CICS Transaction Gateway

CICS Transaction Gateway for z/OS: Programming Guide

Version 8.0

SC34-7059-02

CICS Transaction Gateway

CICS Transaction Gateway for z/OS: Programming Guide

Version 8.0

SC34-7059-02

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 151.

This edition applies to Version 8.0 of CICS Transaction Gateway, program number 5655-W10. It will also apply to all subsequent versions, releases, and modifications until otherwise indicated in new editions.

This edition replaces SC34-7059-01.

Contents

About this book

This information is an introduction to programming for CICS® Transaction Gateway and CICS Transaction Gateway Desktop Edition. It provides the information that you need to enable your user applications to interact with CICS server applications.

CICS Transaction Gateway provides secure, easy access to multiple CICS servers and supports user applications developed in C++, C, COBOL and COM based programming languages.

In addition, it supports remote access to CICS servers from user applications developed in Java. It provides support for the J2EE Connector Architecture (JCA) allowing user applications to be deployed into WebSphere® Application Server.

This book starts by describing the application programming interfaces (APIs) in a way that is independent of programming languages. It then gives guidance on programming in the supported languages.

Terminology

The term <install_path> is used in file paths to represent the directory where you installed the product. See the *CICS Transaction Gateway: Administration* book for your operating system, for the default installation locations.

What's new in CICS Transaction Gateway Version 8.0

CICS Transaction Gateway Version 8.0 includes important new enhancements for the product and the information centers.

Enhanced support for high availability

You can now use the high availability support provided by CICS Transaction Gateway with two-phase commit XA requests. You can create a sysplex wide highly available Gateway solution that supports XA transactions and maintains transactional integrity with CICS systems across the Parallel Sysplex®.

A new CICS request exit management verb CREXIT is supplied that enables dynamic interaction with the CICS request exit. The new verb enables you to perform dynamic tasks, such as influencing the dynamic server selection policies or controlling diagnostics.

New information is available in the CICS request exit to simplify the development of request retry logic. For more information about the description of the LastServer and LastError fields see RequestDetails in the JavaDoc.

Channels and containers support for remote C clients

You can use channels and containers when you connect to CICS using the IPIC protocol from a C Client application in remote mode.

.NET support

A new .NET API that allows remote clients to access CICS servers using COMMAREA-based ECI requests. ECI requests with channels and containers are not supported.

Identity propagation for user authentication

CICS Transaction Gateway can pass user security identity credentials, known as a *distributed identity*, over the network to CICS. The information is preserved in its original form for authorization by CICS and RACF®, and for subsequent accountability and trace purposes. Identity propagation provides a unified security solution when connecting applications in WebSphere Application Server to CICS.

ECI resource adapter support for 64-bit local mode and IPIC

The ECI resource adapters can now be deployed into a 64-bit application server and used in local mode with IPIC connections to CICS.

Additional product updates

- **JEE installation verification test suite:** JEE application server support is further extended by the provision of an installation verification test suite. This allows simple verification of the CICS JCA resource adapters for usage in any third-party application server compliant with the J2EE 1.4 specification.
- **Cold start option for resolving "in forget" units of recovery:** Transactions that are heuristically resolved on a recover flow are returned, whether or not a cold start of CICS Transaction Gateway is specified. A cold start option on the CICS Transaction Gateway start command ensures that "forget" calls are issued to RRS, if any transactions are found to be in an "in-forget" state.
- **Applets and SSL support:** Applets can connect securely to a Gateway daemon using Java Secure Sockets Extension (JSSE), the Java technology version of SSL. SSL support for applets is available for remote mode topologies.
- **Multi-sockets support for IPIC connections to CICS:** CICS Transaction Gateway supports IPIC protocol connections to CICS that use up to two TCP/IP sockets per connection. The availability of multi-sockets can reduce CPU usage and improve throughput when compared with the previous version of the product.
- **Additional statistics:** Two new statistics enable the identification of IPIC connections within a CICSplex. The new statistics identify the CICS IPIC connection definition APPLID and APPLID qualifier. A third new statistic identifies the total number of times a connection manager thread had been allocated to a client application, for the lifetime of the Gateway daemon.
- **New data available to request monitoring exits:** This data includes the distributed identity and the CICS correlator.
- **Support for Java:** CICS Transaction Gateway requires Java Version 6.0.

Information center enhancements

- **New configuration scenarios:**
  - **Configuring a secure autoinstalled IPIC connection:** This scenario uses real data to show how to configure an autoinstalled connection to CICS. A sample client application is provided for testing the connection.
  - **Configuring a secure predefined IPIC connection:** This scenario uses real data to show how to configure a predefined connection to CICS. A sample client application is provided for testing the connection.

  - **Configuring a highly available Gateway group with two-phase commit and IPIC:** This scenario uses real data to show how to set up a highly available Gateway group that can support XA transactions to CICS over IPIC.
  - **Configuring identity propagation for a remote mode topology:** This scenario uses real data to show how to configure identity propagation to provide a way of authorizing requests by associating security information in WebSphere Application Server with security information in CICS Transaction Server for z/OS®.
  - **Configuring SSL between a Java Client and the Gateway daemon:** This scenario uses real data to show how to configure SSL security on the connection between a Java Client and CICS Transaction Gateway.
- **Sample programs** Information about the sample programs shipped with the CICS Transaction Gateway has been added. This information was previously supplied as text file samples.txt in the <install-path>/samples directory.
- **Product overview** This section has been revised to provide an enhanced high-level introduction to the product.
- **Security** This section has been revised to improve the navigation, and to provide more information on the available security options for different types of connection to CICS.

Chapter 1. Programming overview

Programming information for CICS Transaction Gateway including information on APIs, ancillary functions, user applications, and supported programming languages.

Application programming interfaces

The CICS Transaction Gateway supports the integration of CICS systems and client systems. There is a standard set of functions to allow user applications to call CICS programs.

The Application Programming Interface (API) that is available to enable user applications to access and update CICS facilities and data is the External Call Interface (ECI).

There are also statistical data APIs, which enable a user application to collect statistical information about a running CICS Transaction Gateway.

Related information

Chapter 2, “External Call Interface (ECI),” on page 5
The External Call Interface (ECI) enables a client application to call a CICS program synchronously or asynchronously. It enables the design of new applications to be optimized for client/server operation, with the business logic on the server and the presentation logic on the client.

Chapter 3, “Statistics APIs,” on page 15
This topic describes the statistics APIs. If you want to use the statistics APIs in remote mode, you need to set up a statistics API protocol handler.

Ancillary functions

Several ancillary functions are provided with the CICS Transaction Gateway.

List CICS systems

To determine which CICS servers ECI requests can be directed to, user applications can query the CICS Transaction Gateway for a list of CICS systems.

The query returns a list of the CICS servers that have been defined within the CICS Transaction Gateway. There is no guarantee that communication links exist between the CICS servers and the CICS Transaction Gateway or that any of the CICS servers are actually available.

Code page information

When using the application programming interfaces of the CICS Transaction Gateway to start CICS programs, data conversion is an important consideration.

If the code page of the user application is different from the code page of the CICS server, or the byte order of binary data is in a different format, you might need to convert the data in a COMMAREA or container. You can do this conversion by using CICS supplied data conversion capabilities on the CICS server, provided by the DFHCCNV program and controlled by the DFHCNV macro definitions. In this case all data conversion is performed on the CICS server. Alternatively, you can use the data marshalling utilities provided within your user application development environment.

RACF user ID certificate mapping

The CICS Transaction Gateway provides a java class that can be used to map an X.509 certificate to a RACF user ID.

For more information about this utility see "CICS Transaction Gateway security classes" on page 48.

CICS Transaction Gateway Applications

The CICS Transaction Gateway supports running Client applications in both local and remote modes. Client applications enable access to CICS server transactions and programs from the host machine.

In local mode, the Client application is installed and run on the CICS Transaction Gateway host machine. In remote mode, the Client application is installed and run on a machine remote to the CICS Transaction Gateway host machine.

Client applications have the following capabilities in common:

- They can be written to access one or more CICS servers.
- They can connect to several CICS servers at the same time.
- They can have several program calls running concurrently.

Java Client applications use the Gateway classes to communicate with CICS servers. JCA applications use the J2EE CICS resource adapters to communicate with CICS servers.

Non-Java Client applications running in remote mode use the ECI version 2 C language bindings to communicate with CICS servers.

The following figure shows Client applications running in both local and remote mode on a z/OS system.

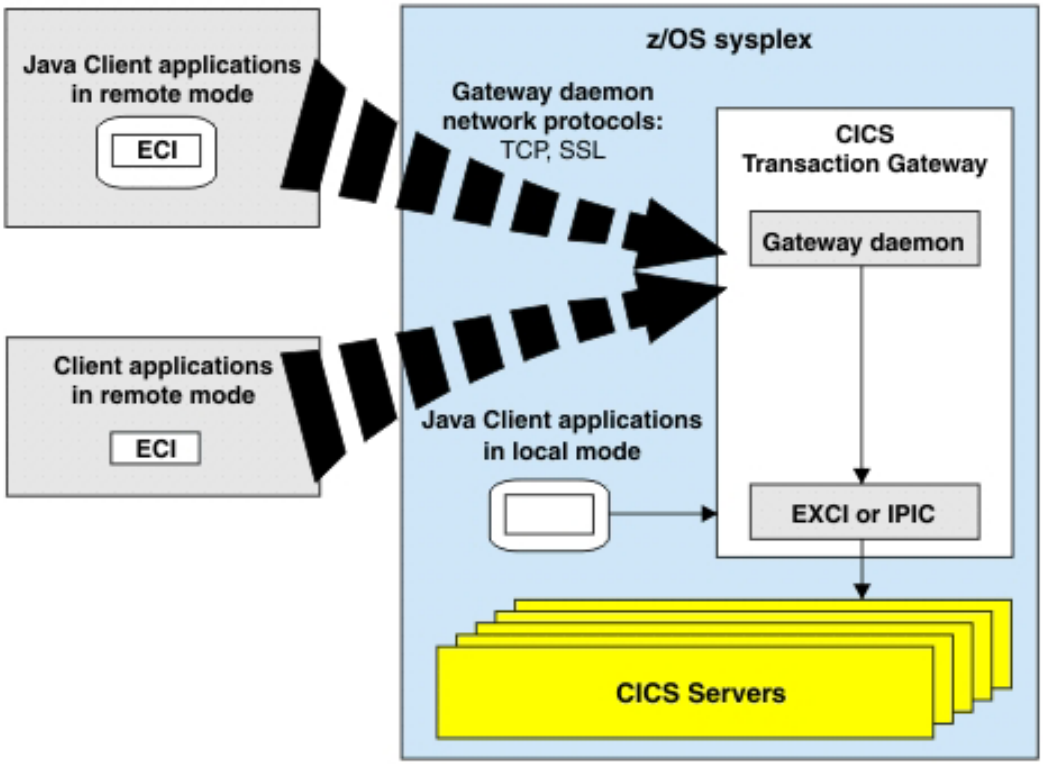


Figure 1. CICS Transaction Gateway for z/OS

Supported programming languages

This table shows which programming languages are supported by the CICS Transaction Gateway for each platform and application programming interface (API).

Table 1. Supported application programming languages for the CICS Transaction Gateway in local mode

API	C	C++	COBOL	COM	Java Support Classes	Java Base Classes	JCA
ECI						X	X

Table 2. Supported application programming languages for the CICS Transaction Gateway in remote mode

API	C	C++	COBOL	COM	Java Support Classes	Java Base Classes	JCA
ECI	X					X	X

J2EE Connector Architecture (JCA) applications

The CICS Transaction Gateway implements the JCA by providing J2EE CICS resource adapters.

These resource adapters support the J2EE Common Client Interface (CCI) defined by the JCA and are a middle-tier between JCA compliant applications and the CICS Transaction Gateway. The J2EE application server can be run locally on the same machine as the CICS Transaction Gateway, or remotely as shown in the following figure.

JCA compliant applications can be developed and deployed in a managed or nonmanaged environment. In a managed environment, JCA applications can exploit the quality of service provided by the J2EE application server.

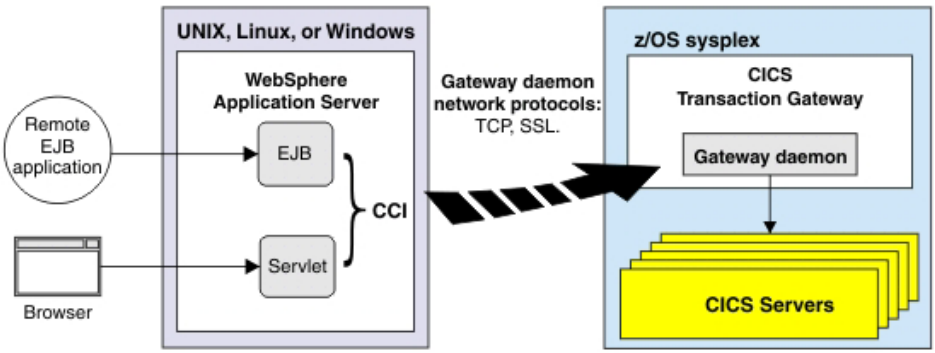


Figure 2. CICS Transaction Gateway with WebSphere Application Server in remote mode

Chapter 2. External Call Interface (ECI)

The External Call Interface (ECI) enables a client application to call a CICS program synchronously or asynchronously. It enables the design of new applications to be optimized for client/server operation, with the business logic on the server and the presentation logic on the client.

The external interfaces allow non-CICS applications to access and update CICS resources by calling CICS programs. When used in conjunction with CICS communication, the external interfaces enable non-CICS programs to access and update resources on any CICS system. This method of using the external interfaces supports such activities as the development of graphical user interface (GUI) front ends for CICS applications and it allows the integration of CICS systems and non-CICS systems.

The application can connect to several CICS servers at the same time and have several called CICS programs running concurrently. The CICS programs can transfer information using COMMAREAs or channels.

CICS programs that are invoked by an ECI request must follow the rules for distributed program link (DPL) requests. For information on DPL requests see the *CICS Application Programming Guide*. For information on the API restriction for DPL requests see Appendix G of the *CICS Application Programming Reference*.

Introduction to channels and containers

Channels and containers provide a method of transferring data between CICS programs, in amounts that far exceed the 32 KB limit that applies to communication areas (COMMAREAs).

Each container is a "named COMMAREA" that is not limited to 32 KB. Containers are grouped together in sets called channels.

The channel and container model has several advantages over the communication areas (COMMAREAs) used by CICS programs to exchange data:

- Unlike COMMAREAs, channels are not limited in size. Any number of containers can be added to a channel, and the size of individual containers is limited only by the amount of storage that you have available. Consider the amount of storage available to other applications when you create large containers.

- Because a channel can consist of multiple containers, it can be used to pass data in a more structured way, allowing you to partition your data into logical entities. In contrast, a COMMAREA is a monolithic block of data.
- Unlike COMMAREAs, channels do not require that the programs that use them to record and keep track of the exact size of the data returned.
- Channels can be used by CICS application programs written in any of the CICS-supported languages. For example, a Java client program on one CICS server can use a channel to exchange data with a COBOL server program on a back-end AOR.
- CICS automatically destroys containers and their storage when they go out of scope.

When you are using channels and containers in preference to COMMAREAs, note that:

- a channel can use more storage than a COMMAREA to pass the same data.
  1. Container data can be held in more than one place.
  2. COMMAREAs are accessed by pointer, whereas the data in containers is copied between programs.

For more information on using channels and containers in the JCA framework, see "The ECI resource adapters with channels and containers" on page 56.

For more information on using channels and containers with ECI calls for Java, see "Creating Java channels and containers for ECI calls" on page 41.

For more information on using channels and containers with C ECIv2 calls, see "Creating channels and containers for ECI calls for C" on page 79.

The ECI request

An ECI request is a call that a Client application makes to a CICS server program.

External calls to CICS

An ECI request calls a CICS program on a CICS server. This is known as making an external call to CICS and is the primary purpose of the ECI request. If no CICS server is selected, the default CICS server is used.

The ECI request can make four different types of call:

- Program link calls
- Status information calls
- Reply solicitation calls

- Callbacks

Related information

"Input and output information for external calls to CICS"
The input parameters passed to the CICS server with an ECI call, and the output parameters returned to the user application following an ECI call.

"Program link calls" on page 8
Program link calls cause the CICS mirror transaction to be attached to run a server program on the CICS server.

"Status information calls" on page 10
Status information calls retrieve status information about the connection between the client and server systems.

"Retrieving replies from asynchronous ECI requests" on page 10

Input and output information for external calls to CICS

The input parameters passed to the CICS server with an ECI call, and the output parameters returned to the user application following an ECI call.

Input parameters passed to the CICS server with an ECI call

The following input parameters can be passed to the CICS server with an ECI call:

CHANNEL
: A communication area used for passing containers to a server program.

COMMAREA
: A communication area used for passing input to a server program.

ECI timeout
: The maximum wait time for a response to an ECI request.

LUW control
: The way in which a Logical Unit of Work (LUW) is started, continued and ended.

LUW identifier
: A token which identifies the ECI call as part of an LUW.

Password
: The password provided for security checking on an ECI call.

Program name
: The name of a program to be run on a CICS server.

Server name
: The name of the CICS server that the ECI call is directed to. This can be a logical server or an actual server.

TPNName
: The transaction ID of the CICS mirror program.

TranName
: The transaction ID seen in the exec interface block (EIB) by the CICS mirror program.

Userid
The user ID provided for security checking on an ECI call.

Output parameters returned to the user application following an ECI call

The following output parameters can be returned to the user application following an ECI call.

Abend code
The code returned when a server program has ended abnormally.

CHANNEL
A communication area that holds containers passed from a server program.

COMMAREA
The communication area that contains output from a server program.

LUW identifier
A token which identifies the ECI call as part of an LUW.

Program link calls

Program link calls cause the CICS mirror transaction to be attached to run a server program on the CICS server.

