.
4. Verify that the Control Broker is connected. To do this, select and right-click the Control Broker in the Schema Manager, and select **Status**. (The message in the Control tab of the console will indicate command *succeeded* and status as *up*.)
5. Select the IQ Manager, ***hostname***_igmgr, then right-click and select **Start**. (This will already be started if **Start automatically** is selected in the IQ Manager properties.)
6. Select each of the e*Ways, right-click select **Start**. (These will already be started if **Start automatically** is selected in the e*Way's properties.)
7. To view the output, copy the output file (specified in the Outbound e*Way configuration file). Save to a convenient location, and open.

Note: *Opening the destination file while the schema is running will cause errors.*

5.12 Running CTG on Multiple CICS Servers

To select specific CICS Servers when running CICS Transaction Gateway with multiple CICS Servers configured, set the **setServer( )** method to select the CICS region with which to connect. The CICS e*Way always defaults to the first array element (CICS server) that gets returned from the g**etServerList( )** method. To select a CICS server other that the first server on the list the **setServer( )** method must be used.

The following example shows two CICS servers that are available to CTG on a Windows computer. The second CICS server (OS390R2A) has been set as the CICS server with which to connect.

```
String [] CICS_List_Str = getCICSClient().getServerList(2);

System.err.println("CICS Server 1 " + CICS_List_Str[0]);

System.err.println("CICS Server 2 " + CICS_List_Str[1]);

System.err.println("CICS Server Default " +
getCICSClient().getServer());

getCICSClient().setServer(CICS_List_Str[1]);

System.err.println("Setting CICS Server to " +
getCICSClient().getServer());
```

Output for the above example:

```
CICS Server 1 OS390R29
CICS Server 2 OS390R2A
CICS Server Default OS390R29
Setting CICS Server to OS390R2A
```

Chapter 6

Java Methods

A number of Java methods have been added to make it easier to set information in the e*Way ETD Editor and to get information from it. These methods are contained in the CicsClient Class. In addition, helper methods used by with toPackedDecimal( ) are documented at the end of this chapter.

6.1 The CicsClient Class

```
java.lang.Object
    com.stc.jcsre.SimpleETDImpl(implements com.stc.jcsre.ETDExt)
        com.stc.eways.cics.CicsClient
```

public class **CicsClient** extends com.stc.jcsre.

The `CicsClient` class represents an ETD through which a Collaboration can invoke transaction programs on a CICS server. Nodes and methods are exposed so that the Collaboration can conveniently prepare a request for a CICS program, invoke the program, and get result from the program. There are two underlying transport mechanisms that can be used to achieve the remote invoking of CICS programs: (1) CTG, the IBM CICS Transaction Gateway, and (2) SBYND CICS Listener, the SeeBeyond CICS Listener, a light weight proprietary protocol based on TCP/IP. configuration parameters can be roughly categorized into CTG specific, SBYND CICS Listener specific, or necessary for both CTG and SBYND CICS Listener. Also, the exposed ETD methods are categorized as CTG specific (such as getServerList()), SBYND CICS Listener specific (such as prepareAPCRecord(), returnCodeIs(), returnOK(), and getResponse(...)) or common to both CTG and SBYND CICS Listener (such as execute(), execute(....), and sendRequest(...)).

Methods of the CicsClient Class

These methods are described in detail on the following pages:

CicsClient

Description

Constructor for class com.stc.eways.cics.CicsClient

Syntax

```
public CicsClient()
```

Parameters

None.

Return Values

None.

Throws

None.

commAreaToPackedDecimal

Description

Builds a packed decimal from the payload in the Commarea section specified by (offset, intSize, decSize).

Syntax

```
public com.stc.eways.cics.PackedDecimal commAreaToPackedDecimal(int
offset, int intSize, int decSize)
```

Parameters

Name	Type	Description
offset	int	Offset of the packed decimal field relative to the start of the Commarea (a field starting in byte 1 would have an offset of 0)
intSize	int	The number of integer digits in the resulting object.
decSize	int	The number of decimal digits in the resulting object.

Return Values

com.stc.eways.cics.PackedDecimal
Returns the packed decimal object.

Throws

None

commAreaZonedToString

Description

Convert the zoned decimal (COBOL PIC S9) byte array Commarea field to a String using current value of ETD node `Encoding` as encoding;

Syntax

```
public java.lang.String commAreaZonedToString(int offset, int length)
```

Parameters

Name	Type	Description
offset	int	Start of the zone.
length	int	The length of the zone.

Note: *Methods that include an "encoding" parameter should specify encoding as "ISO-8859-1" when the e*Way is run on z/OS.*

Return Values

java.lang.String
Returns the resultant string.

Throws

None.

commAreaZonedToString

Description

Converts the zoned decimal (COBOL PIC S9) byte array Commarea field to a String.

Syntax

```
public java.lang.String commAreaZonedToString(int offset, int length,
java.lang.String encoding)
```

Parameters

Name	Type	Description
offset	int	Start of the zone.
length	int	The length of the zone.
encoding	java.lang.String	Encoding used for conversion.

Note: *Methods that include an "encoding" parameter should specify encoding as "ISO-8859-1" when the e*Way is run on z/OS.*

Return Values

java.lang.String
Returns the resultant string.

Throws

None.

connect

Description

Establish a connection to the CICS server, used by the Collaboration to send requests (CICS program calls) to the server. The underlying transport used can be CTG or SBYND Listener, it is transparent to the Collaboration as far as the connect() is concerned.

Syntax

```
public void connect()
```

Parameters

None.

Return Values

None.

Throws

com.stc.common.collabService.CollabConnException
Thrown when there's an external connection problem.

com.stc.common.collabService.CollabDataException
Thrown when there's a data problem.

disconnect

Description

Disconnect the connection established through connect(). The underlying transport used can be CTG or SBYND Listener, it is transparent to the Collaboration as far as the connect() is concerned.

Syntax

```
public void disconnect()
```

Parameters

None.

Return Values

None.