ECI request program link calls can be synchronous or asynchronous:

Synchronous
Synchronous calls are blocking calls. The user application is suspended until the called server program has finished and a reply is received from CICS. The received reply is immediately available.

Asynchronous
Asynchronous calls are nonblocking calls. The user application gets control back without waiting for the called server program to finish. The reply from CICS can be retrieved later using one of the reply solicitation calls or a callback. See "Retrieving replies from asynchronous ECI requests" on page 10. An asynchronous program link call is outstanding until a reply solicitation call, or the callback, has retrieved the reply.

Synchronous and asynchronous program link calls can be nonextended or extended:

Nonextended
The CICS server program, not the user application, controls whether recoverable resources are committed or backed out. Each program link call corresponds to one CICS transaction. This is referred to as SYNCONRETURN.

Extended

The user application controls whether recoverable resources are committed or rolled back. Multiple calls are possible, allowing an LUW to be extended across successive ECI requests to the same CICS server. This is known as an *extended logical unit of work* (extended LUW).

CICS user applications are often concerned with updating recoverable resources. An LUW is the processing that a CICS server program performs between sync points. A sync point is the point at which all changes to recoverable resources that were made by a task since its last sync point are committed. LUW management is performed by the user application, using the *commit* and *rollback* functions:

Commit

Ends the current LUW and any changes made to recoverable resources are committed.

Rollback

Terminates the current LUW and backs out (rolls back) any changes made to recoverable resources since the previous sync point.

ECI-based communications between the CICS server and the CICS Transaction Gateway are known as conversations. A nonextended program link ECI call is one conversation. A series of extended ECI calls followed by a commit or rollback is one conversation.

Managing logical units of work

On a successful return from the first of a sequence of extended ECI calls for an LUW, the user application is returned an LUW identifier corresponding to an instance of a CICS mirror transaction.

Specifying this LUW identifier in subsequent ECI calls means that these calls will be processed by the same CICS mirror transaction. All program link calls for the same LUW are sent to the same server.

When the user application makes an ECI commit or rollback call, the CICS server attempts to commit or back out changes to recoverable resources. The user application is advised whether or not the attempt was successful. If an LUW is outstanding (incomplete), the user application issues an extended ECI commit or rollback call to the CICS server. If the execution of a user application completes without committing or rolling back an outstanding LUW, the CICS Transaction Gateway attempts to back out the LUW.

If an extended ECI call fails, the user application must check if a nonzero LUW identifier was returned. If so, this indicates that the LUW is still

outstanding and you must commit or rollback the LUW. If you do not, the problem is a lost communications link with the CICS server.

An ECI user application using an extended LUW might cause other user applications to be suspended waiting for CICS resources, which are held for the duration of the LUW.

Status information calls

Status information calls retrieve status information about the connection between the client and server systems.

The status of connected servers is updated as a result of requests being flowed and protocol specific events. The status returned is the last known state of connected servers, which might not be the same as the current state.

ECI request status link calls can be synchronous or asynchronous.

There are three types of status information call:

Immediate
Requests status information to be sent to the user application immediately it becomes available.

Change
Requests status information to be sent to the user application when the status changes from some specified value. Change calls are always asynchronous.

Cancel
Cancels an earlier **change** call.

Retrieving replies from asynchronous ECI requests

Callbacks

Callbacks enable the CICS server to drive specific function provided by the user application when an asynchronous program link call completes. This is the recommended way of handling replies from ECI requests.

ECI and CICS transaction IDs

The transaction ID of the mirror transaction for an ECI call can be controlled through the parameters TPNName and TranName.

Specify TPNName to change the name of the CICS mirror transaction that the called program will run under. For example, you can specify TPNName if you need a transaction definition with different attributes from those defined for the default mirror transaction. This option is like the TRANSID option on an EXEC CICS LINK command. The transaction ID is available to the server

program in the exec interface block (EIB). You must define a transaction on the CICS server for this transaction ID that points to the DFHMIRS program. Note that TPNName takes precedence if both TranName and TPNName are specified. If neither TPNName nor TranName is specified, the ECI Program Link call is attached to the default mirror transaction on the server. The default mirror transaction is CSMI.

If TranName is specified, the called program runs under the default mirror transaction, but is linked to under the TranName transaction ID. This name is available to the called program in the (EIB) for querying the transaction ID.

Table 3 shows the name of the CICS mirror transaction and the name stored in EIBTRNID according to whether or not TPNName and TranName are specified.

Table 3. Specifying TPNName and TranName

TPNName specified	**TranName specified**	**Mirror transaction name**	**Name in EIBTRNID**
Y	Y	TPNName	TPNName
Y	N	TPNName	TPNName
N	Y	default	TranName
N	N	default	default

Timeout of the ECI request

An ECI timeout is the time that the CICS Transaction Gateway will wait for a response to an ECI request sent to a CICS server before returning a timeout error to the Client application.

An ECI timeout can occur either before or after the ECI request has been sent to the CICS server, so there are two timeout conditions, request timeout and response timeout.

Request timeout

A request timeout occurs before the request has been forwarded to the CICS server. The requested program was not called, and no server resources have been updated.

This can happen for the following reasons:

- The call was intended to start, or be the whole of, a new LUW. The LUW is not started, and no recoverable resources are updated.

- The call was intended to continue an existing LUW. The LUW continues, but no recoverable resources are updated, and the LUW is still uncommitted.
- The call was intended to end an existing LUW. The LUW continues, no recoverable resources are updated, and the LUW is still uncommitted.

If a timeout occurs during an XA transaction it is recommended that the EJB sets the transaction to be rolled back.

Response timeout

A response timeout occurs after the request has been forwarded to the CICS server. It can happen to a synchronous call, an asynchronous call, or to the reply solicitation call that retrieves the reply from an asynchronous call.

This can happen for the following reasons:

- The call was intended to be the only call of a new LUW. The LUW was started, but the user application cannot determine whether updates were performed, and whether they were committed or backed out.
- The call was intended to end an existing LUW. The LUW has ended, but the user application cannot determine whether updates were performed, and whether they were committed or backed out.
- The call was intended to continue or to end an existing LUW. The LUW persists, and changes to recoverable resources are still pending.

If a timeout occurs during an XA transaction it is recommended that the EJB sets the transaction to be rolled back.

Security in the ECI

The ECI uses conversation-level security based on the SNA LU 6.2 model.

ECI security involves:

Authentication
Checking that the user ID and password information associated with an ECI call is valid.

Authorization
Checking that the authenticated user is allowed access to the requested resource. This check is performed on the CICS server.

The user application can set the user ID and password on an ECI request for a conversation with a specific CICS server. These values override any default values set for the server connection. For information about how to set the connection user ID and password, refer to the *CICS Transaction Gateway: Administration* book for your operating system.

IPIC support for ECI

IPIC connections do not support ECI State calls or asynchronous ECI calls that use message qualifiers. If you are using local mode, IPIC connections are not displayed in the CICS_ECIListSystems call.

IPIC does not support the following ECI calls:

- ECI_ASYNC, with a message qualifier (Callbackable objects are supported)
- ECI_ASYNC_TPN, with a message qualifier (Callbackable objects are supported)
- ECI_GET_REPLY
- ECI_GET_REPLY_WAIT
- ECI_GET_SPECIFIC_REPLY
- ECI_GET_SPECIFIC_REPLY_WAIT
- ECI_STATE_ASYNC
- ECI_STATE_ASYNC_JAVA
- ECI_STATE_CANCEL
- ECI_STATE_CHANGED
- ECI_STATE_IMMEDIATE
- ECI_STATE_SYNC
- ECI_STATE_SYNC_JAVA

If you are using local mode, IPIC servers are not displayed in a **CICS_EciListSystems** call. This is because the IPIC information is passed using a URL and is not known in advance of the connection. However, if you are using remote mode, you define your IPIC servers in the configuration file (the URL function is not available for remote mode), and the servers are displayed in the **CICS_EciListSystems** call.

ECI performance considerations when using COMMAREAs

The performance of ECI might be affected by the amount of data transmitted over the network in the COMMAREA between the client application and the CICS server.

To reduce the number of bytes transmitted over network protocols between the Gateway daemon and the CICS server the CICS Transaction Gateway removes trailing nulls from the COMMAREA before transmission and restores them again after transmission, this is referred to as null stripping. Null stripping is transparent to client application programs which always see the full-size COMMAREA.

The CICS server adds trailing nulls to the data received to extend it to the length specified in Commarea_Length so that the server program always receives a full COMMAREA. The CICS server also performs null stripping before transmitting the COMMAREA back over the network.

To reduce the number of bytes transmitted between a Client application and the Gateway daemon, functions are provided to set the length of data in the COMMAREA that is to flowed to the CICS server, COMMAREA outbound length, and to set the length of COMMAREA data returned from the Gateway daemon to the client application, COMMAREA inbound length.

For J2EE applications:

- the outbound COMMAREA length is set automatically by the CICS Transaction Gateway to remove trailing nulls
- use the setReplyLength and getReplyLength methods of the ECIInteractionSpec for the inbound COMMAREA length

For Java Client applications use the following methods:

- setCommareaOutboundLength
- setCommareaInboundLength
- getInboundDataLength

For ECI v2 applications use the **CTG_ECI_PARMS** parameter block fields:

- commarea_outbound_length
- commarea_inbound_length

For .NET applications use the EciRequest class fields:

- CommareaInboundLength
- CommareaOutboundLength

Chapter 3. Statistics APIs

This topic describes the statistics APIs. If you want to use the statistics APIs in remote mode, you need to set up a statistics API protocol handler.

Statistical data overview

The statistics API allows a single-threaded or multithreaded user application to access statistical data from one or more running Gateway daemons.

API functions

The API provides functions to:

- Connect to specific Gateway daemons, using gateway tokens.
- Disconnect from specific Gateway daemons, using gateway tokens.
- Obtain a set of statistical group IDs from a specific Gateway daemon.
- Obtain statistical IDs associated with one or more statistical group IDs from a specific Gateway daemon.
- Obtain data for statistical IDs from a particular Gateway daemon.

The functions are grouped into five categories:

- Connection functions
- ID data retrieval functions
- Statistical data retrieval functions
- Result set manipulation functions
- Utility functions

API and protocol version control

The API version represents the programming interface available from the ctgstats runtime library. The protocol version represents the set of responses that may be returned by a connected Gateway daemon in response to a statistics API function call. Comparison of compile time versus runtime values can be made to establish compatibility.

A statistics API application, and the Gateway daemon providing the statistics, might be from different versions of the CICS Transaction Gateway. API and protocol version control helps ensure that a statistics API application can issue meaningful requests to a CICS Transaction Gateway daemon, and get meaningful responses in return. API and protocol versions have a format of four digits, separated by the underscore character. For example: `1_0_0_0`

Note: The API and protocol versions might look like the product version, but they are not related. The statistics API can only be used to collect statistical data from Gateway daemons at version 7.0 or higher.

A statistics API application can:

- Find the API version that it was compiled with by using the compile-time string `CTG_STAT_API_VERSION`, defined in `ctgstats.h`.
- Find which API version is used at run time by a CICS Transaction Gateway daemon, by using the "getStatsAPIVersion" on page 31 function.
- Find which API version is used at run time by a CICS Transaction Gateway daemon, or Java statistics API by using the "getStatsAPIVersion" on page 31 function.
- Find the protocol version that it was compiled with by using the compile-time string `CTG_STAT_PROTOCOL_VER`, defined in `ctgstdat.h`.
- Find which protocol version is used at run time by a CICS Transaction Gateway daemon, by using the "openGatewayConnection" on page 24 or "openRemoteGatewayConnection" on page 25 function.

API version

The major version number, first digit, of the statistics API version must match between the application at compile time and ctgstats runtime library.

For example; if `CTG_STAT_API_VERSION` is 1_0_0_0 and the runtime function *getStatsAPIVersion* returns 1_1_0_0 then the major version (1_x_x_x) matches. Therefore the application is guaranteed to be runtime compatible with at least those functions available for version 1_1_0_0.

If the major version numbers differ, runtime compatibility is not guaranteed and API calls might fail.

Assuming that the major version number matches, then the minor version number (second digit) of the statistics API version at application compile time must be the lower than or equal to the ctgstats runtime library.

For example; if `CTG_STAT_API_VERSION` is 1_0_0_0 and the runtime function *getStatsAPIVersion* returns 1_1_0_0 then the major version (1_x_x_x) matches, and the minor version (x_0_x_x) used by the application is lower than the runtime library. Therefore, the application is guaranteed to be runtime compatible because it can only use those functions that are available at runtime version 1_0_0_0.

If the minor version number. second digit, of the statistics API version at application compile time is greater than the `ctgstats` runtime library, then some functions available at compile time will not be available at run time. The

3rd and 4th digits are reserved for IBM® service and maintenance usage.

Protocol version

The protocol version adheres to similar rules between compile time and run time as the API Version. However, the protocol version represents the interface between the compiled statistics application and the Gateway daemon connected at run time.

The major version number, first digit, of the protocol version must match between the application at compile time and the connected Gateway daemon.

Assuming that the major version number matches, then the minor version number, second digit, of the statistics API application at application compile time, must be the greater than or equal to the minor version number returned by the connected CICS Transaction Gateway daemon upon connection. If the minor version number is lower than that of the connected Gateway daemon, then the statistics API application might be unable to interpret all responses from function calls.

Statistics C API

The statistics C API enables a C client application to request statistics.

Calling the C API

This section explains how applications call API functions.

Applications call C API functions defined in "C language header files" on page 18, and a dynamic link library (DLL). Each function call returns an integer result code, defined in the `ctgstdat.h` header file. A function that completes normally returns the code `CTG_STAT_OK`. A function that needs to report a problem returns a negative code, detailed in the `ctgstdat.h` header file.

The statistics C API does not provide logging messages. Runtime operation of the C API functions can be monitored using trace facilities. Statistics C API tracing can be enabled programatically with data written to `stderr`, or a specified file. C API errors are reported to the calling application using an integer result code.

Statistics C API components

The statistics C API is made available to user applications by two C language header files and a dynamic link library (DLL).

C language header files

Two platform-independent C language header files are provided for developing user applications.

`ctgstats.h` defines the C API function calls and data types required to use the C API functions.

`ctgstdat.h` defines the set of query return codes that might be seen by a statistical user application. The set of query return codes can vary according to the statistics protocol version provided by the CICS Transaction Gateway daemon.

Runtime DLL

The statistics C API runtime DLL is provided for each of the supported CICS Transaction Gateway hardware platforms. It is supplied as a platform-specific DLL or shared library. It must be available during the run time of the statistical user application.

Sample code

A sample file `ctgstat1.c` is supplied. This provides a simple example for using the statistics C API. For further details, see "Statistics sample programs" on page 112.

Runtime components

This section describes the runtime components.

Data set names and SMP/E types

On z/OS, the runtime DLL and header file are delivered by SMP/E. The details are provided in the following table.

Table 4. Data set names and SMP/E types

Deliverable	Distribution	Target	Member	Type
DLL	*hlq*.ACTGMOD	*hlq*.SCTGDLL	CTGSTATS	++MOD
C Header	*hlq*.ACTGINCL	*hlq*.SCTGINCL	CTGSTATS	++SRC
C Header	*hlq*.ACTGINCL	*hlq*.SCTGINCL	CTGSTDAT	++SRC
C Sample	*hlq*.ACTGSAMP	*hlq*.SCTGSAMP	CTGSTAT1	++SRC
Sample JCL	*hlq*.ACTGSAMP	*hlq*.SCTGSAMP	CTGSTJOB	++SRC
Sidedeck	SMP/E *generated*	*hlq*.SCTGSID	CTGSTATS	Not applicable

The DLL load module is link-edited during installation. When the SCTGDLL library is added to the STEPLIB concatenation, user applications can use the statistics C API. If the application uses implicit DLL loading, the sidedeck might be required to complete the link-edit cycle.

Statistics C API program structure

Outline of a basic statistics C API program.

A basic statistics C API program typically has an outline similar to the example later in this section.

Example

This pseudo-code program connects to a CICS Transaction Gateway daemon, obtains the statistics IDs related to the "GD" resource group, obtains the current values for the given "GD" related statistical IDs and finally iterates through the returned values, writing out the details.

```
/* Create a connection to a local Gateway daemon */
openGatewayConnection(&gwyToken,port,&gwyProtocolVersPtr)

verify connected Gateway protocol level

/* Set the resource group id of interest */
queryString1="GD"

/* Obtain the list of associated statistical IDs */
getStatIdsByStatGroupId(gwyToken, queryString1, &resultSetToken)

/* Extract the returned IDs as a query string */
getIdQuery(resultSetToken,&queryString2)

/* Obtain the live statistical values for the given set IDs */
getStatsByStatId(gwyToken, queryString2, &resultSetToken)

/* Iterate over the result set, outputting */
/* the details of each result set element */

/* Obtain the first statistical result set element */
getFirstStat(resultSetToken, &statDataItem)

do
   if statDataItem.queryElementRC == CTGSTATS_SUCCESSFUL_QUERY
      /* output details of statDataItem */
   endif
   /* Obtain the next statistical result set element */
   getNextStat(resultSetToken, &statDataItem)
until end-of-resultset
```

C API data types

Data types defined and used by the statistics API.

This information describes the main data types used by the statistics C API.

Gateway tokens

A Gateway token represents a single connection to a specific Gateway daemon.

When a connection to a Gateway daemon is made, all subsequent C API calls that retrieve statistical data must include the Gateway token as a parameter.

The statistics C API handler in a Gateway daemon is restricted to five connection threads. This means that a single Gateway daemon can only deal with five connected statistics C API programs, or threads, at the same time.

A statistical C API program avoids holding more than one connection to the same Gateway daemon at the same time.

A statistical C API program can hold multiple Gateway tokens, but can only use them on the thread that called the "openGatewayConnection" on page 24 or "openRemoteGatewayConnection" on page 25 to retrieve the token.

A Gateway token type (`CTG_GatewayToken_t`) is defined in the "C language header files" on page 18.

Query strings

A query string is an input parameter, specifying the statistical data to be retrieved.

A query string is an input parameter to statistical C API functions which provide a result set token pointer. The string is a null-terminated, colon-separated list of IDs. The IDs can be statistical group IDs, or statistical IDs. An empty query string `""` is interpreted as matching all IDs appropriate to the function call.

Query strings are of type `(char *)`, and contain character data in the native encoding. The null terminator is added implicitly when creating strings in C using the `""` characters.

The user application creates and manages the query string character buffer.

Where an C API function produces a data result set, the function "getIdQuery" on page 29 can be used to obtain a query string suitable for input to another C API call.

Example

A pseudo-code example showing the query string used to retrieve the Gateway daemon status and all connection manager statistics is:

```
result = getStatsByStatId(gwyTok, "GD_CSTATUS:CM", &rsToken1);
```

Result set tokens

A result set token is a reference to a set of results from a single statistics C API function call.

If a statistics C API function calculates a set of data, the function provides a reference to the result set. This reference is called a result set token. The result set can contain either:

- ID data, including statistical group IDs or statistical IDs

or:

- Statistical data

A result set token is used to work with result set data. For example, a result set token enables a user application to browse through the result set, or extract specific details. The application can use functions such as “getFirstId” on page 29 or “getNextStat” on page 30 to manipulate the result set data.

An “ID data” on page 23 type is populated by the “getFirstId” on page 29 and “getNextId” on page 29 functions. A “Statistical data” on page 23 type is populated by the “getFirstStat” on page 30 and “getNextStat” on page 30 functions. The data types are used to access the data in the result sets, as described in “Correlating results and error checking” on page 34.

Note: All ID data and statistical data is in character format, using the default native string encoding.

Result set tokens returned by a statistics C API function are 'owned' by the C API. The token is freed when either:

- The associated Gateway daemon connection is closed using the “closeGatewayConnection” on page 26 function.

or

- The function “closeAllGatewayConnections” on page 26 is called.

The result set token returned by the “copyResultSet” on page 30 function is *not* 'owned' by the C API. The token can only be freed using the “freeResultSet” on page 31 function.

Result set tokens 'owned' by the C API cannot be 'freed' using the "freeResultSet" on page 31 function. The tokens must be freed using the "closeGatewayConnection" on page 26 or "closeAllGatewayConnections" on page 26 functions.

Result sets which are C API-owned can only be manipulated on the thread which obtained them. Result sets that were not created by C API calls can be manipulated by any thread.

Working with multiple result sets:

Working with multiple result sets requires special attention.

Calling a statistics C API function produces a result set token. This token identifies a result set owned by the statistics C API. The result set is also associated with the Gateway identified by the gateway token used during the function call. This means that each result set owned by the statistics C API is associated with a specific Gateway connection. It is helpful to think of the gateway token and the corresponding result set token as a pair.

Tokens referring to C API-owned result sets can only be used by the thread which created them. To create a result set token usable by any thread, call the "copyResultSet" on page 30 function.

For example, an application using the same gateway token to make two separate C API function calls will be given two logically different result set tokens. Since the same gateway token was used for both calls, the different result set tokens will iterate over the *same* result set. The result set will be the one returned by the last C API function call.

This means that the result set identified by an result set token is only valid until another C API call is made, specifying the same gateway token. The most recent C API call overwrites the existing result set.

Use the "copyResultSet" on page 30 function to make a copy of a result set before it is overwritten by another C API call. When the application finishes using the copied result set, free the storage using the "freeResultSet" on page 31 function.

Example

In the following example code, two statistics C API calls are made. The same Gateway token is used for both calls. Two separate addresses are supplied for the result set tokens.

```
getStatsByStatGroupId(gwyTok, "", &rsTok1);
/* Tasks after getStatsByStatGroupId function call. */
getStatsByStatId(gwyTok, "", &rsTok2);
/* Tasks after getStatsByStatId function call. */
```

Using the same Gateway token both calls means that the result set pointed to by `&rsTok1` will be overwritten when the second C API call is made. The two separate result set tokens `&rsTok1` and `&rsTok2` will iterate over the same result set.

If the result set obtained from the first C API call is still required later in the application, take a copy of the result set by calling the "copyResultSet" on page 30 function.

ID data

An ID data structure maps an individual result returned from an ID C API function.

The data type `CTG_IdData_t` is defined in the "C language header files" on page 18. The data provides a name for individual results within statistical groups or statistics.

Individual results can be accessed using the "getFirstId" on page 29 and "getNextId" on page 29 functions.

`CTG_IdData_t` provides two fields, a character pointer and length, to enable access to individual elements of an ID result set, as described in "Correlating results and error checking" on page 34.

Statistical data

A statistical data structure maps an individual result returned from a statistics C API function.

The data type `CTG_StatData_t` is defined in the "C language header files" on page 18. The statistical data represents individual statistics, or name-value pairs.

Individual results can be accessed using the "getFirstStat" on page 30 and "getNextStat" on page 30 functions.

`CTG_StatData_t` provides two fields, a character pointer and length, to enable access to individual elements of a statistical result set. These elements are the statistical ID and statistical value data, as described in "Correlating results and error checking" on page 34.

Statistics C API trace levels

The CICS Transaction Gateway statistics C API provides several levels of diagnostic trace information.

Trace levels

The CICS Transaction Gateway statistics C API can produce diagnostic trace information, depending on the trace level setting.

Each level automatically includes all the detail provided by the lower levels. For example, `CTG_STAT_TRACE_LEVEL2` indicates that all events and exceptions will be traced.

Table 5. Statistics C API Trace Levels

Trace level	Output details
`CTG_STAT_TRACE_LEVEL0`	No trace output.
`CTG_STAT_TRACE_LEVEL1`	Exceptions only.
`CTG_STAT_TRACE_LEVEL2`	Events.
`CTG_STAT_TRACE_LEVEL3`	Entries and exits.
`CTG_STAT_TRACE_LEVEL4`	Debug information.

The default trace destination is `stderr`. Use the function "setAPITraceFile" on page 33 to choose a different trace destination.

C API functions

The statistics C API functions.

Many ID functions create a result set. A result set is over-written the next time an ID function call is made using the same gateway token. This means an application working with several result sets from the same Gateway connection at the same time must take a local copy of each result set. To take a local copy of a result set, use the "copyResultSet" on page 30 function.

For details of the return codes provided by the C API functions, see `ctgstats.h` in the "C language header files" on page 18.

Gateway daemon connection functions

This information describes the main functions provided in the statistics API for connections to a Gateway daemon.

openGatewayConnection:

This function establishes a connection to a local Gateway daemon statistics protocol handler, using the specified port number, a pointer to a gateway token, and the address of a `char` pointer for the statistics C API protocol version.

Detail

This function is called with an integer for the target port number, a pointer to a gateway token, and the address of a `char` pointer to hold a string describing the version of the statistics C API protocol provided by the target gateway daemon.

The function creates a connection to a local Gateway daemon statistics protocol handler using the specified port number.

When the call returns, the gateway token represents the connection to the specified Gateway daemon. The token is required to interact with that Gateway daemon in subsequent C API calls.

The `char` pointer points to a null-terminated character string. The C API owns the storage for the protocol version character array, and the C API program does not free this storage.

The user application must check that the version of the statistics C API protocol provided by the target Gateway daemon is at least the same as major version number in the compile-time string `CTG_STAT_PROTOCOL_VER`. This compile-time string is defined in `ctgstdat.h`, described in the "C language header files" on page 18 section. The major version number is the first digit in the compile-time string.

openRemoteGatewayConnection:

This function establishes a connection to a remote Gateway daemon statistics protocol handler, using the specified host name, port number, a pointer to a gateway token, and the address of a `char` pointer for the statistics C API protocol version.

Detail

This function is called with:

- A character pointer for the host name. This is a null terminated string containing the IP address or host name of the machine running the Gateway daemon.
- An integer for the target port number.
- A pointer to a gateway token.

- The address of a char pointer to hold a string describing the version of the statistics C API protocol provided by the target gateway daemon.

The function creates a connection to a remote Gateway daemon statistics protocol handler using the specified port number.

When the call returns, the gateway token represents the connection to the specified Gateway daemon. The token is required to interact with that Gateway daemon in subsequent C API calls.

The char pointer points to a null-terminated character string. The C API owns the storage for the protocol version character array, and the C API program does not free this storage.

The user application must check that the version of the statistics C API protocol provided by the target Gateway daemon is at least the same as major version number in the compile-time string CTG_STAT_PROTOCOL_VER. This compile-time string is defined in ctgstdat.h, described in the "C language header files" on page 18 section. The major version number is the first digit in the compile-time string.

closeGatewayConnection:

This function closes a connection to a local Gateway daemon statistics protocol handler, using the gateway token provided.

Detail

This function is called with a pointer to a gateway token. The function closes the connection to the local or remote Gateway daemon statistics protocol handler identified by the gateway token. Any resources associated with the connection, including result sets, are freed, and result set tokens obtained with the specified gateway token are no longer valid.

When the call returns, the gateway token pointer is set to null, showing that it is no longer valid.

closeAllGatewayConnections:

This function releases all resources owned by the statistics C API, including any open Gateway daemon connections.

Detail

An application can use this function as part of a typical shutdown. The function can also be used in the event of a severe error, for example where some form of controlled shutdown is required but references to gateway tokens have been lost.

Copied result sets are not be freed by this function, because the C API does not own or maintain a record of copied result sets.

ID functions

This information describes the ID functions provided in the statistics C API.

getResourceGroupIds:

This function returns a result set token, representing the set of resource group IDs currently available for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set returned can be parsed with functions "getFirstId" on page 29 and "getNextId" on page 29, or used to generate a query string with "getIdQuery" on page 29.

Depending on when "getResourceGroupIds" is called, dynamic resource groups for a specific CICS server might not be returned in the list. The dynamic list of server resource group IDs can be obtained directly via the appropriate resource group statistical ID.

getStatIds:

This function returns a result set token, representing the set of all statistical IDs currently available for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set created can be parsed with functions "getFirstId" on page 29 and "getNextId" on page 29, or used to generate a query string with "getIdQuery" on page 29.

getStatIdsByStatGroupId:

This function returns a set of statistical IDs associated with the statistical group IDs supplied in the query string, for the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical group IDs, and a result set token pointer. The result set created can be parsed with functions “getFirstId” on page 29 and “getNextId” on page 29, or used to generate a query string with “getIdQuery” on page 29.

Retrieving statistical data functions

This information describes the data retrieval functions provided in the statistics C API.

getStats:

This function creates a result set token representing the set of all available statistical name-value pairs for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set created can be parsed with functions “getFirstStat” on page 30 and “getNextStat” on page 30, or used to generate a query string with “getIdQuery” on page 29.

getStatsByStatId:

This function creates a result set token. The token represents the set of name-value pairs that is generated when a query string of statistical IDs is applied to the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical IDs, and a result set token pointer. The result set created can be parsed with functions “getFirstId” on page 29 and “getNextId” on page 29, or used to generate a query string with “getIdQuery” on page 29.

getStatsByStatGroupId:

This function creates a result set token. The token represents the set of name-value pairs that is generated when a query string containing statistical group IDs is applied to the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical group IDs, and a result set token pointer. The result set returned can be

parsed with functions “getFirstStat” on page 30 and “getNextStat” on page 30, or used to generate a query string with “getIdQuery.”

Result set functions

This information describes the result set functions provided in the statistics C API.

getIdQuery:

This function provides a pointer to a character array, containing the ID result set.

Detail

This function is called with a result set token pointer, and the address of a character pointer. The function sets the pointer to point to a character array. This character array contains the ID result set, formatted for direct use as a query string.

The storage for the character array is created by the C API. The C API owns the storage for the character array, and the C API program does not free this storage.

getFirstId:

This function populates a `CTG_IdData_t` variable with details of the first ID in a result set.

Detail

This function is called with an ID result set token. The function populates a `CTG_IdData_t` variable with details of the first ID in the result set. If there are no further IDs in the result set, the `CTG_IdData_t` variable is unchanged.

For more information on the `CTG_IdData_t` data type, see “ID data” on page 23

getNextId:

This function populates a `CTG_IdData_t` variable with details of the next ID in a result set.

Detail

This function is called with an ID result set token. The function populates a CTG_IdData_t variable with details of the next ID in the result set. If there are no further IDs in the result set, the CTG_IdData_t variable is unchanged.

For more information on the CTG_IdData_t data type, see “ID data” on page 23

getFirstStat:

This function populates a CTG_StatData_t variable with details of the first ID in a result set.

Detail

This function is called with a statistical result set token. The function populates a CTG_StatData_t variable with details of the first ID in the result set. If there are no further IDs in the result set, the CTG_StatData_t variable is unchanged.

For more information on the CTG_StatData_t data type, see “Statistical data” on page 23.

getNextStat:

This function populates a CTG_StatData_t variable with details of the next ID in a result set.

Detail

This function is called with a statistical result set token. The function populates a CTG_StatData_t variable with details of the next ID in the result set. If there are no further IDs in the result set, the CTG_StatData_t variable is unchanged.

For more information on the CTG_StatData_t data type, see “Statistical data” on page 23.

copyResultSet:

This function creates a copy of a result set. The copy is owned by the calling application.

Detail

An application might need to make several C API calls on a result set. This is useful because some C API calls overwrite an existing result set with new results. A local copy of the result set is created using this function.

The `copyResultSet` function takes two result set tokens. The source token refers to the original result set. The target token refers to a copy of the result set. The copy is created by this function. The calling application owns the target result set.

There is no structural difference between the original and the target result sets. "Result set functions" on page 29 work with C API-owned result sets or application-owned result sets.

When the application finishes using the copied result set, free the storage using the "freeResultSet" function.

freeResultSet:

This function frees the storage used by an application-owned result set.

Detail

When an application finishes using a result set, the storage must be freed. This function takes a pointer to a result set token, frees the storage, and sets the pointer to null.

Use this function only for result sets created using the "copyResultSet" on page 30 function. If the result set is owned by the statistics C API, an attempt to free the result set returns an error.

Utility functions

This information describes the utility functions provided in the statistics C API.

getStatsAPIVersion:

This function provides version information about the statistics C API.

Detail

This function takes the address of a character pointer to be modified. The function modifies the character pointer to point to a null-terminated character array. The string represents the version of the active statistics DLL. Version information is described in "API and protocol version control" on page 15.

The C API owns the storage for the character array, and the C API program does not free this storage.

getAPITraceLevel:

This function provides information about the current trace status of the statistics C API.

Detail

This function takes a pointer to a local `int` variable. The function sets the variable to the current trace level of the statistics C API.

The levels are defined in the "C language header files" on page 18. Valid values are:

- `CTG_STAT_TRACE_LEVEL0`
- `CTG_STAT_TRACE_LEVEL1`
- `CTG_STAT_TRACE_LEVEL2`
- `CTG_STAT_TRACE_LEVEL3`
- `CTG_STAT_TRACE_LEVEL4`

For further information on trace levels, see "Statistics C API trace levels" on page 24.

setAPITraceLevel:

This function sets the trace level of the statistics C API.

Detail

This function takes an `int` value. The function sets the trace level of the C API to this value.

The default trace destination is `stderr`. Use the function "setAPITraceFile" on page 33 to choose a different trace destination.

The status values are defined in the "C language header files" on page 18. Valid values are:

- `CTG_STAT_TRACE_LEVEL0`
- `CTG_STAT_TRACE_LEVEL1`
- `CTG_STAT_TRACE_LEVEL2`
- `CTG_STAT_TRACE_LEVEL3`
- `CTG_STAT_TRACE_LEVEL4`

For further information on trace levels, see “Statistics C API trace levels” on page 24.

setAPITraceFile:

This function sets the destination for statistics C API trace details.

Detail

This function takes a character pointer to a null-terminated string. The string is the file name of the intended trace destination.

If the file name already exists, trace data is appended to the file.

If the file name cannot be opened for writing, trace data is sent to `stderr`.

Passing a null pointer to this function sets the trace destination back to `stderr`.

dumpResultSet:

This function outputs a result set in a printable form.

Detail

This function takes a result set token. The function writes the contents of the result set to the trace destination, regardless of the current trace level. The contents are written using printable characters.

This function is typically used for debug purposes.

Related reference

“Statistics C API trace levels” on page 24
The CICS Transaction Gateway statistics C API provides several levels of diagnostic trace information.

dumpState:

This function outputs internal information about the C API.

Detail

This function writes internal information about the C API to the trace destination.

This function is normally used for debug purposes.

Correlating results and error checking

Individual results within a result set from a statistics C API function call can be correlated back to the original query string data.

ID or statistical results within a result set from an C API call can be correlated back to the original query string data using the `struct` elements `queryElementPtr` and `queryElementLen`. The status of the result is given by `queryElementRC`. These return codes are defined in the `ctgstdat.h` header file.

After a call to "getFirstId" on page 29 or "getNextId" on page 29, the `CTG_IdData_t` elements `query` and `queryLen` represent the specific ID in the query string associated with the result.

After a call to "getFirstStat" on page 30 or "getNextStat" on page 30, the `CTG_StatData_t` elements `query` and `queryLen` represent the specific statistic in the query string associated with the result.

If the specific ID in the query string is in error, the `struct` element `queryElementRC` will have a non-zero value, defined in the `ctgstdat.h` header file.

Statistics Java API

The statistics Java API enables a Java-based client application to request statistics.

Calling the Java API

Applications can collect statistics from a Gateway using the Java classes in the com.ibm.ctg.client.stats package. The classes are supplied in a the ctgstats.jar and can be used with Gateways from V7.1 onwards. A sample file Ctgstat1.java is supplied that provides a simple example for using the Java statistics API.