Throws

com.stc.common.collabService.CollabConnException
Thrown when there's an external connection problem.

com.stc.common.collabService.CollabDataException
Thrown when there's a data problem.

---

execute

Description

Launches the CICS program. Uses the CICSClientETD node parameter values set in the configuration file.

- For **SBYND CICS Listener** use **SBYND CICS Listener host** as the server name and select the following to execute: (eciSync, server, userId, password, program, transId,commArea, commAreaLength, eciExtend, eciLuwToken, messageQualifier, eciCallbackable). Set irrelevant parameters to 0 or null. Only the following parameters are still used by the SBYND CICS Listener: (1) eciSync (2) userId (3) password (4) program (5) transId (6) commArea.
- For **CTG** select the following to execute: (eciSync, server, userId, password, program, transId,commArea, commAreaLength, eciExtend, eciLuwToken, messageQualifier, eciCallbackable).

Syntax

```
public void execute()
```

Parameters

None.

Return Values

None.

Throws

com.stc.common.collabService.CollabConnException
Indicating a connection error.

com.stc.common.collabService.CollabDataException
Indicating a data error.

---

execute

Description

Launches the CICS program. The CICSClientETD node parameter set for this method override the values set in the configuration file. Values must be entered for all fields.

- For **SBYND CICS Listener** invoke the remote CICS program using the following steps (defined by a proprietary protocol)

A Prepare an APC record (Application Control Record) with a request code of **SBYND_LISTENER_REQCODE_REQ_SYNC** or **SBYND_LISTENER_REQCODE_REQ_ASYNC** depending on the parameter `eciSynCall` with `byteArray` as the payload; the prepare APC record is in the outbound buffer in `SBYNDCicsProxy`.

B Send the request to SBYND CICS Listener.

C Get the response from SBYND CICS Listener.

D Further get returned data from the program if it is a sync call

E ACK or NACK accordingly

- For CTG: prepare an ECIRequest object with the data given via the parameters, call method flow() to send the request and get result (synchronous) or proceed (asynchronous).

Syntax

```
public void execute(boolean eciSynCall,
                    java.lang.String cicsServerName,
                    java.lang.String cicsUserId,
                    java.lang.String cicsPassword,
                    java.lang.String cicsProgram,
                    java.lang.String cicsTransId,
                    byte[] byteArray,
                    int length,
                    boolean eciExtendMode,
                    int eciLUWToken,
                    int msgQualifier,
                    com.stc.eways.cics.Callbackable eciCallbackableObj)
```

Parameters

Name	Type	Description
eciSynCall	boolean	A Boolean value indicating whether to use ECI Synchronous Call. **True** invokes the program synchronously, and **false** invokes the program asynchronously.
cicsServerName	java.lang.String	The CICS server name (CTG only).
cicsUserId	java.lang.String	The CICS user Id.
cicsPassword	java.lang.String	The CICS password.
cicsProgram	java.lang.String	The CICS Program.
cicsTransId	java.lang.String	The CICS transaction Id.
byteArray	byte []	The payload area (INOUT & OUTPUT).
length	int	The length of the payload (CTG only).
eciExtendMode	boolean	A Boolean value indicating whether to implement ECI extend mode (CTG only).
eciLUWToken	int	An ECI LUW token (Logical Unit of Work token) (CTG only).
msgQualifier	int	Application provided identifier (CTG only).

Name	Type	Description
eciCallbackableObj	com.stc.eways.cics.Callbackable	ECI callbackable object. This may be null if no callback is required (CTG only).

Return Values

None.

Throws

com.stc.common.collabService.CollabConnException
Indicating a connection error.

com.stc.common.collabService.CollabDataException
Indicating a data error.

getCommArea

Description

"Get" method for the ETD node **CommArea**. Returns the COMMAREA, that is, the current value in ETD node **CommArea**.

Syntax

```
public byte[] getCommArea()
```

Parameters

None.

Return Values

byte array
Returns the COMMAREA used when invoking CICS programs.

Throws

None.

getCommAreaLength

Description

"Get" method for ETD node **CommAreaLength**. Returns the **CommAreaLength**, that is, the current value in ETD node **CommAreaLength**. The initial value is taken from the parameter **CommArea Length** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public int getCommAreaLength()
```

Parameters

None

Return Values

int
Returns the COMMAREA length used when invoking CICS programs (not necessarily the length of the byte array represented as COMMAREA).

Throws

None.

getCommAreaString

Description

Constructs a COMMAREA String by converting the COMMAREA array of bytes using the platform's default character encoding.

Syntax

```
public java.lang.String getCommAreaString()
```

Parameters

None.

Return Values

java.lang.String
Returns the COMMAREA in String form.

Throws

None.

getCommAreaString

Description

Constructs a COMMAREA String by converting the COMMAREA array of bytes using the character encoding specified as an argument.

Syntax

```
public java.lang.String getCommAreaString(java.lang.String enc)
```

Parameters

Name	Type	Description
enc	java.lang.String	The encoding used when translating the COMMAREA byte array into a String.

Note: *Methods that include an "encoding" parameter should specify encoding as "ISO-8859-1" when the e*Way is run on z/OS.*

Return Values

java.lang.String
Returns the COMMAREA as a String.

Throws

java.io.UnsupportedEncodingException

getCommAreaString

Description

Construct a COMMAREA String by converting the COMMAREA array of bytes with offset and length using the character encoding specified as an argument.

Syntax

```
public java.lang.String getCommAreaString(int offset, int length)
```

Parameters

Name	Type	Description
offset	int	Offset of the area to be converted relative to the start of the COMMAREA (a field starting in byte 1 would have an offset of 0).
length	int	The length of the area to be converted.

Return Values

java.lang.String
Returns the String instantiated from the COMMAREA section specified by (offset, length) using system default encoding.

Throws

None.

getCommAreaString

Description

Constructs a COMMAREA String by converting the COMMAREA array of bytes with offset and length using the platform's default character encoding.

Syntax

```
public java.lang.String getCommAreaString(int offset, int length,
java.lang.String enc)
```

Parameters

Name	Type	Description
offset	int	Offset of the area to be converted relative to the start of the COMMAREA (a field starting in byte 1 would have an offset of 0).
length	int	The length of the area to be converted.
enc	java.lang.String	The encoding used when translating the COMMAREA byte array into a String.

Note: *Methods that include an "encoding" parameter should specify encoding as "ISO-8859-1" when the e*Way is run on z/OS.*

Return Values

java.lang.String
Returns the String instantiated from COMMAREA section specified by (offset, length) using encoding specified by parameter enc.

Throws

java.io.UnsupportedEncodingException

getEciCallbackable

Description

"Get" method for the ETD node **EciExtend**. Returns the EciExtend flag, that is, the current value in ETD node **EciExtend**. The initial value is taken from the parameter **ECI extend mode** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public com.stc.eways.cics.Callbackable getEciCallbackable()
```

Parameters

None.

Return Values

com.stc.eways.cics.Callbackable
Returns the ECI callbackable value.

Throws

None.

getEciExtend

Description

CTG specific. "Get" method for the ETD node **EciExtend**. Returns the EciExtend flag, that is, the current value in the ETD node **EciExtend**. The initial value is taken from the parameter **ECI extend mode** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public boolean getEciExtend()
```

Parameters

None

Return Values

Boolean
Returns **true** to indicate that the current call is and extension of a LUW; otherwise, returns **false**.

Throws

None.

getEciLuwToken

Description

CTG specific. "Get" method for the ETD node **EciLuwToken**. Returns the ECI Luw token, that is, the current value for the ETD node **EciLuwToken**. The initial value is taken from the parameter **ECI LUW token** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public int getEciLuwToken()
```

Parameters

None.

Return Values

int
Returns the current value of ECI Luw token.

Throws

None.

getEciSync

Description

"Get" method for the ETD node **EciSync**. Returns the ECI call type, that is, the current value for the ETD node **EciSync**. The initial value is taken from the parameter **ECI call type** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public boolean getEciSync()
```

Parameters

None.

Return Values

Boolean
Returns true to indicate that the call is synchronous.

Throws

None.

getEncodedCommAreaString

Description

Constructs a COMMAREA String by converting the COMMAREA array of bytes using the character encoding specified earlier for the ETD.

Syntax

```
public java.lang.String getEncodedCommAreaString()
```

Parameters

None.

Return Values

java.lang.String
Returns the COMMAREA in String form using encoding to do the translation.

Throws

java.io.UnsupportedEncodingException
Indicating unsupported encoding.

getEncodedCommAreaString

Description

Construct a COMMAREA String by converting the COMMAREA array of bytes with the offset and length using the platform's default character encoding.

Syntax

```
public java.lang.String getEncodedCommAreaString(int offset, int length)
```

Parameters

Name	Type	Description
offset	int	Offset of the area to be converted relative to the start of the COMMAREA (a field starting in byte 1 would have an offset of 0).
length	int	The length of the area to be converted.

Return Values

java.lang.String
Returns the String instantiated from COMMAREA section specified by (offset, length) using encoding indicated by the current value of the ETD node **`Encoding`**.

Throws

java.io.UnsupportedEncodingException
Indicating unsupported encoding.

getEncoding

Description

"Get" method for the ETD node **Encoding**. Returns the encoding which can be used to translate the data to and from the CICS program, that is, the current value for the ETD node **Encoding**. The initial value is taken from the parameter **Encoding** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getEncoding()
```

Parameters

None.

Return Values

java.lang.String
Returns the encoding type.

Throws

None.

getListenerTimeout

Description

SBYND CICS Listener specific. "Get" method for Listener time out ETD node **ListenerTimeout**. The initial value is taken from the parameter **Listener Timeout** in the **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

Syntax

```
public int getListenerTimeout()
```

Parameters

None.

Return Values

int

Returns the timeout value in milliseconds. This is usually set to the time that SBYND listener waits for the program invoking the request from the e*Way before it closes the connection.

Throws

None.

getMessageQualifier

Description

"Get" method for ETD node **MessageQualifier**. Returns the MessageQualifier for ECI call, that is, the current value for the ETD node **MessageQualifier**. The initial value is taken from the parameter **Message qualifier** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public int getMessageQualifier()
```

Parameters

None.

Return Values

int

Message qualifier.

Throws

None.

getPaddingCharacter

Description

SBYND CICS Listener specific. "Get" method for the padding character ETD node **PaddingCharacter**. The initial value is from parameter **COMMAREA Padding Character** in **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

```
public java.lang.String getPaddingCharacter()
```

Parameters

None.

Return Values

java.lang.String
Returns the EBCDIC code for the character used by the SBYND listener to pad the COMMAREA at the CICS server when the actual length of the payload in the COMMAREA is shorter than the length given by COMMAREALength. The default is hexadecimal 40 - EBCDIC space.

Throws

None.

getPassword

Description

"Get" method for the ETD node **Password**. Returns the CICS user password, that is, the current value for the ETD node **Password**. The initial value is taken from the parameter **CICS Password** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getPassword()
```

Parameters

None.

Return Values

java.lang.String
Returns the CICS user password used when making a CICS call to a program on the CICS server.

Throws

None.

getPollingRate

Description

SBYND CICS Listener specific. "Get" method for the Listener Polling Rate ETD node **PollingRate**. The initial value is taken from parameter **Polling Rate** in the **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

Syntax

```
public int getPollingRate()
```

Parameters

None.

Return Values

int
Returns a 1-255 value used by listener internally.

Throws

None.

getPort

Description

"Get" method for ETD node **Port**. Returns the port of the host where CTG gateway is connected, that is, the current value for the ETD node **Port**. The initial value is taken from the parameter **Port** in the **CICS Gateway** section of the e*Way Connection configuration.

Syntax

```
public int getPort()
```

Parameters

None.

Return Values

int
Returns the port number.