Packaging restrictions with ctgstats.jar

If an application needs to use classes from both the com.ibm.ctg.client.stats package provided by ctgstats.jar and another API package supplied in ctgclient.jar, both jar files must be on the class path and must be from the same product version and release. The implication is that such an application can only connect to a Gateway daemon at the same version or higher for non-statistical requests.

The ctgstats.jar file can be used in isolation for standalone monitoring applications. Although there is compatibility with ctgclient.jar provided both ctgclient.jar and ctgstats.jar are from the same version, co-existence, on the

same class path, with versions of ctgclient.jar from earlier or future releases is not supported.

Sample code

A sample file Ctgstat1.java is supplied that provides a simple example for using the statistics API.

Java API classes

The Java API classes are responsible for connecting and making statistical requests to a statistics port provided by the Gateway daemon. The constructors allow the destination to be supplied by the application.

The statistic resource groups are available through the *getResourceGroupIds* method. An *IdResultSet* object is returned that contains a collection of *IdData* objects that hold the names of the resource groups. You can iterate over the *IdResultSet* to search the resource groups available.

If the names of the available statistics are required use the *getStatIds* method. This method returns an *IdResultSet*, functioning the same as *getResourceGroupIds*.

You can retrieve actual statistic values using the *getStats* method. This method returns a *StatResultSet* object that contains a collection of *StatData* objects. These *StatData* objects contain both the statistic names, and their current values. You can iterate over the *StatResultSet* to search the statistics available from the request.

If a result set returned has the return code set you can map this to the reason using the *getReturnString* method of the *ResponseData* class.

Tracing

You can enable statistics API tracing programmatically using the Java tracing options, see "Tracing in Java client programs" on page 45. Java API errors are reported to the calling application.

Related information

Package com.ibm.ctg.client.stats

Chapter 4. Programming in Java

This information provides an introduction to writing Java Client programs for the CICS Transaction Gateway.

Related information

"Supported programming languages" on page 3
This table shows which programming languages are supported by the CICS Transaction Gateway for each platform and application programming interface (API).

Overview of the programming interface for Java

The CICS Transaction Gateway enables Java Client applications to communicate with programs on a CICS server by providing base classes for the External Call Interface (ECI).

The following list of classes are the basic classes provided with the CICS Transaction Gateway. For a full description of all the classes and methods discussed in this section, see the Javadoc supplied with the CICS Transaction Gateway.

com.ibm.ctg.client.JavaGateway
: This class is the logical connection between a program and a CICS Transaction Gateway. You need a JavaGateway object for each CICS Transaction Gateway that you want to talk to.

com.ibm.ctg.client.ECIRequest
: This class contains the details of an ECI request to the CICS Transaction Gateway.

Writing Java Client applications

Before a Java Client application can send a request to the CICS server, it must create and open a JavaGateway object. The JavaGateway object is a logical connection between your application and the Gateway daemon when the application is running in remote mode. If a Java Client application is running in local mode, the JavaGateway is a connection between the application and the CICS server, bypassing the Gateway daemon.

When the JavaGateway is open, the Java Client application can flow requests to the CICS server using the flow method of the JavaGateway. When there are no more requests for the CICS Transaction Gateway, the Java Client application closes the JavaGateway object.

Use one of the constructors provided to create a JavaGateway. You must specify the protocol you are using, and the network address and port number of the remote Gateway daemon. You can specify this information either by using the setAddress, setProtocol and setPort methods, of the JavaGateway class, or by providing all the information in URL form: **Protocol:// Address:Port**. If you specify a local connection, you must specify a URL of **local:** You can use the setURL method or pass the URL into one of the JavaGateway constructors.

Note: The IP address can be in IPv6 format.

The JavaGateway supports the following protocols :

- TCP/IP
- SSL
- Local

There are several constructors available for creating a JavaGateway. The default constructor creates a JavaGateway with no properties. You must then use the set methods to set the required properties and the open method to open the Gateway. There are other constructors which set different combinations of properties and open the Gateway for you.

Java Client application suite select feature

Cipher suites define the key exchange, data encryption, and hash algorithms used for an SSL session between a client and server.

Cipher suites define the key exchange, data encryption, and hash algorithms used for an SSL session between a client and server. During the SSL handshake, both sides present the cipher suites that they are able to support and the strongest one common to both sides is selected. In this way, you can restrict the cipher suites that a Java Client application presents. CICS Transaction Gateway uses cipher suites provided by the Java runtime environment for the SSL protocol. The cipher suites available to be used are dependant on the Java version. See the documentation supplied with your Java runtime environment for valid cipher suites.

Restricting cipher suites for a Java Client application

To restrict the cipher suites used by a JavaGateway object, use the **`setProtocolProperties()`** method to add the property **`(JavaGateway.SSL_PROP_CIPHER_SUITES)`** to the properties object passed to it. The value of the property must contain a comma-separated list of the cipher suites that the application is restricted to using.

For example:

```
Properties sslProps = new Properties();
sslProps.setProperty(JavaGateway.SSL_PROP_KEYRING_CLASS, strSSLKeyring);
sslProps.setProperty(JavaGateway.SSL_PROP_KEYRING_PW, strSSLPassword);
sslProps.setProperty(JavaGateway.SSL_PROP_CIPHER_SUITES,
                     "SSL_RSA_WITH_NULL_SHA");
javaGatewayObject = new JavaGateway(strUrl, iPort, sslProps);
```

Deploying remote Java Client applications

Remote Java client applications are deployed to the runtime environment as Java Archive (.jar) files.

You are licensed to copy the following files to the computer that is running the Java Client application:

Non-J2EE applications
File ctgclient.jar

J2EE applications in a managed environment
The resource adapters (RAR files) in the <install_path>\deployable directory.

J2EE applications in a non-managed environment
The following files in the <install_path>\classes directory:

cicsj2ee.jar
ctgclient.jar
ccf2.jar
connector.jar
screenable.jar

Ensure that any JAR files that you copy are listed on the class path of the remote computer.

JavaGateway security

When you connect to a remote CICS Transaction Gateway, resources allocated to a particular connection, and identifiers specified on the request objects on a particular connection, are available only to that connection.

If you specify the **local:** protocol, all JavaGateways that are created in the same JVM, that is, the same process, have access to each other's allocated resources or specified identifiers.

Making ECI calls from a Java Client program

This section describes how to run a program on a CICS server using ECI calls from a Java Client application.

Use the com.ibm.ctg.client.ECIRequest base class and the JavaGateway flow method to pass details of an ECI request to the CICS Transaction Gateway.

The following table shows Java objects corresponding to the ECI terms described in "Input and output information for external calls to CICS" on page 7.

Table 6. ECI terms and corresponding Java objects

ECI term	Java object.field or object.method()
Abend code	ECIRequest.Abend_Code
Channel	setChannel(Channel)
COMMAREA	ECIRequest.Commarea
ECI timeout	ECIRequest.setECITimeout(short)
LUW control	ECIRequest.Extend_Mode
LUW identifier	ECIRequest.Luw_Token
Password	ECIRequest.Password
Program name	ECIRequest.Program
Server name	ECIRequest.Server
SocketConnectTimeout	ECIRequest:SocketConnectTimeout
TPNName	ECIRequest.Call_Type = ECI_SYNC_TPN or ECI_ASYNC_TPN and ECIRequest.Transid
TranName	ECIRequest.Call_Type = ECI_SYNC or ECI_ASYNC and ECIRequest.Transid
User ID	ECIRequest.Userid

Linking to a CICS server program

A link to a CICS program is made using an ECIRequest constructor to set the required parameters for the ECI call.

You can either use the default constructor which sets all parameters to their default values, or one of the other constructors which allow you to set different combinations of parameters. Place any data to be passed to the server program in a COMMAREA or container.

You can create ECI requests for synchronous and asynchronous program link calls by setting the value of Call_Type to ECI_SYNC or ECI_ASYNC.

If you use the ECI_ASYNC call type with CICS Transaction Gateway for z/OS, you must use the Callbackable interface.

Creating Java channels and containers for ECI calls

You can use channels and containers when you connect to CICS using the IPIC protocol. You must construct a channel before it can be used in an ECIRequest.

1. Add the following code to your application program, to construct a channel to hold the containers:

   ```
   Channel myChannel = new Channel("CHANNELNAME");
   ```

2. You can add containers with a data type of BIT or CHAR to your channel. Here is a sample BIT container:

   ```
   byte[] custNumber = new byte[]{0,1,2,3,4,5};
   myChannel.createContainer("CUSTNO", custNumber);
   ```

   And a sample CHAR container:

   ```
   String company = "IBM";
   myChannel.createContainer("COMPANY", company);
   ```

3. The channel and containers can now be used in an ECIRequest, as the example shows:

   ```
   ECIRequest eciReq = new ECIRequest("CICSA", "USERNAME", "PASSWORD",
   "CHANPROG", channel, ECIRequest.ECI_NO_EXTEND, 0);
   jgateway.flow(eciReq);
   ```

4. When the request is complete, you can retrieve the can retrieve the contents of the containers in the channel by interpreting the type, as the example shows:

   ```
   Channel myChannel = eciReq.getChannel();

   for(Container container: myChannel.getContainers()){
      System.out.println(container.getName());

      if (container.getType() == ContainerType.BIT){
         byte[] data = container.getBITData();
      }
      if (container.getType() == ContainerType.CHAR){
         String data = container.getCHARData();
      }
   }
   ```

Managing an LUW

Set the extend mode to ECI_EXTENDED if the ECI call is part of an extended LUW. If the call is the last, or only call for the LUW, the extend mode must be ECI_NO_EXTEND, ECI_COMMIT or ECI_BACKOUT.

Retrieving replies from asynchronous requests

Replies to asynchronous requests can be retrieved by using callbacks or reply solicitation calls.

Callbacks

ECIRequest supports callback objects. A callback object, which must implement the **Callbackable** interface, receives the results of the flow via the setResults method. When the results have been applied, a new thread is started to execute the run method.

If you specify a callback object for a synchronous call the results are passed to your Callbackable object as well as to your ECIRequest object in the flow request.

Reply solicitation calls

Use the **automatic message qualifier generator** feature of ECIRequest to ensure that the message qualifiers that you assign are unique within the CICS Transaction Gateway.

Turn the feature on by invoking the method setAutoMsgQual(true) on your ECIRequest object. This will assign a message qualifier that is unique on all asynchronous requests (ECI_ASYNC, ECI_ASYNC_TPN, ECI_STATE_ASYNC, ECI_STATE_ASYNC_JAVA), when the request is flowed. Use this message qualifier to retrieve replies when you use the ECI_GET_SPECIFIC_REPLY and ECI_GET_SPECIFIC_REPLY_WAIT call types.

For remote connections you cannot get replies on a different connection to the one that flowed the original request with a message qualifier.

If you use ASYNC calls with message qualifiers, you might have to pass a user ID and password when you retrieve the reply with one of the various GET_REPLY call types. The user ID and password are not used to validate whether the reply can be retrieved; they are passed to the Gateway to hold in case security is required to clean up (BACKOUT) an LUW if the connection is lost while the server program is still running.

For a local connection, the message qualifier must be unique for each request, although this is not enforced. Provided the JavaGateways are within the same JVM, any connection can get a message using a message qualifier, even if the request was flowed over a different connection. However, it is recommended that you use automatic message qualifier generation:

- To avoid problems resulting from reusing the same message qualifier
- To allow you to switch your application between local and remote connection

The following considerations apply:

- You cannot use the variable Message_Qualifier, or the methods isAutoMsgQual(), setAutoMsgQual(), setMessageQualifier(), or setMessageQualifier()

- You cannot use reply solicitation call types such as GET_REPLY, GET_REPLY_WAIT, GET_SPECIFIC_REPLY, or GET_SPECIFIC_REPLY_WAIT

ECI timeouts

Java methods cannot be used for setting ECI request timeout values in some situations.

When an EXCI connection to CICS is used by an ECI application either through a Gateway daemon or in local mode, you cannot use the methods getECITimeout(), or setECITimeout(). You can set the TIMEOUT parameter in the EXCI options table DFHXCOPT.

For more information on ECI timeouts, see the *CICS External Interfaces Guide*.

See "Timeout of the ECI request" on page 11

ECI return codes and server errors

This section describes how the return codes from the EXCI are returned to the user of the **ECIRequest** object.

The following table shows how EXCI return codes map to ECI return codes. The EXCI return codes are documented in the *CICS External Interfaces Guide*.

Table 7. EXCI return codes and ECI return codes

EXCI return codes	ECI symbolic names/return codes	rc
201, 203	ECI_ERR_NO_CICS	–3
202	ECI_ERR_RESOURCE_SHORTAGE	–16
401, 402, 403, 404, 410, 411, 412, 413, 418, 419, 421	ECI_ERR_SYSTEM_ERROR	–9
422	ECI_ERR_TRANSACTION_ABEND	–7
423	ECI_ERR_SECURITY_ERROR	–27
601, 602, 603, 604, 605, 606, 607, 608, 621, 622, 623, 627, 628	ECI_ERR_SYSTEM_ERROR	–9
609	ECI_ERR_SECURITY_ERROR	–27
624	ECI_ERR_REQUEST_TIMEOUT	–5

EXCI support

Version 2 of the EXCI is supported, and it provides support for **eci_transid** and resolves previous problems with ASCII/EBCDIC conversion.

If you use EXCI Version 2 and **eci_tpn** is specified on the ECI request, then the definition of the user mirror transaction must now specify PROGRAM(DFHMIRS), regardless of whether the transaction is defined as local or remote.

EPI and z/OS

The EPI classes are not available for z/OS. If you want to run transactions in the manner of the EPI, use the ECI and set up a request for DFHWBTTA. This is described in the *CICS Internet Guide*.

Compiling and running a Java Client application

Issues to consider when compiling and running a Java client application include performance, the Java class path and whether or not you are running a Web browser on the same machine as CICS Transaction Gateway.

Performance issues

There are several performance issues to consider when you run Java client applications.

The Java Virtual Machine (JVM) allocates a fixed size of stack space for each running thread in an application. You can usually control the amount of space that Java allocates by setting limits on the following sizes:

- The native stack size, allocated when running native JIT (Just-In-Time) compiled code.
- The Java stack size, allocated when running Java Bytecode.
- The initial Java heap size.
- The maximum Java heap size.

How you set these limits depends on your JVM. See your Java documentation for more information.

Setting up the CLASSPATH

Before you write any Java client programs, update the CLASSPATH environment variable to include the jar files supplied with CICS Transaction Gateway.

```
CLASSPATH = <install_path>/classes/ctgclient.jar;
            <install_path>/classes/ctgserver.jar
```

The ctgserver.jar file is required in CLASSPATH only for JavaGateways using the local URL.

Unable to load class that supports TCP/IP

If Java attempts to use class files from the local file system, this contravenes security rules and generates an exception.

Symptom

The following error occurs when running applications:

```
ERROR: java.io.IOException:

CTG6664E: Unable to load relevant class to support the tcp protocol
```

Probable cause

You are using a Web browser and CICS Transaction Gateway on the same workstation, and the ctgclient.jar and ctgserver.jar are referenced in the CLASSPATH setting.

Java searches the CLASSPATH environment variable before downloading classes across the network. If the required class is local, Java attempts to use it. However, use of class files from the local file system contravenes the application security rules, and generates an exception.

Action

Edit the CLASSPATH setting to remove ctgclient.jar and ctgserver.jar.

Problem determination for Java Client programs

Use tracing to help determine the cause of any problems when running Java clients.

Tracing in Java client programs

You can control tracing in Java client programs using the following calls and properties.

- calls to the com.ibm.ctg.client.T class

  For example, from within a user application:

  ```
  if (getParameter("trace") != null)
   {
     T.setOn(true);
   }
  ```

  where `trace` is a startup parameter that can be set on the user program.
- Gateway.T system properties

  For example:

  ```
  java -Dgateway.T=on com.usr.smp.test.testprog1
  ```

  which specifies full debug for `testprog1`.

For more information on the use of system properties, see your Java documentation.

It is recommended that applications implement an option to turn trace on.

The following is an explanation of the various trace levels available. The names of calls and properties are case sensitive.

Trace level
Standard

com.ibm.ctg.client.T call
T.setOn (true/false)

System property
gateway.T.trace=on

Definition
The standard option for application tracing.

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.

Trace level
Full Debug

com.ibm.ctg.client.T call
T.setDebugOn (true/false)

System property
gateway.T=on

Definition
The debugging option for application tracing.

By default, it traces out the whole of any data blocks. The trace contains more information about the CICS Transaction Gateway than the standard trace level.

This trace level is equivalent to the Gateway debug trace set by the `ctgstart –x` option.

Trace level
Exception Stacks

com.ibm.ctg.client.T call
T.setStackOn (true/false)

System property
gateway.T.stack=on

Definition
The exception stack option for application tracing.

It traces most Java exceptions, including exceptions which are expected during typical 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.

You can further configure the tracing by using the following options:

com.ibm.ctg.client.T call
T.setTFile(true,*filename*)

System property
gateway.T.setTFile=*filename*

Option usage
The value *filename* specifies a file location for writing of trace output. This is as an alternative to the default output on stderr. Long file names must be surrounded by quotation marks, for example: "trace output file.log"

com.ibm.ctg.client.T call
T.setTruncationSize(*number*)

System property
gateway.T.setTruncationSize=*number*

Option usage
The value *number* specifies the maximum size of any data blocks that will be written in the trace. Any positive integer is valid. If you specify a value of *0*, then no data blocks will be written in the trace. If a negative value is assigned to this option the exception java.lang.IllegalArgumentException will be raised.

com.ibm.ctg.client.T call
T.setDumpOffset(*number*)

System property
gateway.T.setDumpOffset=*number*

Option usage
The value *number* 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* will be used. If a negative value is assigned to this option the exception java.lang.IllegalArgumentException will be raised.

com.ibm.ctg.client.T call
T.setTimingOn (true/false)

System property
gateway.T.timing=on

Option usage
Specifies whether or not to display time-stamps in the trace.

Use the options in addition to one of the directives to switch tracing on.