Throws

None.

getProgram

Description

"Get" method for the ETD node **Program**. Returns the CICS program name to be invoked, that is, the current value for the ETD node **Program**. The initial value is taken

from the parameter **CICS Program** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getProgram()
```

Parameters

None.

Return Values

java.lang.String
Returns the CICS program to be invoked.

Throws

None.

getProgramName

Description

SBYND CICS Listener specific. Returns the program name of the response, assuming a response is in the current inbound buffer. Should be called immediately after getResponse();

Syntax

```
public java.lang.String getProgramName()
```

Parameters

None.

Return Values

java.lang.String
Returns the program name.

Throws

com.stc.common.collabService.CollabDataException
Thrown when there's a data error.

getRequestCode

Description

SBYND CICS Listener specific. Returns the request code of the response, assuming a response is in the current inbound buffer. Should be called immediately after getResponse().

Syntax

```
public int getRequestCode()
```

Parameters

None.

Return Values

int
Returns the request code.

Throws

com.stc.common.collabService.CollabDataException
Thrown when there's a data error.

getREQUESTCODES

Description

SBYND CICS Listener specific. Returns a **SBYNDAppControlRecordRequestCodes** object. This get method is for the ETD node **REQUESTCODES**.

Syntax

```
public com.stc.eways.cics.SBYNDAppControlRecordRequestCodes
getREQUESTCODES()
```

Parameters

None.

Return Values

com.stc.eways.cics.SBYNDAppControlRecordRequestCodes
Returns SBYNDAppControlRecordRequestCodes.

Throws

None.

getRequestDesc

Description

SBYND CICS Listener specific. Return a description for the request code, assuming a response is in the current inbound buffer. Should be called immediately after getResponse().

Syntax

```
public java.lang.String getRequestDesc()
```

Parameters

None.

Return Values

java.lang.String
Returns the description of the request code.

Throws

com.stc.common.collabService.CollabDataException

Thrown when there's a data error.

---

getResponse

Description

SBYND CICS Listener specific. Read from the SBYND CICS Listener in blocking mode until timed out or a response occurs.

Syntax

```
public boolean getResponse()
```

Parameters

None.

Return Values

Boolean

`Returns` **`true`** if get a good ACR otherwise, **`false`**. The SBYND CICS Listener parameter: Transport Timeout is used for the timeout.

Throws

com.stc.common.collabService.CollabConnException
com.stc.common.collabService.CollabDataException
com.stc.eways.cics.SBYNDCicsProxyTimeoutException

---

getResponse

Description

SBYND CICS Listener specific. Read from the SBYND CICS Listener in blocking mode until timed out or a response occurs.

Syntax

```
public boolean getResponse(int timeout)
```

Parameters

Name	Type	Description
`timeout`	int	Timeout in milli-seconds that the e*Way will wait on a response (ACR) from the listener. If the e*Way did not get an ACR in that amount of time, an exception (SBYNDCicsProxyTimeoutException) is thrown.

Return Values

Boolean

Returns **true** for a good ACR, otherwise, returns **false**.

Throws

com.stc.common.collabService.CollabConnException
com.stc.common.collabService.CollabDataException
com.stc.eways.cics.SBYNDCicsProxyTimeoutException

---

getReturnCode

Description

SBYND CICS Listener specific. Returns the error code of the current response, assuming a response is in the current inbound buffer. It should be called immediately after getResponse().

Syntax

```
public int getReturnCode()
```

Parameters

None.

Return Values

int

Returns the error code.

Throws

com.stc.common.collabService.CollabDataException

Thrown when there's a data error.

---

getRETURNCODES

Description

SBYND CICS Listener specific. Returns a **SBYNDAppControlRecordRequestCodes** object. This getter is for the ETD node **REQUESTCODES**.

Syntax

```
public com.stc.eways.cics.SBYNDAppControlRecordRequestCodes
getREQUESTCODES()
```

Parameters

None.

Return Values

com.stc.eways.cics.SBYNDAppControlRecordRequestCodes

Returns SBYNDAppControlRecordRequestCodes.

Throws

None.

getReturnMessage

Description

SBYND CICS Listener specific. Returns the error text of the current response, assuming a response is in the current inbound buffer. getReturnMessage should be called immediately after getResponse().

Syntax

```
public java.lang.String getReturnMessage()
```

Parameters

None.

Return Values

java.lang.String
Returns the text of the error message.

Throws

com.stc.common.collabService.CollabDataException
Thrown when there's a data error.

getSBYNDCicsProxyConfig

Description

SBYND CICS Listener specific. Returns the SBYND CICS Listener specific configuration parameters.

Syntax

```
public com.stc.eways.cics.SBYNDCicsProxyConfig
getSBYNDCicsProxyConfig()
```

Parameters

None.

Return Values

com.stc.eways.cics.SBYNDCicsProxyConfig
Returns SBYNDCicsProxyConfig.

Throws

None.

getSBYNDListenerTransID

Description

SBYND CICS Listener specific. "Get" method for the SBYND CICS Listener TransID ETD node **SBYNDListenerTransID**. The initial value is taken from the parameter **SeeBeyond CICS Listener TransId** in the **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getSBYNDListenerTransID()
```

Parameters

None.

Return Values

java.lang.String
Returns the listener trans ID. The default value is "STCL".

Throws

None.

getServer

Description

"Get" method for ETD node **Server**. Returns the name of the CICS server where the request is to be sent, that is, the current value in ETD node **Server**.

Syntax

```
public java.lang.String getServer()
```

Parameters

None

Return Values

java.lang.String
Returns the name of the CICS server.

Throws

None.

getServerList

Description

Obtains a list of CICS servers defined as name-description pairs with SBYND Listener as the transport. Only one server is returned, that is, the SBYND Listener host (this should be the same host as the CICS server).

Syntax

```
public java.lang.String[] getServerList(int maxNumSystems)
```

Parameters

Name	Type	Description
maxNumSystems	int	The maximum number of systems.

Return Values

java.lang.String[]
Returns a list of the available CICS servers to which CICS call can be issued.