For example, the following switches standard tracing on, and sets the maximum size of any data blocks to be dumped to 20 000 bytes:

```
java -Dgateway.T.trace=on -Dgateway.T.setTruncationSize=20000
```

Security for Java Client programs

CICS Transaction Gateway provides the Java classes for implementing security. Java provides the Security Manager.

CICS Transaction Gateway security classes

The CICS Transaction Gateway provides the following classes (security exits) for implementing security.

com.ibm.ctg.security.JSSEServerSecurity
Use this interface to allow the exposure of of X.509 Client Certificates when using the JSSE protocol.

See your JSSE, or Java, documentation for information on using X.509 certificates.

com.ibm.ctg.security.ServerSecurity
Use this interface for server-side security classes that do not require the exposure of SSL Client Certificates.

com.ibm.ctg.security.ClientSecurity
Use this interface for all client-side security classes.

com.ibm.ctg.util.RACFUserid
This class tries to map an X.509 Client Certificate to a RACF user ID. The certificate must already be associated with a valid RACF user ID.

The **JSSEServerSecurity** and **ServerSecurity** interfaces and partner **ClientSecurity** interface define a simple yet flexible model for providing security when using CICS Transaction Gateway. Implementations of the interfaces can be as simple, or as robust, as necessary; from simple XOR (exclusive-OR) scrambling to use of the Java Cryptography Architecture.

The JSSEServerSecurity interface works in conjunction with the Secure Sockets Layer (SSL) protocol. The interface allows server-side security objects access to a Client Certificate passed during the initial SSL handshake. The exposure of

the Client Certificate depends on the the CICS Transaction Gateway being configured to support Client Authentication.

An individual JavaGateway instance has an instance of a ClientSecurity class associated with it, until the JavaGateway is closed. Similarly, an instance of the partner JSSEServerSecurity or ServerSecurity class is associated with the connected Java client, until the connection is closed.

The basic model consists of:

- An initial handshake to exchange pertinent information. For example, this handshake could involve the exchange of public keys. However, at the interface level, the flow consists of a simple byte-array, therefore an implementation has complete control over the contents of its handshake flows.
- The relevant ClientSecurity instance being called to encode outbound requests, and decode inbound replies.
- The partner JSSEServerSecurity or ServerSecurity instance, being called to decode inbound requests and to encode outbound replies.

  The inbound request, and Client Certificate, is exposed via the **afterDecode()** method. For JSSE, the afterDecode() method exposes the GatewayRequest object, along with the **javax.security.cert.X509Certificate[]** certificate chain object.

ClientSecurity, JSSEServerSecurity, or ServerSecurity class instances maintain as data members sufficient information from the initial handshake to correctly encode and decode the flows. At the server, each connected client has its own instance of the ServerSecurity implementation class.

Using a Java 2 Security Manager

Java 2 provides a Security Manager system that controls access to Java resources.

The Security Manager restricts access to Java resources using a security policy. Some examples of protected resources are: reading a file, and opening a network socket. When a program tries to access a protected resource, the Java Security Manager verifies that both the code trying to access the resource, and, possibly, the caller of that code, have appropriate permissions. Without these permissions, the program cannot run.

If you are using any of the CICS Transaction Gateway Java APIs under a Java 2 security environment (such as a J2EE server), your application needs Java permissions to run correctly. The only exception to this is if you are using the J2EE APIs in a managed environment.

Figure 3 shows the minimum permissions that your application needs to use Gateway Java APIs. It might need additional permissions to run correctly.

```
java.net.SocketPermission "*", "resolve";
java.util.PropertyPermission "*", "read";
java.io.FilePermission "${user.home}${file.separator}ibm${file.separator}
  ctg${file.separator}-","read,write,delete";
java.lang.RuntimePermission "loadLibrary.*", "";
java.lang.RuntimePermission "shutdownHooks", "";
java.lang.RuntimePermission "modifyThread", "";
java.lang.RuntimePermission "modifyThreadGroup", "";
java.lang.RuntimePermission "readFileDescriptor", "";
java.lang.RuntimePermission "writeFileDescriptor", "";
java.security.SecurityPermission "putProviderProperty.IBMJSSE", "";
java.security.SecurityPermission "insertProvider.IBMJSSE", "";
java.security.SecurityPermission "putProviderProperty.IBMJCE", "";
java.security.SecurityPermission "insertProvider.IBMJCE", "";
javax.security.auth.PrivateCredentialPermission "* * \"*\"","read";
java.lang.RuntimePermission "accessClassInPackage.sun.io", "";
```

Figure 3. Required Java 2 Security Manager permissions

Permissions to access the file system

Depending on the functions performed by your program, the CICS Transaction Gateway Java APIs might require access to the file system, for example to write trace files.

The following permission statement gives permission for the CICS Transaction Gateway to access and create an `ibm/ctg` subdirectory in the users' home directory on the Unix System Services file system:

```
permission java.io.FilePermission "${user.home}${file.separator}ibm
${file.separator}ctg${file.separator}-","read,write,delete";
```

The format of the permission might vary depending on the installation, and you can specify alternative locations, or none at all. CICS Transaction Gateway classes require access to the file system in the following cases:

- For writing trace information to a file
- For accessing key rings, if you are using JSSE for your SSL protocol implementation

  See the information about Network security management in the *CICS Transaction Gateway: z/OS Administration* for information on how JSSE is selected as the implementation.

For example, you can specify the following permission to allow access to the directory `/tmp/ibm` and all subdirectories:

```
permission java.io.FilePermission "/tmp/ibm/",
                                  "read,write,delete";
```

Chapter 5. Programming using the J2EE Connector Architecture

This information describes how to program using the ECI resource adapters provided by the CICS Transaction Gateway.

Related information

"Supported programming languages" on page 3
This table shows which programming languages are supported by the CICS Transaction Gateway for each platform and application programming interface (API).

Overview of the programming interface of the J2EE Connector Architecture

The purpose of the J2EE Connector Architecture (JCA) is to connect Enterprise Information Systems (EISs), such as CICS, into the J2EE platform. The JCA offers a number of qualities of service which can be provided by a J2EE application server. These qualities of service include security credential management, connection pooling and transaction management.

Qualities of service are provided through system level contracts between a resource adapter provided by CICS Transaction Gateway and the J2EE application server. In most cases there is no need for any extra program code to be provided, therefore the programmer is free to concentrate on writing business code and need not be concerned with quality of service. For information about the provided qualities of service and configuration guidance, refer to the appropriate documentation for the J2EE application server.

The JCA defines a programming interface called the Common Client Interface (CCI). This interface can be used, with minor changes, to communicate with any EIS. CICS Transaction Gateway provides resource adapters which implement the CCI for interactions with CICS.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapters.

The Common Client Interface (CCI)

The CCI is a high-level programming interface defined by the J2EE Connector Architecture (JCA).

The CCI is available to J2EE developers who want to use the External Call Interface (ECI) to enable client applications to communicate with programs running on a CICS server.

The CCI has two class types:

- Generic CCI classes used for requesting a connection to an EIS such as CICS, and for executing commands on that EIS, passing input and retrieving output. These classes are generic because they do not pass information specific to a particular EIS. Examples are Connection and ConnectionFactory.
- CICS-specific classes used for passing specific information between the Java Client application and CICS. Examples are ECIInteractionSpec and ECIConnectionSpec.

The programming interface model

Applications using the CCI have a common structure, independent of the EIS that is being used. The JCA defines Connections and ConnectionFactories which represent the connection to the EIS. These objects allow a J2EE application server to manage security, transaction context, and connection pools for the resource adapter.

An application must start by obtaining a ConnectionFactory from which a Connection can be obtained. The properties of this Connection can be overridden by a ConnectionSpec object. The ConnectionSpec class is CICS-specific, so it can be either an ECIConnectionSpec or an EPIConnectionSpec.

After a connection has been obtained, an Interaction can be created from the Connection to make a particular request. As with the Connection, Interactions can have custom properties set by the CICS-specific InteractionSpec class (ECIInteractionSpec or EPIInteractionSpec). To perform the Interaction, call the execute() method and use CICS-specific Record objects to hold the data. For example:

```
Obtain a ConnectionFactory
Connection c = cf.getConnection(ConnectionSpec)
Interaction i = c.createInteraction()
InteractionSpec is = newInteractionSpec();
i.execute(spec, input, output)
```

The ConnectionFactory can be obtained in two ways:

- If you are using a J2EE application server, the ConnectionFactory is normally created from the resource adapter by means of an administration interface. This ConnectionFactory has custom properties set for it, for example the Gateway to be used would be set as a ConnectionURL. When the ConnectionFactory has been created, it can be made available for use by any enterprise applications through JNDI. This type of environment is called a managed environment. A managed environment allows a J2EE application server to manage the qualities of service of the connections. Consult your J2EE application server's documentation for information about deployment into a managed environment.

- If you are not using a J2EE application server, you must create a ManagedConnectionFactory and set its custom properties. You can then create a ConnectionFactory from the ManagedConnectionFactory. This type of environment is called a non-managed environment. A non-managed environment does not allow a J2EE application server to manage connections.

Record objects

Record objects are used to represent data passing to and from the EIS.

This is a representation of a COMMAREA or channels and containers, and a sample Record is provided for the ECI. It is recommended that application development tools are used to generate these Records.

ECI resource adapters

The ECI resource adapters provide a high level CCI interface to the ECI for sending ECI requests to CICS.

The ECI resource adapters are used for connecting to CICS server programs and for passing data to COMMAREAs or channels and containers without having to issue ECI requests. The resource adapters can be deployed into a J2EE application server to allow J2EE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the J2EE application server not the application.

Two resource adapters are supplied: cicseci.rar and cicseciXA.rar.

1. Adapter cicseci.rar supports the LocalTransaction interface, and global transactions in local mode under WebSphere.
2. Adapter cicseciXA.rar supports both XA and local transactions.

Use the cicseciXA.rar resource adapter for two-phase commit function with IPIC. For one-phase commit function you can use the cicseciXA.rar resource adapter, however performance might be improved by using cicseci.rar resource adapter. See "Transaction management" on page 61 for details of the transaction management models that each resource adapter supports.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapters.

Managed and non-managed environments

The connection, transaction and security qualities of service can either be managed by the application server or they can be provided by the Java application.

In a managed environment, a J2EE application server such as WebSphere Application Server manages the connections, transactions, and security. In this situation, the application developer does not have to provide the code for these.

In a non-managed environment, the Java application uses the resource adapters directly without the intervention of a J2EE application server. In this situation the application must contain code for the management of connections, transactions and security.

The Common Client Interface

The Common Client Interface (CCI) of the J2EE Connector Architecture provides a standard interface that allows developers to communicate with any number of Enterprise Information Systems (EISs) through their specific resource adapters, using a generic programming style.

The CCI is closely modeled on the client interface used by Java Database Connectivity (JDBC), and is similar in its idea of Connections and Interactions.

Generic CCI Classes

The generic CCI classes define the environment in which a J2EE component can send and receive data from an EIS. When you are developing a J2EE component you must follow these steps.

1. Use the ConnectionFactory object to create a Connection object.
2. Use the Connection object to create an Interaction object.
3. Use the Interaction object to run commands on the EIS.
4. Close the Interaction and Connection.

The following example shows the use of the J2EE CCI interfaces to run a command on an EIS.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
Connection conn = cf.getConnection();
Interaction interaction = conn.createInteraction();
interaction.execute(<Input output data>);
interaction.close();
conn.close();
```

CICS-specific classes

The CICS Transaction Gateway resource adapters provide additional classes specific to CICS. The following object types are used to define the ECI--specific properties:

- InteractionSpec objects
- ConnectionSpec objects

Spec objects define the action that a resource adapter carries out, for example by specifying the name of a program which is to be executed on CICS.

Record objects store the input/output data that is used during an interaction with an EIS, for example a byte array representing an ECI COMMAREA.

The following example shows a complete interaction with an EIS. In this example input and output Record objects and Spec objects are used to define the specific attributes of both the interaction and the connection. The example uses setters to define any component-specific properties on the Spec objects before they are used.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setXXX();    //Set any connection specific properties

Connection conn = cf.getConnection( cs );
Interaction interaction = conn.createInteraction();
ECIInteractionSpec is = new ECIInteractionSpec();
is.setXXX();    //Set any interaction specific properties

RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();

interaction.execute( is, in, out );
interaction.close();
conn.close();
```

The following sections cover the ECI implementations of the CCI classes in detail.

Using the ECI resource adapters

A J2EE developer can use the ECI resource adapters to access CICS programs, using COMMAREAs and channels, to pass information to and from the server.

Table 8 shows the JCA objects corresponding to the ECI terms listed in "Input and output information for external calls to CICS" on page 7. The CCI interfaces for CICS are in the com.ibm.connector2.cics package.

Table 8. ECI terms and corresponding JCA objects

ECI term	JCA object: property
Abend code	CICSTxnAbendException
COMMAREA	Record
Channel	ECIChannelRecord
Container with a data type of BIT	byte[]

Table 8. ECI terms and corresponding JCA objects (continued)

ECI term	JCA object: property
Container with a data type of CHAR	String
ECI timeout	ECIInteractionSpec:ExecuteTimeout
LUW identifier	J2EE transaction
Password	ECIConnectionSpec:Password
Program name	ECIInteractionSpec:FunctionName
Server name	ECIConnectionFactory:ServerName
SocketConnectTimeout	ECIConnection:SocketConnectTimeout
TPNName	ECIInteractionSpec:TPNName
TranName	ECIInteractionSpec:TranName
User ID	ECIConnectionSpec:UserName

The ECI resource adapters with channels and containers

To use channels and containers in the J2EE Connector Architecture (JCA), use an ECIChannelRecord to hold your data. When the ECIChannelRecord is passed to the execute() method of ECIInteraction, the method uses the ECIChannelRecord itself to create a channel and converts the entries inside the ECIChannelRecord into containers before passing them to CICS.

The ECIChannelRecord allows multiple data records to pass over the same interface to and from the execute() method of ECIInteraction. A container is created for each entry in the channel. You can have a combination of container types in one channel. The containers are of the following types:

- A container with a data type of BIT. This type of container is created when the entry is a `byte[]`, or implements the `javax.resource.cci.Streamable` interface. No code page conversion takes place.
- A container with a data type of CHAR. This type of container is created when you use a String to create the entry.

You can create your own data records, which must conform to existing JCA rules (they must implement the `javax.resource.cci.Streamable` and `javax.resource.cci.Record` interfaces). Any data records you create are treated as containers with a data type of BIT.

You can also use an existing Record type, for example, JavaStringRecord, to create a container with a data type of BIT.

The ECIChannelRecord.getRecordName method obtains the name of the channel. When creating your Record, you must make sure that the name is not an empty string. The record.getRecordName method retrieves the name of the containers.

The JCA resource adapter handles ECIChannelRecord and Records differently, when it receives the data in the execute() method of ECIInteraction.

- When an ECIChannelRecord is received, the resource adapter uses a channel to send the data.
- When a Record (that is not an ECIChannelRecord) is received, the resource adapter uses a COMMAREA to send the data.

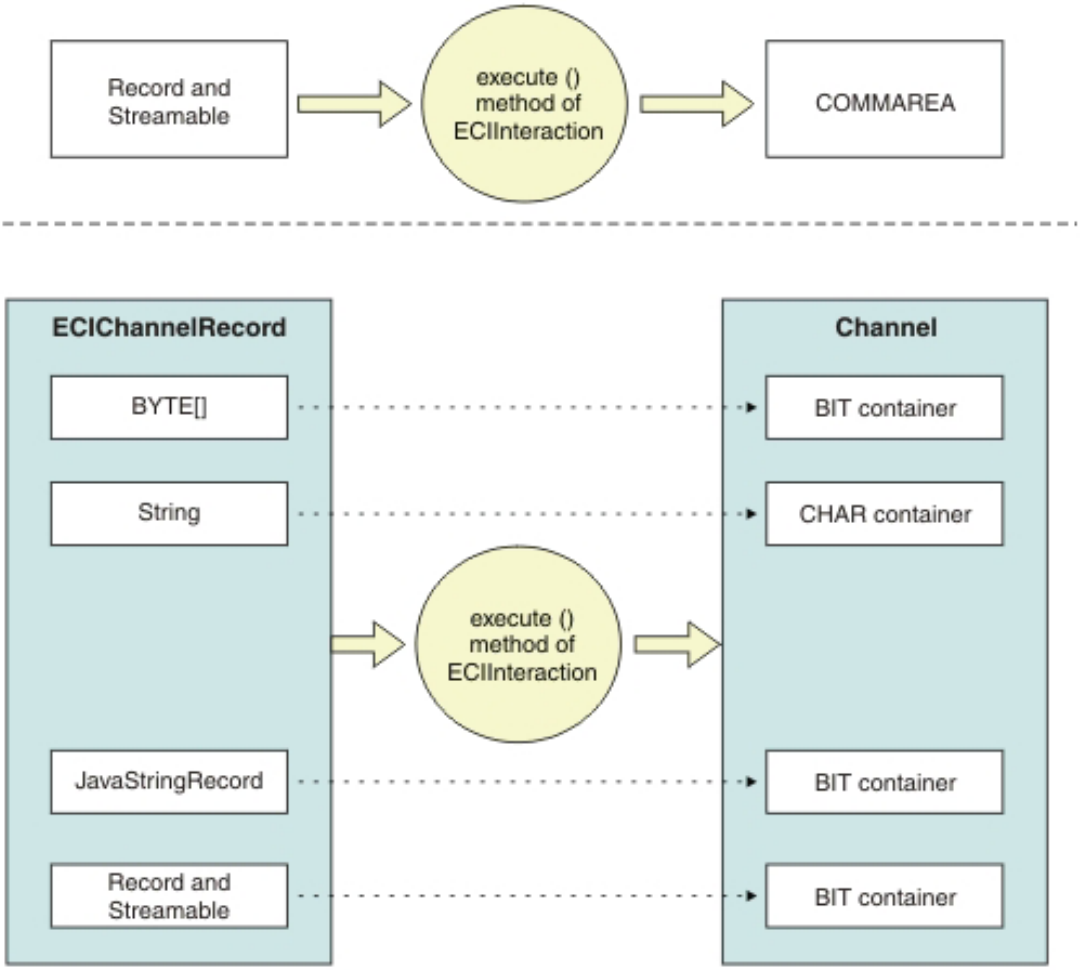


Figure 4. Data conversion by the execute() method of ECIInteraction, depending on whether it receives a Record or ECIChannelRecord

Connection to a CICS server using the ECI resource adapter

Use the ConnectionFactory and Connection interfaces to establish a connection with a CICS server. The ECI resource adapter provides implementations of the connection interfaces, but you do not work directly with the ECI implementations. Use the ECIConnectionSpec class directly to define the properties of the connection.