Throws

com.stc.common.collabService.CollabConnException
Indicating a connection error.

com.stc.common.collabService.CollabDataException
Indicating a data error.

getSslClass

Description

CTG specific. "Get" method for the ETD node **SslClass**. Returns the SSL class for SSL authentication, that is, the current value in the ETD node **SslClass**. The initial value is taken from the parameter **SSL KeyRing Class** in the **CICS Gateway** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getSslClass()
```

Parameters

None

Return Values

java.lang.String
Returns the full classname of the SSL KeyRing class.

Throws

None.

getSslPassword

Description

CTG specific. "Get" method for the ETD node **SslPassword**. Returns the password for the encrypted KeyRing class, that is, the current value in the ETD node **SslPassword**. The initial value is from the parameter SSL KeyRing Password in the **CICS Gateway** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getSslPassword()
```

Parameters

None

Return Values

java.lang.String
Returns the SSL KeyRing Password.

Throws

None.

getStartDelay

Description

SBYND CICS Listener specific. "Get" method for the Startup delay ETD node **StartDelay**. The initial value is taken from the parameter **Start Delay** in the **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getStartDelay()
```

Parameters

None

Return Values

java.lang.String
Returns the IBM CICS Listener needed parameter **Startup delay**, a string in the format of HHMMSS, indicating the time elapsed before the STCL (SBYND CICS Listener) wakes up.

Throws

None.

getStartType

Description

SBYND CICS Listener specific. "Set" method for the Startup delay ETD node **StartType**.

Syntax

```
public void setStartType(java.lang.String starttype)
```

Parameters

None

Return Values

java.lang.String
Returns the startup type for the IBM CICS Listener. The possible values are **IC** or **TD**.

Throws

None.

---

getTPTimeout

Description

SBYND CICS Listener specific. "Get" method for the CICS program time out ETD node **TPTimeout**. The initial value is taken from the parameter **TP Timeout** in the **SeeBeyond CICS Listener** section of the e*Way Connection configuration.

Syntax

```
public int getTPTimeout()
```

Parameters

None

Return Values

int
Returns the timeout value in milli-seconds. This is typically set to the maximum estimated time it takes the CICS program to finish and come back to the inoker.

Throws

None.

---

getTraceDumpOffset

Description

CTG specific. "Get" method for the ETD node **TraceDumpOffset**. Returns the trace dump offset for CTG client log, that is, the current value in ETD node **TraceDumpOffset**. The initial value is taken from the parameter **Dump Offset** in the **Tracing** section of the e*Way Connection configuration.

Syntax

```
public int getTraceDumpOffset()
```

Parameters

None.

Return Values

int
Returns the trace dump offset for CTG client logging.

Throws

None.

getTraceFilename

Description

CTG specific. "Get" method for the ETD node **TraceFilename**. Returns the trace file name for the CTG client log, that is, the current value in the ETD node **TraceFilename**. The initial value is taken from the parameter **Filename** in the **Tracing** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getTraceFilename()
```

Parameters

None.

Return Values

java.lang.String
Returns the trace filename for CTG.

Throws

None.

getTraceLevel

Description

CTG specific. "Get" method for the ETD node **TraceLevel**. Returns the trace level for the CTG client log, that is, the current value in the ETD node **Tracelevel**. The initial value is taken from the parameter **Level** in the **Tracing** section of the e*Way Connection configuration.

Syntax

```
public int getTraceLevel()
```

Parameters

None.

Return Values

int
Returns the trace level for the CTG client.

Throws

None.

getTraceTiming

Description

CTG specific. "Get" method for the ETD node **TraceTiming**. Returns the trace timing (time stamp) in the CTG client log, that is, the current value in the ETD node **TraceTiming**. The initial value is taken from the parameter **Timing** in the **Tracing** section of the e*Way Connection configuration.

Syntax

```
public boolean getTraceTiming()
```

Parameters

None

Return Values

Boolean
Returns true to indicate that the time stamp is included in the CTG client tracing, otherwise false.

Throws

None.

getTraceTruncationSize

Description

CTG specific. "Get" method for the ETD node **TraceTruncationSize**. Returns the trace truncation size for the CTG client log, that is, the current value in the ETD node **TraceTruncationSize**. The initial value is taken from the parameter **Truncation Size** in the **Tracing** section of the e*Way Connection configuration.

Syntax

```
public int getTraceTruncationSize()
```

Parameters

None.

Return Values

int
Returns the truncation size for CTG client logging.

Throws

None.

getTransId

Description

"Get" method for the ETD node **TransId**. Returns the **CICS TransId** to be invoked, that is, the current value in the ETD node **TransId**. The initial value is taken from the parameter **CICS TransId** in the **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getTransId()
```

Parameters

None.

Return Values

java.lang.String
Returns the trans ID of the CICS program.

Throws

None.

getTransportTimeout

Description

SBYND CICS Listener specific. "Get" method for the TransportTimeout ETD node **TransportTimeout**. The initial value is taken from the **TransportTimeout** parameter in the SeeBeyond CICS Listener section of the e*Way Connection configuration.

Syntax

```
public int getTransportTimeout()
```

Parameters

None.

Return Values

int
Returns the time in milli-seconds used to timeout a blocking read on a socket between the e*Way and the listener.

Throws

None.

getUrl

Description

CTG specific. "Get" method for the ETD node **Url**. Returns the URL pointing to the remote or local CICS Transaction Gateway with which to connect, that is, the current

value in ETD node **Url**. The initial value is taken from the parameter **URL** in section **CICS Gateway** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getUrl()
```

Parameters

None.

Return Values

java.lang.String
Returns the URL of the CICS Transaction Gateway.

Throws

None.

getUserId

Description

"Get" method for the ETD node **UserId**. Returns the **CICS user Id**, that is, the current value in the ETD node **UserId**. The initial value is taken from the parameter **CICS UserId** in **CICS Client** section of the e*Way Connection configuration.

Syntax

```
public java.lang.String getUserId()
```

Parameters

None.

Return Values

java.lang.String
Returns the CICS user Id used when making a CICS call to a program on the CICS server.

Throws

None.

isConnected

Description

Checks to see if the connection is active.

Syntax

```
public boolean isConnected()
```

Parameters

None.

Return Values

Boolean
Returns **true** when active, **false** otherwise.

Throws

com.stc.common.collabService.CollabDataException
Thrown when there's a data problem.

packedDecimalToString

Description

Gets the string from a packed decimal object.

Syntax

```
public static java.lang.String
packedDecimalToString(com.stc.eways.cics.PackedDecimal pd)
```

Parameters

Name	Type	Description
pd	com.stc.eway.cics.PackedDecimal	The PackedDecimal object.

Return Values

java.lang.String
Returns the String, applying toString() against the packed decimal object

Throws

None.

prepareAPCRecord

Description

SBYND CICS Listener specific. Prepares an APC Record in the outbound buffer of **SBYNDCicsProxy** using the parameters. This can be sent out by invoking sendRequest().

Syntax

```
public void prepareAPCRecord(java.lang.String progname,
                             int appltimeout,
                             int requestcode,
                             int errorcode,
                             java.lang.String errortext)
```

Parameters

Name	Type	Description
progname	java.lang.String	The transaction program name.
appltimeout	int	Application timeout not used (reserved).
requestcode	int	Request code of the ACR.
errorcode	int	Error code of the ACR.
errortext	java.lang.String	Error message (usually used when the ACR is a NACK).

Return Values

None.

Throws

com.stc.common.collabService.CollabDataException
Throne when there is an error in data.

returnCodeIs

Description

SBYND CICS Listener specific. Check to see if the error code in the APC record is the same as code.

Syntax

```
public boolean returnCodeIs(int code)
```

Parameters

Name	Type	Description
code	int	The code

Return Values

Boolean
Returns **true** if the error code is the same as code, otherwise false.

Throws

com.stc.common.collabService.CollabDataException
Indicating a data error.

returnOK

Description

SBYND CICS Listener specific. Checks to see if the error code in the APC record is **SBYND_LISTENER_RC_OK**.

Syntax

```
public boolean returnOK()
```

Parameters

None

Return Values

Boolean
Returns **true** if the error code true error code is SBYND_LISTENER_RC_OK, otherwise **false**.

Throws

None

sendRequest

Description

- **For SBYND CICS Listener:** Sends an ACR to the listener, assuming an ACR is properly prepared and is in the outbound buffer. **sendRequest()** only sends an ACR if a response is expected. Need to call **getResponse()** subsequently.
- **For CTG:** The sendRequest method flows data contained in the ECIRequest object to the Gateway and determines whether send has been successful by checking the return code. If an error has occurred, a CollabConnException is thrown.

Syntax

```
public void sendRequest(com.stc.eways.cics.ECIRequest request)
```

Parameters

Name	Type	Description
request	com.stc.eway.cics.ECIRequest	ECIRequest object contains all the data needed to invoke a CICS program through CTG. For the SBYND CICS Listener this parameter is ignored.

Return Values

None.

Throws

com.stc.common.collabService.CollabConnException
Indicating a connection error.

com.stc.common.collabService.CollabDataException
Indicating a data error.

setCommArea

Description

"Set" method for the ETD node **CommArea**. Sets the payload into the COMMAREA usually used by the subsequent invoking program.

Note: *When using CTG, the setCommArea method must have a byte array of the same size as that specified in the setCommAreaLength method.*

Syntax

```
public void setCommArea(byte[] byteArray)
```

Parameters

Name	Type	Description
byteArray	byte[]	The payload used by the program to be invoked.

Return Values

None.

Throws

None.

setCommAreaLength

Description

Sets the COMMAREA length.

Syntax

```
public void setCommAreaLength(int i)
```

Parameters

Name	Type	Description
i	int	COMMAREA length.

Return Values

None.

Throws

None.

setEciCallbackable

Description

Sets the ECI callbackable value.

Syntax

```
public void setEciCallbackable(com.stc.eways.cics.Callbackable c)
```

Parameters

Name	Type	Description
c	com.stc.eway.cics.Callbackable	ECI callbackable value.

Return Values

None.

Throws

None.

setEciExtend

Description

CTG specific. "Set" method for ETD node EciExtend.

Syntax

```
public void setEciExtend(boolean b)
```

Parameters

Name	Type	Description
b	boolean	true if the current call is an extension of a LUW, false otherwise.

Return Values

None.

Throws

None.

setEciLuwToken

Description

CTG specific. "Set" method for the ETD node **EciLuwToken**. An integer identifying an LUW. The initial value is taken from the parameter **ECI LUW token** in the CICS Client section of the e*Way Connection configuration.

Syntax

```
public void setEciLuwToken(int i)
```

Parameters

Name	Type	Description
i	int	The LUW identifier.

Return Values

None.

Throws

None.

setEciSync

Description

"Set" method for the ETD node **EciSync**. Indicates whether the call will be synchronous or asynchronous.

Syntax

```
public void setEciSync(boolean b)
```

Parameters

Name	Type	Description
b	boolean	The EciSync flag. True indicates that the call will be a synchronous call, false indicates asynchronous.

Return Values

None.

Throws

None.

setEncoding

Description

"Set" method for the ETD node **Encoding**.

Syntax

```
public void setEncoding(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The encoding used for payload translation.

Return Values

None.

Throws

None.

setListenerTimeout

Description

SBYND CICS Listener specific. "Set" method for the Listener Timeout ETD node **ListenerTimeout**.

Syntax

```
public void setListenerTimeout(int timeout)
```

Parameters

Name	Type	Description
Timeout	int	The time that the SBYND listener will wait for program invoking request from the e*Way before it close the connection.

Return Values

None.

Throws

None.

setMessageQualifier

Description

CTG specific. "Set" method for the ETD node **MessageQualifier**.

Syntax

```
public void setMessageQualifier(int i)
```

Parameters

Name	Type	Description
i	int	The message qualifier.

Return Values

None.

Throws

None.

setPaddingCharacter

Description

SBYND CICS Listener specific. "Set" method for padding character ETD node **PaddingCharacter**.

Syntax

```
public void setPaddingCharacter(java.lang.String padcharhex)
```

Parameters

Name	Type	Description
padcharhex	java.lang.String	The EBCDIC code for the character used by the SBYND listener to pad the COMMAREA at the CICS server when the actual length of the payload in the COMMAREA is shorter than the length given by CommAreaLength. The default value is hexadecimal 40 - EBCDIC space.

Return Values

None

Throws

None.

setPassword

Description

Sets the password associated with the terminal.

Syntax

```
public void setPassword(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	Description

Return Values

None.