The ECIConnectionSpec class allows the J2EE component to override the user ID and password set at deployment time. So, to obtain a connection, you code something like this:

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setUserName("myuser");
cs.setPassword("mypass");
Connection conn = cf.getConnection(cs);
```

Linking to a program on a CICS server

Use the Interaction interface to link to a server program. The ECI resource adapter provides an implementation of the Interaction interface but you do not use this directly.

To define the properties of the interaction use the ECIInteractionSpec class directly.

1. Set the FunctionName property to the name of the CICS server program.
2. Set the InteractionVerb to SYNC_SEND for an asynchronous call or SYNC_SEND_RECEIVE for a synchronous call. Use SYNC_RECEIVE to retrieve a reply from a asynchronous call.

   Note:

   a. When a SYNC_SEND call has been issued with the execute() method of a particular ECIInteraction object, that instance of ECIInteraction cannot issue another SYNC_SEND, or SYNC_SEND_RECEIVE, until a SYNC_RECEIVE has been run.

   b. Simultaneous asynchronous calls to the same connection are permitted, provided they do not result in two asynchronous calls being outstanding in the same transaction scope. In that case an exception is thrown.

   c. If you are using the adapter in local mode with IBM WebSphere Application Server for z/OS, and you require transactional support, specify the SYNC_SEND_RECEIVE interaction type. If you use SYNC_SEND and SYNC_RECEIVE to issue asynchronous requests, the ECI requests are issued with SYNCONRETURN, and are outside the scope of the current global transaction. In remote mode, asynchronous calls work in the usual way.
3. If you are using channels and containers, the program receiving the data does not need the exact size of the data returned. If you are using COMMAREAs, set the CommareaLength property to the length of the COMMAREA being passed to CICS. If it is not supplied, a default is used:

   SYNC_SEND, SYNC_SEND_RECEIVE
   Length of input record data

SYNC_RECEIVE
The value of ReplyLength

4. Set the ReplyLength property to the length of the data stream to be returned from the Gateway daemon to the JCA application. This value can reduce the data transmitted over the network if the data returned by CICS is less than the full COMMAREA size, and you know the size of the data in advance.

   The JCA application still receives a full COMMAREA of the size specified in CommareaLength, but the amount of data sent over the network is reduced. This method is equivalent to the setCommareaInboundLength() method available for the ECIRequest class.

   If you do not set ReplyLength, CICS Transaction Gateway automatically strips trailing zeros from the COMMAREA sent from the Gateway daemon to the JCA application, without needing the size of the data in advance.

   For more information on COMMAREA stripping, see "ECI performance considerations when using COMMAREAs" on page 13.

As with ECIConnectionSpec, you can set properties on the ECIInteractionSpec class at either construction time or by using setters. Unlike ECIConnectionSpec, the ECIInteractionSpec class behaves like a Java bean. So, in a managed environment, your server might provide tools to allow you to define these properties using a GUI without writing any code.

To specify a value for ECI timeout, set the ExecuteTimeout property of the ECIInteractionSpec class to the ECI Timeout value. Allowable values are:

0 No timeout default value.

A positive integer
Time in milliseconds.

ECI timeout restrictions

When an EXCI connection to CICS is used by an ECI resource adapter either in remote mode through a Gateway daemon running on z/OS or in local mode on z/OS, ECI timeout is not supported.

Any value set by the setExecuteTimeout method of the ECIInteractionSpec class is ignored. If you are using EXCI, you can set the TIMEOUT parameter in the EXCI options table DFHXCOPT. If you are using IPIC in remote mode, you can set the CONNECTTIMEOUT parameter in the configuration file. If you are using IPIC in local mode, you can set this in the JavaGateway.setSocketConnectTimeout() method.

For more information on ECI timeouts, see the *CICS External Interfaces Guide* and "ECI timeouts" on page 43.

For performance implications of J2EE resource adapters, see the information relating to J2EE transactional considerations in *CICS Transaction Gateway: z/OS Administration*.

ECI resource adapter CICS-specific records using the streamable interface

For input and output, the ECI resource adapter supports only records that implement the `javax.resource.cci.Streamable` interface.

MappedRecords that are used to make up channels and containers also conform to the `javax.resource.cci.Streamable` interface. This interface allows the ECI resource adapter to read streams of bytes that make up the CICS COMMAREAs or channels and containers directly from, and write them to, the Record objects supplied to the execute() method of ECIInteraction.

The following example shows how to build a record for use as input by the ECI resource adapter, using the method supplied in the `javax.resource.cci.Streamable` interface.

```
Byte commarea[] = new byte[10];
ByteArrayInputStream stream = new ByteArrayInputStream(commarea);
Record in = new RecordImpl();
in.read(stream);
int.execute(..., in, ...);
```

To retrieve a byte array from the output record, use output records write() method using a ByteArrayOutputStream object as the parameter to reverse the process shown in the above example. The streams toByteArray() method then provides the CICS COMMAREA or channel and container output in the form of a byte array. In the above example a class called RecordImpl is used as the concrete implementation class of the `javax.resource.cci.Record` interface. To provide more function for your specific J2EE components, you can write implementations of the Record interface that allow you to set the contents of the record using the constructor. In this way, you avoid the use of the ByteArrayInputStream used in the above example. A managed environment might provide tools that allow you to build implementations of the Record interface that are customized for your J2EE components needs without writing any code.

Transaction management

Two resource adapters are supplied, one provides LocalTransaction support and the other both LocalTransaction and XATransaction support.

`cicseci.rar`
: This resource adapter provides LocalTransaction support when deployed on any supported J2EE application server. Local transactions are not supported when using WebSphere Application Server for z/OS with CICS

Transaction Gateway on z/OS in local mode, because the resource adapter and MVS™ RRS provide global transaction support.

cicseciXA.rar
This resource adapter provides both XATransaction and LocalTransaction support when deployed on any supported J2EE application server connecting to a remote CICS Transaction Gateway for z/OS. It also provides global transaction support when using WebSphere Application Server for z/OS with CICS Transaction Gateway on z/OS in local mode.

Use the cicseciXA.rar resource adapter for two–phase commit with IPIC. For one–phase commit you can use the cicseciXA.rar resource adapter; however, performance might be improved by using cicseci.rar resource adapter.

To provide for different transactional qualities of service for J2EE applications, you can deploy both CICS resource adapters into the same J2EE application server. When multiple resource adapters are used in the same J2EE application server, they must all be at the same version.

See the information about Deploying CICS resource adapters in the *CICS Transaction Gateway: z/OS Administration* for information about installing the resource adapters.

When carrying out multiple interactions with CICS using the ECI resource adapter, you might want to group all actions together to ensure that they either all succeed or all fail. The preferred way is to let the J2EE application server manage this; such transactions are known as *container-managed transactions*. However, to do this yourself, use the LocalTransaction or UserTransaction interface. Such transactions are known as *bean-managed transactions*. Bean-managed transactions that use the LocalTransaction interface can group work performed only through the resource adapter; the UserTransaction interface allows all transactional resources in the application to be grouped.

cicseciXA.rar with bean-managed transactions
Supports the UserTransaction and LocalTransaction interfaces.

cicseci.rar with bean-managed transactions
Supports the LocalTransaction interface.

Restrictions on WebSphere Application Server for z/OS

On WebSphere Application Server for z/OS, you cannot use the local transaction interface if you have configured the ECI resource adapter to run in local mode. In this environment, if you plan to connect to CICS using the local protocol, do not attempt to get a LocalTransaction object from the connection (that is, do not call the method getLocalTransaction() on your

connection object). In managed mode, attempts to call getLocalTransaction() result in a NotSupportedException being thrown. In non-managed mode, the results are unpredictable.

Samples

J2EE ECI sample programs are provided in the <install_path>\samples subdirectory and as a deployable EAR file in the <install_path>\deployable subdirectory.

See "Resource adapter samples" on page 67, for more information.

XA overview

A global transaction is a recoverable unit of work performed by one or more resource managers in a distributed transaction processing environment, coordinated by an external transaction manager.

The resources that are updated by the transaction can take many forms, such as a database table, a messaging queue, or the resources updated by running a CICS transaction. Each of these resources is managed by a resource manager. Where the recoverable resources updated by the global transaction are all managed by the same resource manager, a one-phase commit protocol is adequate to ensure that all resources are updated in an atomic manner.

However, where the resources updated by a global transaction are managed by multiple resource managers, a two-phase commit protocol is required. With this protocol the atomic nature of the transaction is maintained by ensuring that all resource managers update their resources in a consistent manner. The cicseciXA.rar supports the two-phase commit XA protocol and enables J2EE applications to include CICS resources in such global transactions.

In both the one-phase commit and XA scenarios, a transaction manager is responsible for controlling the running of the transaction and for coordinating the resource managers to ensure that the transaction works in an atomic manner.

An example of where this behavior is required is an online flight booking, which uses one resource manager to debit a customer's bank account and another to reserve the customer a flight. The customer's account must be updated only if the flight is booked; and vice versa.

For information on using XA transactions with J2EE applications, see *Redpaper: Transactions in J2EE, REDP-2659-00*.

WebSphere optimizations

The following optimizations are supported:

- Last participant support
- Only-agent optimization

See the documentation supplied with WebSphere Application Server for more details.

Samples

JCA ECI sample programs are provided in the samples subdirectory of your CICS Transaction Gateway installation or as a deployable EAR in the <install_path> deployable subdirectory.

These are documented in "Resource adapter samples" on page 67.

Using the resource adapters in a nonmanaged environment

You can use the resource adapters in a nonmanaged environment.

In this environment, you are responsible for:

- Defining the EIS connection
- Creating the ConnectionFactory object
- Providing your own connection pooling
- Supplying your log writer
- Managing transactions

Your nonmanaged environment can be either inside, or outside, a J2EE server environment. The resource adapters provide a default connection manager to support execution within the nonmanaged environment.

Transaction management applies only to the ECI resource adapter. See "Transaction management" on page 61 for information on managing transactions in a nonmanaged environment.

Creating the appropriate ConnectionFactory object

Your application needs to get an appropriate ConnectionFactory object.

In the managed environment, the server or application does this for you, and you can reference it using JNDI (see "Storing ConnectionFactory objects" on page 65). In the nonmanaged environment, unless you have previously registered one that you can access, you must create a ConnectionFactory object with the appropriate EIS connection information.

Creating an ECI ConnectionFactory

You must first create an ECIManagedConnectionFactory and set the appropriate properties on this object.

The properties are the same as the deployment parameters described in *Deployment parameters for the ECI resource adapters* in the *CICS Transaction Gateway: z/OS Administration*.

These are accessible using setter and getter methods. The *J2EE Programming Reference* documentation lists the setter and getter methods for the ECIManagedConnectionFactory and shows the relationship between deployment parameters and properties. The following example shows how to create a ConnectionFactory for ECI:

```
ECIManagedConnectionFactory eciMgdCf = new ECIManagedConnectionFactory();
eciMgdCf.setConnectionURL("local:");
eciMgdCf.setPortNumber("0");
eciMgdCf.setServerName("tp600");
eciMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
eciMgdCf.setUserName("myUser");
eciMgdCf.setPassword("myPass");
eciMgdCf.setTraceLevel(new
                    Integer(ECIManagedConnectionFactory.RAS_TRACE_ENTRY_EXIT));
ConnectionFactory cxf = (ConnectionFactory)eciMgdCf.createConnectionFactory();
```

Storing ConnectionFactory objects

You can store ConnectionFactory objects for later reuse, so that your application does not have to rebuild them.

Inside a J2EE server environment, IBM recommends that you register your ConnectionFactory object, which has links to your EIS connection information, in the J2EE *Java Naming and Directory Interface* (JNDI) service. This makes upgrade from nonmanaged to managed Java environments easier because applications can get ConnectionFactory objects in the same manner. However, this might not be possible outside a JNDI environment unless either an *LDAP* server, or an appropriate *JNDI Service Provider*, is available within your environment.

The resource adapter ConnectionFactory objects support both the *serializable* and *referenceable* Java interfaces. This means that you can choose how to register them in the JNDI. For more information, see the *J2EE Connector Architecture Specification*.

If you plan to use serializable interfaces, see "Issues with tracing if ConnectionFactory serialized" on page 67. This gives information on how serialization and deserialization of ConnectionFactory objects affects the setting of the LogWriter property.

Running the J2EE CICS resource adapters in a nonmanaged environment

In a J2EE environment, all of the required Java libraries are available. You might need to ensure that your J2EE server adds the following delivered jar files to your class path. These files are in the <install_path>\classes subdirectory:

- cicsj2ee.jar
- ctgclient.jar
- ctgserver.jar (required only for local: protocol)
- ccf2.jar
- connector.jar

Outside a J2EE environment, you must ensure that, in addition to the above libraries being listed in the class path, the following Java extensions are also available:

- JCA 1.5 Connector class file (required for ECI resource adapter)
- Java Transaction API (required for XA transactions)

The JCA 1.5 Connector class file and the Java Transaction API (JTA) libraries are available for download from the Oracle Java Web site.

Compiling applications

To enable Java applications to be compiled in a managed or nonmanaged environment, the relevant .jar details must be added to the class path.

To compile supplied applications in both managed and nonmanaged environments, include the following in the CLASSPATH:

- cicsj2ee.jar (required for access to Connection and Interaction Specs)
- ctgclient.jar (required for AIDkey objects)
- ccf2.jar (required for creating LogonLogoff classes)
- connector.jar (required for all resource adapter applications)

Security credentials and the CICS resource adapters

Security Credentials for accessing CICS can come from three different places.

These are the ConnectionSpec properties, the deployed security credentials, or the server itself (for nonmanaged environments, the third option does not apply). The precedence for these credentials is:

1. The Server Supplied Credentials (highest precedence)
2. The ConnectionSpec Supplied Credentials
3. The Deployed Security Credentials.

Managed enterprise applications can be deployed with "container" or "application" as a security choice. If "container" is specified, the J2EE application server will provide the credentials by means of a user interface. If "application" is specified, security is determined from the deployment properties and can be overridden by the ConnectionSpec.

J2EE tracing

In a nonmanaged environment where the DefaultConnectionManager is used the application can set the LogWriter property on the class to define where trace messages are sent.

It is important to note however that in a nonmanaged environment, if the ConnectionFactory is serialized for storage the `LogWriter` must be set after deserialization in order for it to be used, as it is not restored automatically after deserialization. This process is shown in the following example:

```
ECIManagedConnectionFactory MCF = new ECIManagedConnectionFactory();
MCF.setLogWriter(myLogWriter);

ECIConnectionFactory cf = MCF.createConnectionFactory();
objOutStream.write(cf);

ECIConnectionFactory cf2 = (ECIConnectionFactory) objInStream.read();
DefaultConnectionManager.setLogWriter(myLogWriter);
```

Issues with tracing if ConnectionFactory serialized

As described above, if you use the serializable interface to store your ConnectionFactory then you lose the reference to your LogWriter.

This is because LogWriters are not serializable and cannot be stored. When you deserialize your ConnectionFactory it will not contain a reference to the LogWriter. To ensure that your LogWriters are stored on any connections created from this ConnectionFactory you must do the following. This only applies in a nonmanaged environment.