Throws

None.

Additional Information

Invoking this method automatically flags the terminal as an extended type of terminal. The password will not be picked up until another send is completed or the terminal is connected.

setPollingRate

Description

SBYND CICS Listener specific. "Set" method for the Polling Rate ETD node **PollingRate**.

Syntax

```
public void setPollingRate(int rate)
```

Parameters

Name	Type	Description
rate	int	A 1-255 value used by listener internally.

Return Values

None.

Throws

None.

setPort

Description

Sets the port number necessary to communicate with the Gateway.

Syntax

```
public void setPort(int i)
```

Parameters

Name	Type	Description
i	int	The Gateway port number.

Return Values

None.

Throws

None.

setProgram

Description

"Set" method for ETD node Program. The CICS program to be called.

Syntax

```
public void setProgram(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The name of the CICS program.

Return Values

None.

Throws

None.

setSBYNDListenerTransID

Description

SBYND CICS Listener specific. "Set" method for the SBYND CICS Listener TransID ETD node **SBYNDListenerTransID**.

Syntax

```
public void setSBYNDListenerTransID(java.lang.String transid)
```

Parameters

Name	Type	Description
transid	java.lang.String	The transaction ID of the SBYND CICS Listener - STCL.

Return Values

None.

Throws

None.

setServer

Description

"Set" method for ETD node **Server**.

Syntax

```
public void setServer(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The CICS server name.

Return Values

None.

Throws

None.

setSslClass

Description

CTG specific. "Set" method for the ETD node **SslClass**.

Syntax

```
public void setSslClass(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The SSL class name.

Return Values

None.

Throws

None.

setSslPassword

Description

CTG specific. "Set" method for the ETD node **SslPassword**. The PASSWORD for the encrypted KeyRing class.

Syntax

```
public void setSslPassword(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The SSL password.

Return Values

None.

Throws

None.

setStartDelay

Description

SBYND CICS Listener specific. "Set" method for Startup delay ETD node **StartDelay**.

Syntax

```
public void setStartDelay(java.lang.String startdelay)
```

Parameters

Name	Type	Description
startdelay	java.lang.String	The delay value, in the format HHMMSS, used by IBM CICS Listener to delay the wake up of the SBYND listener. The default value is 000000.

Return Values

None.

Throws

None.

setStartType

Description

SBYND CICS Listener specific. "Set" method for the Startup delay ETD node **StartType**.

Syntax

```
public void setStartType(java.lang.String starttype)
```

Parameters

Name	Type	Description
starttype	java.lang.String	The Startup type value, either IC or TD, used by the IBM CICS Listener to decide how the SBYND Listener will be waked up.

Return Values

None.

Throws

None.

setTPTimeout

Description

SBYND CICS Listener specific. "Set" method for the TP Timeout ETD node **TPTimeout**.

Syntax

```
public void setTPTimeout(int timeout)
```

Parameters

Name	Type	Description
timeout	int	The timeout value in milli-seconds. typically, this is set to the maximum estimated time it takes the CICS program to finish and come back to the inoker.

Return Values

None.

Throws

None.

setTraceDumpOffset

Description

CTG specific. "Set" method for the ETD node **TraceDumpOffset**.

Syntax

```
public void setTraceDumpOffset(int i)
```

Parameters

Name	Type	Description
i	int	The offset amount.

Return Values

None.

Throws

None.

setTraceFilename

Description

CTG specific. "Set" method for the ETD node **TraceFilename**.

Syntax

```
public void setTraceFilename(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The CTG client tracing file name.

Return Values

None.

Throws

None.

setTraceLevel

Description

CTG specific. "Set" method for the ETD node **TraceLevel**.

Syntax

```
public void setTraceLevel(int i)
```

Parameters

Name	Type	Description
i	int	The CTG client tracing level.

Return Values

None.

Throws

None.

setTraceTiming

Description

CTG specific. "Set" method for the ETD node **TraceTiming**.

Syntax

```
public void setTraceTiming(boolean b)
```

Parameters

Name	Type	Description
b	boolean	**true** includes the time stamp in CTG client tracing, otherwise is **false**.

Return Values

None.

Throws

None.

setTraceTruncationSize

Description

CTG specific. "Set" method for the ETD node **TraceTruncationSize**.

Syntax

```
public void setTraceTruncationSize(int i)
```

Parameters

Name	Type	Description
i	int	The CTG client tracing truncation size.

Return Values

None.

Throws

None.

setTransId

Description

"Set" method for ETD node Program. The CICS program's trans ID to be called.

Syntax

```
public void setTransId(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The trans ID of the CICS program.

Return Values

None.

Throws

None.

setTransportTimeout

Description

SBYND CICS Listener specific. "Set" method for the Transport Timeout ETD node **TransportTimeout**.

Syntax

```
public void setTransportTimeout(int timeout)
```

Parameters

Name	Type	Description
timeout	int	The time in milli-seconds used to timeout a blocking read on a socket between the e*Way and the listener.

Return Values

None.

Throws

None.

setUrl

Description

CTG specific. "Set" method for the ETD node **Url**. Set the URL pointing to the remote or local CICS Transaction Gateway with which to connect.

Syntax

```
public void setUrl(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The URL for the Transaction Gateway.

Return Values

None.

Throws

None.

setUserId

Description

Sets the used ID associated with the terminal.

Syntax

```
public void setUserId(java.lang.String s)
```

Parameters

Name	Type	Description
s	java.lang.String	The terminal user ID.

Return Values

None.

Throws

None.

toPackedDecimal

Description

Builds a packed decimal from a string number. Converts the in String +-99999.99 in a packed decimal. IBM data Flow: each digit is a 0..9. Numerical value of the last digit is

the sign digit: A | C | E | F => + ; B | D => -. The decimal point is virtual. Its position is defined in the second byte of dec_length.

Note: *Helper methods for toPackedDecimal are provided with the CICS e*Way. For information on these helper methods see* **Packed Decimal Java Helper Methods** *on page 185.*

Syntax

```
public static com.stc.eways.cics.PackedDecimal
toPackedDecimal(java.lang.String number,
                int intSize,
                int decSize)
```

Parameters

Name	Type	Description
number	java.lang.String	Decimal String representation to be converted
intSize	int	The number of integer digits in the resulting object.
decSize	int	The number of decimal digits in the resulting object.

Return Values

com.stc.eways.cics.PacedDecimal
Returns the packed decimal object.

Throws

java.lang.NumberFormatException

toZoned

Description

Converts a number in the form of a String to a zoned decimal (COBOL PIC S9) byte array using the current value of the ETD node **Encoding** as the encoding.

Syntax

```
public static byte[] toZoned(java.lang.String number)
```

Parameters

Name	Type	Description
number	java.lang.String	The number String.

Return Values

byte []
Returns the resultant byte array.

Throws

None.

toZoned

Description

Converts a number in the form of a String to a zoned decimal (COBOL PIC S9) byte array using the encoding specified by the parameter **enc**.

Syntax

```
public static byte[] toZoned(java.lang.String number,
java.lang.String enc)
```

Parameters

Name	Type	Description
number	java.lang.String	The number String.
enc	java.lang.String	The encryption type.

Return Values

byte []
Returns the resultant byte array.

Throws

None.

zonedToString

Description

Converts the zoned decimal (COBOL PIC S9) byte array specified by **zoned** to a String using current value of the ETD node **Encoding** as encoding.

Syntax

```
public static java.lang.String zonedToString(byte[] zoned)
```

Parameters

Name	Type	Description
zoned	byte[]	Description

Return Values

java.lang.String
Returns the resultant string.

Throws

java.lang.NumberFormatException

zonedToString

Description

Convert the zoned decimal (COBOL PIC S9) byte array specified by **zoned** to a String using the specified encoding enc.

Syntax

```
public static java.lang.String zonedToString(byte[] zoned,
                                              java.lang.String enc)
```

Parameters

Name	Type	Description
zoned	byte[]	The byte array contains zoned decimal
enc	java.lang.String	The encoding used for conversion.

Note: *Methods that include an "encoding" parameter should specify encoding as "ISO-8859-1" when the e*Way is run on z/OS.*

Return Values

java.lang.String
Returns the resultant string.

Throws

java.lang.NumberFormatException

6.2 Packed Decimal Java Helper Methods

The Java helper methods for the toPackedDecimal class. These methods are not exposed in the Collaboration but are available for use. For example, to use the **CopyTo** method to obtain a Hex value in a byte array you would use the following code:

```
MyPacked.toPackedDecimal("327.00") ;
    System.out.println(  " Decimal value is : "  + MyPacked.toString() ) ;
    byte[] work_buf = new byte[7];
    MyPacked.CopyTo(work_buf, 7);
    for ( int  Ii = 0 ; Ii < 7 ; Ii++ )
    {
      int Ib = (int) new Byte(work_buf[Ii]).intValue();
      if (Ib < 16)
      {
        System.out.println("  Byte" + Ii  + " Hexvalue = 0" + Integer.toHexString(Ib));
      }
      else
      {
        System.out.println("  Byte" + Ii  + " Hexvalue = " + Integer.toHexString(Ib));
      }
    }
```

This produces the following output:

```
Decimal value is : 327.00
Byte0 Hexvalue = 00
Byte1 Hexvalue = 00
Byte2 Hexvalue = 00
```

```
Byte3 Hexvalue = 00
Byte4 Hexvalue = 32
Byte5 Hexvalue = 70
Byte6 Hexvalue = 0c
```

These methods are described in detail on the following pages:

ContainerExists

Description

Checks to see if contents of a packed decimal are available.

Syntax

```
public com.stc.eways.cics.PackedDecimal boolean ContainerExists()
```

Parameters

None.

Return Values

Boolean

Throws

None.

CopyBack

Description

Builds a packed decimal from the payload in a byte array. The byte array must contain a valid packed decimal number.

Syntax

```
public com.stc.eways.cics.PackedDecimal void CopyBack(byte Origin[],
                                                      int Size)
```

Parameters

Name	Type	Description
Origin[]	byte	Payload containing a valid packed decimal number.
Size	int	Size of the payload that contains the packed decimal number.

Return Values

None.

Throws

None.

CopyTo

Description

Copies a packed decimal number to a byte array.

Syntax

```
public com.stc.eways.cics.PackedDecimal void CopyTo(byte Dest[],
                                                    int Size)
```

Parameters

Name	Type	Description
Dest[]	byte	A byte array that the packed decimal number will be copied to.
Size	int	Size of the payload that contains the packed decimal number.

Return Values

None.

Throws

None.

GiveElem

Description

Returns a specified byte of a packed decimal number.

Syntax

```
public com.stc.eways.cics.PackedDecimal int GiveElem(int Ii)
```

Parameters

Name	Type	Description
Ii	int	Position of the byte to be returned.

Return Values

int

Returns the value of the position.

Throws

None.

SetElem

Description

Sets the value of a specified byte in a packed decimal number.

Syntax

```
public com.stc.eways.cics.PackedDecimal void SetElem(int Ii,
                                                     byte Value)
```

Parameters

Name	Type	Description
Ii	int	Position of the byte to be set.
Value	byte	Value to set.

Return Values

None.

Throws

None.

Index

A

C

D

E

F

H

I

J

M