```
DefaultConnectionManager.setLogWriter(new java.io.PrintWriter(System.err));
Connection Conn = (Connection)cxf.getConnection();
```

The `setLogWriter` method on the `DefaultConnectionManager`, which is supplied with the resource adapters, is a static method. The example above shows how to set the log to output the `System.err`. The trace level applied to the `ManagedConnectionFactory` remains.

Resource adapter samples

The resource adapter samples consist of ECI COMMAREA and channels and containers samples.

The samples show you how to use the CICS resource adapters and how to write custom records that implement the javax.resource.cci.Streamable interface. For information on how to deploy the ECI resource adapter, see *Deploying CICS resource adapters* in the *CICS Transaction Gateway: z/OS Administration*.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapters.

ECI COMMAREA sample

The ECI COMMAREA sample consists of a stateless session bean, a client application, and a custom record that demonstrates using the Streamable interface.

The following files are part of the sample:

ECIDateTime.java
Enterprise bean remote interface

ECIDateTimeHome.java
Enterprise bean home interface

ECIDateTimeBean.java
Enterprise bean implementation

ECIDateTimeClient.java
Enterprise bean client program

JavaStringRecord.java
Custom record

Ejb-jar-eci-1.1.xml
Example of a deployment descriptor

The deployment descriptor is an example of an EJB 1.1–compliant deployment descriptor for this enterprise bean. If you wish to package it up into a jar file, rename it to `Ejb-jar.xml` and store it in the META-INF directory of the jar file. It might require further entries if it is to be deployed into an EJB 2.0–compliant environment.

See your J2EE Server documentation for information on how to compile and deploy the bean within your environment. However, you need to ensure that the following jar files are also available on the CLASSPATH:

- cicsj2ee.jar
- connector.jar
- ctgclient.jar
- ccf2.jar

The enterprise bean looks for an ECI connection factory named `java:comp/env/ECI`. The bean must refer to this resource when deployed. See your J2EE Server documentation on how to deploy the resource adapter with an entry in the JNDI with this name. The client program looks for the ECIDateTime bean with a name of `ECIDateTimeBean1`. See your J2EE Server documentation for details of how to setup the bean with this JNDI name.

You will need to install the server sample program EC01 on your CICS Server. This file can be found in the samples\server subdirectory of your CICS Transaction Gateway installation. Further details of this sample can be found in Chapter 10, “Sample programs,” on page 99.

The bean is a simple bean that outputs the date and time as known to the CICS Server, and can be deployed as a bean-managed transaction. The Custom record takes a COMMAREA and converts it to a string. Ensure that the EC01 sample program, which you installed on your CICS server, sends its results in ASCII, as the COMMAREA is expected in ASCII. The JavaStringRecord does however allow for the selection of other encodings, and is commented using JavaDoc. The Client program takes no parameters. If your CICS server is running on z/OS, the EC01 sample program will return its results in EBCDIC rather than ASCII. To resolve this, update the DFHCNV table by adding lines similar to the following:

```
*
* CTG Sample conversion
*
*
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=EC01,USREXIT=NO,             *
               SRVERCP=037,CLINTCP=8859-1
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=18,      *
               LAST=YES
```

ECI channels and containers sample

The ECI channels and containers sample uses JCA to send an ECI request to a sample channel program in CICS called EC03. The CICS EC03 sample program adds containers to the channel which is then returned.

The sample can call the CICS sample program EC03, either through the ECI resource adapter, or through the ECI XA resource adapter. The sample includes a client application that invokes an enterprise bean. The enterprise bean then issues the ECI request to CICS.

The sample includes the following files:

EC03ChannelBean.java
: The implementation of the EC03 Channel EJB

EC03Channel.java
The remote interface for the EC03 Channel EJB

EC03ChannelHome.java
The home interface for the EC03 Channel EJB

EC03ChannelClient.java
A basic client which calls the EC03 Channel EJB

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements, in this case, the execute() method. The Home interface handles the life cycle of the enterprise bean.

EC03ChannelClient looks up the enterprise bean as EC03ChannelHome in the JNDI (Java Naming Directory Interface). It then locates an object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. The remote method then looks up the resource adapter connection factory (an instance of the resource adapter) under the name ECI. The method runs EC03 in CICS, passing in a channel with one container. When the ECI call program returns, the containers returned from the program are enumerated and placed into a HashMap which is then returned back to the client application that issued the call.

To use the sample:

1. Deploy the CICS ECI resource adapter (cicseci.rar) or the CICS ECI XA resource adapter (cicseciXA.rar); these are located in the deployable directory of the CICS Transaction Gateway install path.
2. Create a connection factory with parameters to suit your CICS server environment. .

   Note: The connection factory must have a JNDI name of ECI for the sample to work.
3. Deploy your enterprise bean. This automatically generates code that handles remote method calls to your enterprise bean that are made by the enterprise bean client. This process is specific to your J2EE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you will be asked for might include:

   Transaction Type
   This can be set to container-managed, or bean-managed, and determines whether you want to control transactions yourself. The J2EE application server manages Container managed transactions. If you are prompted, select Container managed for the sample.

Enterprise bean Type
EC03Channel is a stateless session bean.

JNDI Name
The enterprise bean client uses JNDI to look up the name of the enterprise bean in the naming directory.

Resource References
The enterprise bean refers to a connection factory. You must add the connection factory (as defined in step 2) as a resource reference for this enterprise bean.

4. Run the client application. You can run the client either from the command line or with the launchClient utility (if you are using WebSphere Application Server). The launchClient utility sets up the necessary parameters to communicate with the JNDI directory in WebSphere to find the EC03Channel enterprise bean. The application calls the bean, passes a text string to the EC03 program, and displays the contents of the container that the EC03 program returns.

Assistance in coding CCI applications

When coding CCI applications, refer to the Javadoc and the specification for the J2EE Connector Architecture (JCA).

Connector specification API Javadoc

You can obtain the connector architecture API Javadoc from the Sun Web site, this will assist in the coding of your CCI applications and provides information such as the exceptions used by CCI implementations.

J2EE Connector Architecture (JCA) API

Refer to the JCA specification when coding CCI applications.

IBM recommends that you get the *J2EE Connector Specification* document from Oracle's Web site at Java EE Downloads, to help in coding your CCI applications. It contains information such as the exceptions used in CCI applications.

Chapter 6. Programming in C

This information describes the external access interfaces specific to C. It does not deal with testing or debugging ECI applications; refer instead to the programming documentation for the environment in which you are working.

Related information

"Supported programming languages" on page 3
This table shows which programming languages are supported by the CICS Transaction Gateway for each platform and application programming interface (API).

Overview of the programming interfaces for C

Parameter blocks are used to pass data between the Client application and the ECI.

A user application must be constructed as a single process, though in environments in which a process can generate several threads, the user application can be multi-threaded.

For remote mode, a C interface is provided for the ECI. See "Making ECI calls from C programs in remote mode." Local mode C clients are not supported.

Making ECI calls from C programs in remote mode

This section describes how to make ECI calls to a CICS server from a C Client application in remote mode.

Use the CTG_ECI_PARMS parameter block structure to communicate with a CICS server. The parameter block fields are used for input and output. To communicate with the CICS server using the Gateway daemon use the CTG_ECI_Execute function. The Remote Client interface requires Version 2 of the ECI Parameter block. The reserved field is ignored by the Remote Client interface. Set the ECI parameter block to nulls before setting the input parameter fields. For guidance on how to use the ECI to manage logical units See "Managing logical units of work" on page 78.

The following table shows the field names in C data structures that correspond to the ECI terms described in "Input and output information for external calls to CICS" on page 7.

Table 9. ECI terms and corresponding fields in C in remote mode

ECI term	C structure.field
Abend code	CTG_ECI_PARMS.eci_abend_Code
COMMAREA	CTG_ECI_PARMS.eci_commarea
ECI timeout	CTG_ECI_PARMS.eci_timeout
LUW control	CTG_ECI_PARMS.eci_extend_mode
LUW identifier	CTG_ECI_PARMS.eci_luw_token
Password	CTG_ECI_PARMS.eci_password
Program name	CTG_ECI_PARMS.eci_program_name
Server name	CTG_ECI_PARMS.eci_system_name
TPNName	CTG_ECI_PARMS.eci_tpn
TranName	CTG_ECI_PARMS.eci_transid
User ID	CTG_ECI_PARMS.eci_userid

Considerations when using multithreaded ECIv2 applications

There are a number of things to consider when using multithreaded ECIv2 applications to connect to CICS.

- The CICS Transaction Gateway ECIv2 API supports multithreaded applications communicating using either CICS COMMAREA or channels and containers. Multithreaded applications that access CICS COMMAREAs do not use thread locking; each thread accesses a single COMMAREA. The application programmer should manage the locking between threads, however, with channels and containers the API locks the channel to prevent data being modified while a flow is in progress.
- All communication operations must be performed on the thread that opened the connection to the Gateway. For concurrent ECI requests, if the application has multiple connections, each connection must have its own dedicated thread.
- Channels can be created, flowed and deleted from any thread.
- Containers can be created, read from, written to and deleted from any thread.
- While an ECI request is flowing on a channel, the channel is locked and no other operations are possible on that channel or container. Operations that are attempted on a locked channel are queued until the flow completes at which point, the API unlocks the channel.
- Asynchronous operation using multithreaded applications can be achieved if the application uses multiple communications threads.

- One connection manager thread in the Gateway is allocated for each successful open Gateway connection request. The Idle time Gateway parameter can be used to prevent the application from holding idle connections open for too long.

Establish a remote Client connection to a Gateway daemon

To use client applications in C in remote mode, you must establish a connection to the Gateway daemon's Client protocol handler using the specified host name and port number.

The following function establishes a remote Client connection to a Gateway daemon:

```
int CTG_openRemoteGatewayConnection(char *                  address,
                                    int                     port,
                                    CTG_ConnToken_t*        gwTokPtr,
                                    int                     connTimeout
                                   )
```

The **CTG_openRemoteGatewayConnection** function takes the character pointer for the host name, an integer for the target port number, a pointer to gateway token and the address of a character pointer to be modified with the returned protocol version. The connection to a Gateway daemon is established by Client protocol handler using the specified host name and port number. If the connection is successful the gateway token will represent the connection to the specified Gateway daemon on return, and is required to interact with that Gateway daemon on further API calls.

Table 10. Parameters

[in]	*address*	The character string containing the host name or IP address of the target Gateway daemon.
[in]	*port*	An int for the port of the target Gateway daemon.
[out]	*gwTokPtr*	Pointer to an int for the new Gateway Token.
[in]	*connTimeout*	The connection timeout in seconds, enter 0 for no timeout.

The return value indicates whether or not the call was successful. Possible return values are:

Table 11. Return values

CTG_OK
CTG_ERR_LOCKFAIL
CTG_ERR_MALLOCFAIL
CTG_ERR_BADPORT
CTG_ERR_CONNECTFAILED

Table 11. Return values (continued)

CTG_ERR_BADGWTOKLIST
CTG_ERR_NULLGWTOKPTR
CTG_ERR_NULLPARM
CTG_ERR_LOSTGWCON
CTG_ERR_BADHOST
CTG_ERR_NULLADDRESS

Close a remote Client connection to a Gateway daemon

The following functions closes a remote Client connection to a Gateway daemon:

```
CTG_closeGatewayConnection(CTG_GatewayToken_t * gwTokPtr)

CTG_closeAllGatewayConnections( )
```

The **CTG_closeGatewayConnection** function attempts to free all resources held by the API, including open Gateway daemon connections. This function can be used by the API program under normal shutdown circumstances, or in the event of a severe error, and it enables some form of controlled shutdown even if references to gateway tokens have been lost.

Table 12. Parameters

[out]	*gwTokPtr*	The reference to the open Gateway connection

The return value indicates whether or not the call was successful. Possible return values are:

Table 13. Return values

CTG_OK
CTG_ERR_LOCKFAIL
CTG_ERR_MALLOCFAIL
CTG_ERR_NULLGWTOK
CTG_ERR_BADGWTOK
CTG_ERR_PIDMISMATCH
CTG_ERR_TIDMISMATCH
CTG_ERR_NULLGWTOKPTR

The **CTG_closeAllGatewayConnections** function attempts to free all resources held by the API, including open Gateway daemon connections. This function is for use in the event of a severe error because it enables some form of controlled shutdown even if all gateway tokens (gwTokens) have been lost.

The return value indicates whether or not the call was successful. Possible return values are:

Table 14. Return values

CTG_OK
CTG_ERR_LOCKFAIL
CTG_ERR_MALLOCFAIL
CTG_ERR_BADGWTOK

Program link calls

Fill in the required fields in the ECI parameter block. Pass any data required by the program you are linking to in the COMMAREA.

Use eci_call_type to define an ECI request as synchronous:

- **ECI_SYNC** for a synchronous program link call

For more information about the field usage see, CTG_ECI_PARMS Struct Reference

CTG_ECI_Execute

Use **CTG_ECI_Execute** for making program link calls.

Use the ECI parameter block, CTG_ECI_PARMS, for passing parameters to the ECI. The eci_call_type parameter in the ECI parameter block indicates the type of **CTG_ECI_Execute**. Only synchronous calls are supported, eci_call_type must be ECI_SYNC. The following example shows the format of the request and associated declarations:

For C programs:

```
CTG_ECI_PARMS      EciBlock;
int  Response;
.
.
.
Response = CTG_ECI_Execute(&EciBlock);
```

Managing logical units of work

To start a logical unit of work, set the **eci_extend_mode** parameter to ECI_EXTENDED and the **eci_luw_token** parameter to zero, when making a program link call.

When a transaction is started, an LUW identifier is generated and is returned in the **eci_luw_token** field. This identifier must be input to all subsequent calls for the same unit of work. To call the last program in an LUW, set the **eci_extend_mode** parameter to ECI_NO_EXTEND. To end an LUW without linking to a program, set the **eci_extend_mode** parameter to ECI_COMMIT or ECI_BACKOUT to commit or back out changes to recoverable resources.

The following table shows how you can use combinations of **eci_extend_mode**, **eci_program_name**, and **eci_luw_token** parameter values to perform tasks associated with managing logical units of work through ECI. In each case you must also store appropriate values in other fields for the call type you have chosen.

Table 15. Logical units of work in ECI

Task to perform	Parameters to use
Call a program that is to be the only program of a logical unit of work. One request flows from client to server and a reply is sent to the client only after all the changes made by the specified program have been committed.	Set up the parameters as follows: • **eci_extend_mode**: ECI_NO_EXTEND • **eci_program_name**: provide it • **eci_luw_token**: zero
Call a program that is to start an extended logical unit of work.	Set up the parameters as follows: • **eci_extend_mode**: ECI_EXTENDED • **eci_program_name**: provide it • **eci_luw_token**: zero Then save the token from **eci_luw_token**.
Call a program that is to continue an existing logical unit of work.	Set up the parameters as follows: • **eci_extend_mode**: ECI_EXTENDED • **eci_program_name**: provide it • **eci_luw_token**: provide it
Call a program that is to be the last program of an existing logical unit of work, and commit the changes.	Set up the parameters as follows: • **eci_extend_mode**: ECI_NO_EXTEND • **eci_program_name**: provide it • **eci_luw_token**: provide it

Table 15. Logical units of work in ECI (continued)

Task to perform	Parameters to use
End an existing logical unit of work, without calling another program, and commit changes to recoverable resources.	Set up the parameters as follows: • **eci_extend_mode**: ECI_COMMIT • **eci_program_name**: null • **eci_luw_token**: provide it
End an existing logical unit of work, without calling another program, and back out changes to recoverable resources.	Set up the parameters as follows: • **eci_extend_mode**: ECI_BACKOUT • **eci_program_name**: null • **eci_luw_token**: provide it

If an error occurs in one of the calls of an extended logical unit of work, you can use the **eci_luw_token** field to see if the changes made so far have been backed out, or are still pending. See the description of the **eci_luw_token** field in *CICS Transaction Gateway for z/OS: Programming Reference* for more information. If the changes are still pending, end the logical unit of work with another program link call, either committing or backing out the changes.

ECI timeouts

Use the **eci_timeout** field in the ECI parameter block to specify the timeout value. If a timeout occurs either the ECI_ERR_RESPONSE_TIMEOUT code or the ECI_ERR_REQUEST_TIMEOUT code is returned.

See "Timeout of the ECI request" on page 11 for more information on ECI timeouts.

Creating channels and containers for ECI calls for C

You can use channels and containers when you connect to CICS using the IPIC protocol. You must create a channel before it can be used in an ECI request.

1. Add the following code to your application program, to create a channel:

```
ECI_ChannelToken_t chanToken;
createChannel(&chanToken);
```

2. You can add containers with a data type of BIT or CHAR to your channel. Here is a sample BIT container:

```
char custNumber[] = {0,1,2,3,4,5};
rc = ECI_createContainer(chanToken, "CUSTNO", ECI_BIT, 0, custNumber,
sizeof(custNumber));
```

Here is a sample CHAR container that uses the channel's CCSID:

```
char * company = "IBM";
rc = ECI_createContainer(chanToken, "COMPANY", ECI_CHAR, 0, company,
strlen(company));
```

3. The channel can now be used in an ECI request, as the example shows:

```
CTG_ECI_PARMS eciParms = {0};

eciParms.eci_version = ECI_VERSION_2;
eciParms.eci_call_type = ECI_SYNC;
strncpy(eciParms.eci_system_name, "CICSA", ECI_SYSTEM_NAME_LENGTH);
strncpy(eciParms.eci_userid, "USERNAME", ECI_USERID_LENGTH);
strncpy(eciParms.eci_password, "PASSWORD", ECI_PASSWORD_LENGTH);
strncpy(eciParms.eci_program_name, "CHANPROG", ECI_PROGRAM_NAME_LENGTH);
eciParms.eci_extend_mode = ECI_NO_EXTEND;
eciParms.channel = chanToken;
```

4. When the request is complete, you can retrieve the current state of the containers in the channel, as the example shows:

```
ECI_CONTAINER_INFO contInfo;

rc = ECI_getFirstContainer(chanToken, &contInfo);

while (rc == ECI_NO_ERROR) {
    printf("Container %s\n", contInfo.name);

    if (contInfo.type == ECI_BIT) {
         printf("Type BIT\n");
    } else {
         printf("Type CHAR\n");
    }

    /* Read block of data into buffer */
    ECI_getContainerData(channelToken, contInfo.name, dataBuff,
       sizeof(dataBuff), offset, &bytesRead);

    rc = ECI_getNextContainer(chanToken, &contInfo);
}
```

Tracing in ECI Version 2 applications

Applications should implement an option to enable trace. You can control tracing in ECI Version 2 applications using the functions and environment variables described here.

You can set trace level, file, data length and offset either by using a function call or by setting an environment variable. Examples of each are shown below. To avoid having to recompile applications, enable trace by setting the environment variable.

Trace level

You can set 5 trace levels:

CTG_TRACE_LEVEL0
Disables all tracing. This is the default setting.

CTG_TRACE_LEVEL1
Enables exception trace points.

CTG_TRACE_LEVEL2
Enables event trace points and those from lower trace levels.

CTG_TRACE_LEVEL3
Enables function entry and exit trace points and those from lower trace levels.

CTG_TRACE_LEVEL4
Enables debug trace points and those from lower trace levels.

Here is an example of the trace level function call:

```
CTG_setAPITraceLevel(CTG_TRACE_LEVEL1);
```

Here is an example of the trace level environment variable:

```
CTG_CLIENT_TRACE_LEVEL=1
```

Trace file

The default trace destination is the standard error stream.

Here is an example of the trace file function call:

```
CTG_setAPITraceFile("filename.trc");
```

Here is an example of the trace file environment variable:

```
CTG_CLIENT_TRACE_FILE=filename.trc
```

Trace data length

The trace data length specifies the maximum amount of data that is written to trace when communicating with CICS Transaction Gateway and the trace level is set to CTG_TRACE_LEVEL4. The default setting is 128 bytes.

Here is an example of the trace data length function call:

```
CTG_setAPITraceDataLength(256);
```

Here is an example of the trace data length environment variable:

```
CTG_CLIENT_DATA_LENGTH=256
```

Trace data offset

The trace data offset specifies an offset into data where tracing begins. When combined with the trace data length this allows a specific section of data to be traced, for example a section of data in a COMMAREA. The default setting is zero.

Here is an example of the trace data offset function call:

```
CTG_setAPITraceDataOffset(40);
```

Here is an example of the trace data offset environment variable:

```
CTG_CLIENT_DATA_OFFSET=40
```

Security in the ECI

The Client application can specify the user ID and password by setting eci_userid and eci_password in the ECI parameter block. The user ID and password can be up to 10 characters in length.

Compiling and linking C applications

This section gives some examples showing how to compile and link typical ECI applications in the various client environments.

Refer to the sample programs supplied with your environment (see Chapter 10, “Sample programs,” on page 99) for more information about compiling and linking programs.

The following table shows the header files for C required for your programs:

Table 16. C header files

Use	File
ECI Version 2	ctgclient_eci.h and ctgclient.h

The files contain the entry points, type definitions, data structures, and constants needed for writing programs using the ECI interface.

When compiling C programs, you might need to pass structures to the external CICS interfaces in packed format. If this is the case, the C header files contain the `#pragma pack` directive, which must not be changed.

Chapter 7. Programming using the .NET Framework

The .NET Framework offers a number of advantages when developing remote client applications.

- A consistent model, provided by the .NET class library, for all supported programming languages.
- High levels of security for applications used in remote mode topologies; method-level security using industry standard security technologies can be explicitly defined.
- Separation of application logic from presentation logic for easier maintenance and upgrade.
- Simplified debugging plus the availability of runtime diagnostics.
- Simpler application deployment.

Overview of the programming interface

.NET classes are supported on all Windows platforms.

- The GatewayConnection class represents a connection to CICS Transaction Gateway. The connection is opened in the constructor and remains open until the Close() method is invoked. The class provides two methods for interacting with CICS Transaction Gateway: Flow(request) which flows an EciRequest to CICS Transaction Gateway, and ListSystems() which returns a list of all CICS servers that have been defined in CICS Transaction Gateway.
- The GatewayRequest class provides a base class for other request types, as with the Java API.
- The EciRequest represents an ECI call to CICS and provides ECI fields such as extend mode, LUW token, and COMMAREA. Channels and containers are not supported.
- The Trace class provide static methods for interacting with the tracing features of the C API.

The .NET Framework includes a RequestMinimum security attribute to execute unmanaged code (UnmanagedCode=true).

Tracing can be dynamically enabled and disabled in .NET applications, if the application has been compiled with a /d:TRACE command line switch. If compiled with this switch, tracing can then be enabled or disabled by using an application configuration file.

Making ECI calls from .NET clients

How the .NET properties map to the component parts of an ECI request.

Use the IBM.CTG.EciRequest class to pass details of an ECI request to CICS Transaction Gateway. The following table shows the .NET class properties that correspond to the ECI terms described in "Input and output information for external calls to CICS" on page 7. For more information see, the GatewayConnection information in the .NET section of the *Programming Reference*.

ECI term	.NET class property
Abend code	EciRequest.AbendCode
Channel	Not currently available in the .NET Framework
COMMAREA	EciRequest.SetCommareaData EciRequest.GetCommareaData EciRequest.CommareaLength
ECI return code	EciRequest.EciReturnCode
LUW control	EciRequest.ExtendMode
LUW identifier	EciRequest.LuwToken
TPNName	EciRequest.MirrorTransId
Password	EciRequest.Password
Program name	EciRequest.Program
Server name	EciRequest.ServerName
ECI timeout	EciRequest.Timeout
TranName	EciRequest.TransId
Userid	EciRequest.UserId

Developing .NET applications

This section describes how to develop ECI applications using the .NET Framework.

Developing using Microsoft Visual Studio

If you are developing using Microsoft Visual Studio, you must add a reference to the IBM.CTG.Client.dll assembly. The assembly has a dependency on ctgclient.dll which must be available on the system path. If CICS TG is installed on the development system, this library is placed on the path during

installation. If you are developing on a remote system using the ctgredist package, you must add the library location to the system path manually.

When you have added the reference, the types in the IBM.CTG namespace can be used to perform ECI calls to CICS. To avoid the need to fully qualify each type, you can add the IBM.CTG namespace to the imports section of your code.

See Microsoft Visual Studio documentation for further information on creating and building projects.

Compiling and linking from the command line

The .NET Framework provides command line tools for compiling and linking .NET applications. Applications that are written in C# can be compiled and linked using the csc tool:

```
csc /target:exe /out:"AppName.exe" /reference:"IBM.CTG.Client.dll"
"SourceFile.cs"
```

Applications that are written in Visual Basic.NET can be compiled and linked using the vbc tool:

```
vbc /target:exe /out:"AppName.exe" /reference:"IBM.CTG.Client.dll"
"SourceFile.vb"
```

For more information on the csc and vbc command line tools see the Microsoft documentation.

Developing or deploying on 64 bit operating systems

If you develop the application on a 64 bit operating system or the application will be deployed on a 64 bit operating system, the application must be modified to target the x86 processor architecture. See "64 bit considerations using the .NET Framework" on page 87 for more information.

Problem determination for .NET client programs

Use tracing to help determine the cause of any problems when running .NET client programs.

Tracing for .NET client programs

There are three possible ways of performing tracing in the ctgclient.dll and the IBM.CTG.Client.dll: with an application configuration file, with the Trace class or by using environment variables.

Trace level

The following trace levels are supported for all three ways of performing tracing:

CtgTrcDisabled
disable tracing

CtgTrcLevel1
includes exception trace points but nothing else

CtgTrcLevel2
includes event trace points and all CtgTrcLevel1 trace points

CtgTrcLevel3
includes function entry and exit trace points and all CtgTrcLevel1 and CtgTrcLevel2 trace points

CtgTrcLevel4
includes debug trace points and all CtgTrcLevel1, CtgTrcLevel2 and CtgTrcLevel3 trace points (the most verbose tracing level)

Enabling trace with an application configuration file

Trace can be enabled using the CtgTrace trace switch in an application configuration file (an XML file). The switch allows the trace level to be specified as a TraceLevel value, a TraceLevel value, or an integer between 0 and 4 inclusive. The switch also supports an optional fileName attribute, which can be set to the name of a file to write ctgclient.dll trace information to. The following example shows how to enable trace at CtgTrcLevel4, using the file traceFile.trc as the ctgclient.dll trace destination:

```
<?xml version="1.0" encoding="utf-8" ?>
 <configuration>
  <system.diagnostics>
   <switches>
    <add name="CtgTrace" value="CtgTrcLevel4" fileName="traceFile.trc"/>
   </switches>
  </system.diagnostics>
 </configuration>
```

where `value="CtgTrcLevel4"` specifies that Level 4 tracing is performed.

Enabling trace with the Trace class

The Trace class provides the following members for trace:

- TraceLevel gets or sets the trace level.
- SetTraceFile(String) sets the destination file to use for ctgclient.dll trace.

For more information see, the Trace information in the .NET section of the *Programming Reference*.

Enabling trace with environment variables

The trace level is controlled by setting the environment variable *CTG_CLIENT_TRACE_LEVEL* to an integer value between 0 and 4 inclusive. The name of the file to write ctgclient.dll trace information to can be set using the environment variable *CTG_CLIENT_TRACE_FILE*.

Threading restrictions

GatewayConnection is the .NET class which represents a TCP/IP connection to a CICS Transaction Gateway.

A GatewayConnection instance can only be used by the thread that creates it. Attempts to use the GatewayConnection from other threads causes an InvalidOperationException to be thrown. This includes flowing requests, listing the servers defined in the Gateway daemon, and closing the connection.

For further information, see the documentation in the .NET API section of the Programming Reference .

64 bit considerations using the .NET Framework

The CICS Transaction Gateway API for .NET does not support running in 64 bit mode. By default, the Common Language Runtime launches .NET applications as 64 bit processes when running on 64 bit operating systems.

To ensure that an application always runs in 32 bit mode, the application must be configured to target the x86 architecture. This can be achieved in the following ways:

- When you compile an application using Microsoft Visual Studio, set the **Platform target** project property to **x86**.
- When you compile an application from a command prompt or script, use the corflags utility which is provided as part of the Microsoft .NET Framework SDK. To modify an application to target the x86 architecture, use the following command:

  ```
  corflags.exe <executable_name> /32BIT+
  ```

Targeting a .NET Framework version

The CICS Transaction Gateway API for .NET is supported on .NET Framework versions 3.5 and 4.0.

Two versions of the IBM.CTG.Client.dll are provided, one for use in applications targeting .NET Framework Version 3.5 and the other for use in applications targeting .NET Framework Version 4.0 or higher. The libraries have the following locations:

```
<install_path>/lib/dotnet35
<install_path>/lib/dotnet40
```

Applications that target .NET Framework Version 3.5 are also supported when executing on .NET Framework Version 4.0, but might experience the following exception:

```
System.IO.FileLoadException: Mixed mode assembly is built against version
'v2.0.50727' of the runtime and cannot be loaded in the 4.0 runtime without
additional configuration information.
```

To enable an application that targets .NET Framework Version 3.5 to execute on .NET Framework Version 4.0, you must add the following configuration settings to the configuration section of the application configuration file:

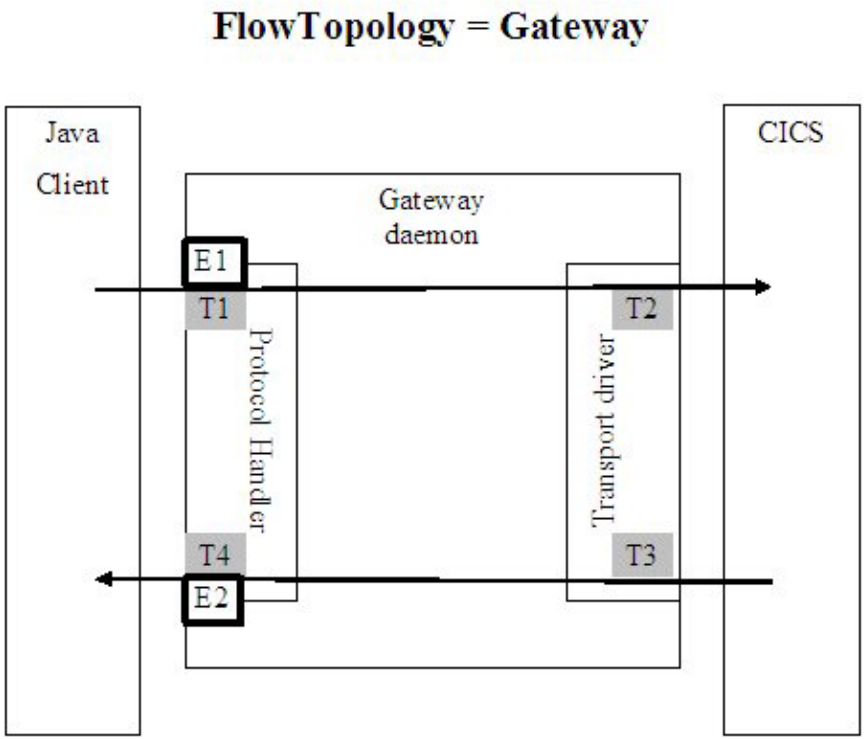
```
<startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0"/>
</startup>
```

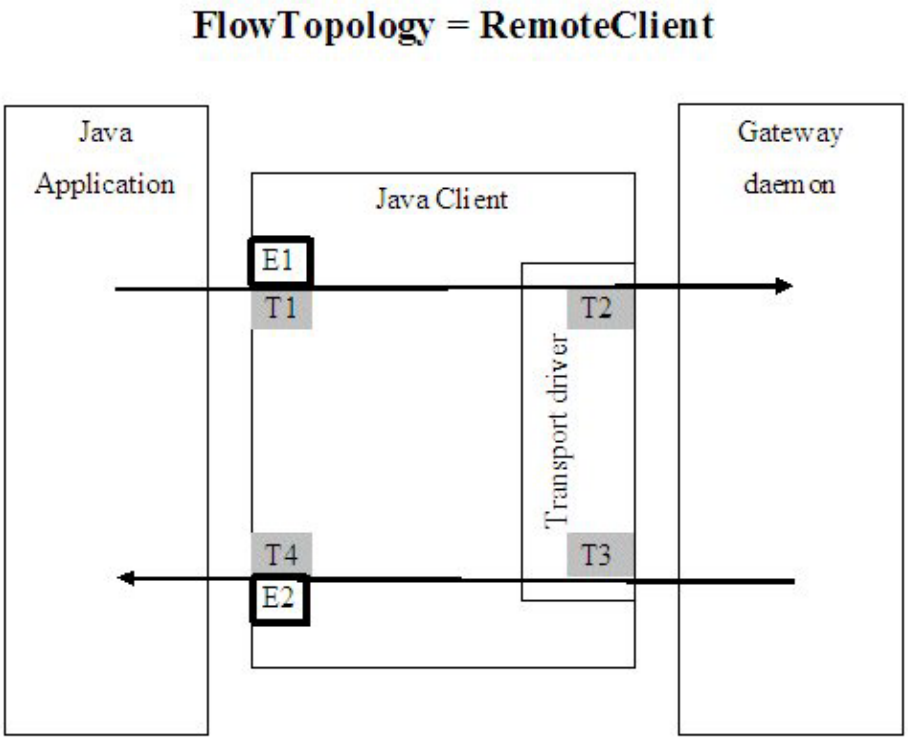
For further information on the `useLegacyV2RuntimeActivationPolicy` attribute, refer to the Microsoft documentation.

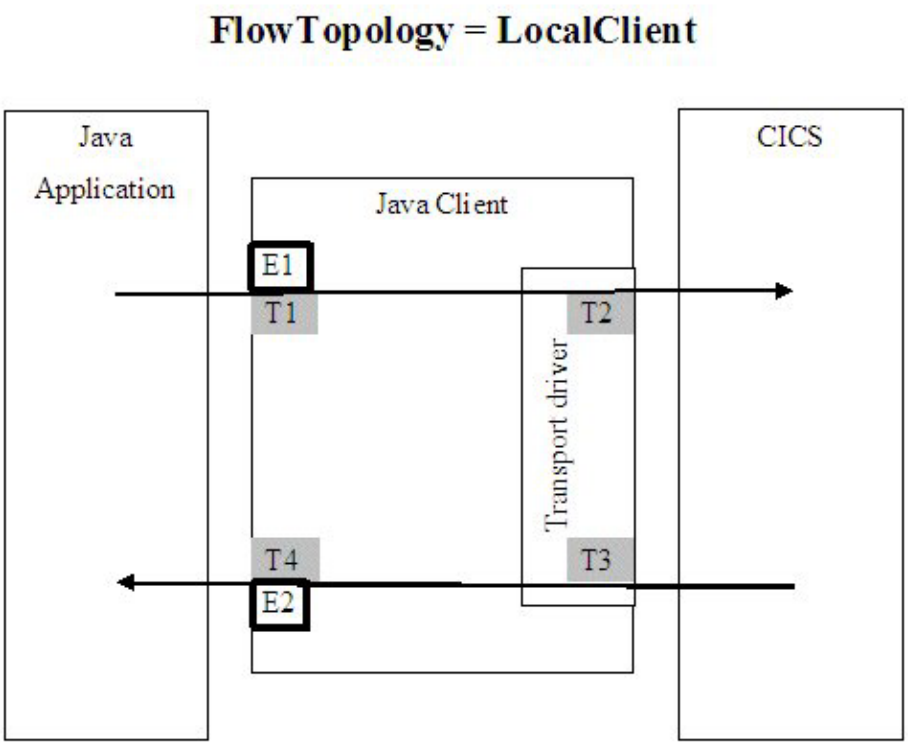
Chapter 8. Java request monitoring exits

With request-level monitoring, applications written by independent software vendors can be called at significant points in the request flow through the Gateway daemon and Gateway classes.

The following diagrams show where the request monitoring user exits are driven depending on the CICS Transaction Gateway configuration. In each diagram, points E1 and E2 show where the exits are driven, and points T1, T2, T3, and T4 show where time stamps are collected for each request.




FlowTopology = RemoteClient
Java
Application
Java Client
Gateway
daemon
E1
T1
T2
Transport driver
T4
T3
E2



The following set of rules applies when writing request monitoring user exits:

- You can configure exits to run on the Gateway classes and on the Gateway daemon independently.
- You can configure multiple exits to be active at the same time. Multiple exits are not called in a defined order.
- Configured exits are loaded at startup and remain active for the life of the JavaGateway object or Gateway daemon.
- Exits run in line, so you must code them to have minimum impact on performance.
- Exits that throw any exceptions or runtime errors are disabled.
- Exits are called for each ECI flow at request entry and response exit.
- Exits can be passed operator commands through the RMExit admin command as a RequestEvent of “Command” with a map containing RequestData key "CommandData"; see the information later in this topic.
- Exits are called at shutdown to allow them to release resources and to end cleanly.

- At call time, exits note which exit point is driving the request. Data is provided to the exit to aid monitoring of the requests.
- All implementations of CICS Transaction Gateway RequestExit monitoring classes must implement the RequestExit interface.
- You can use the default constructor to set up any external resources required by the exits. For example, the sample class com.ibm.ctg.samples.requestexit.ThreadedMonitor creates a background thread to reduce the overhead for each monitored request.

Writing a monitoring application to use the exits

A RequestExit object is defined by a class that implements the RequestExit interface. At run time, a single RequestExit object is created for each configured request-level monitor. Each object receives eventFired() method calls at the start of the request (E1) and at the end of the reply (E2) for each flow. These calls are shown by E1 and E2 on the diagrams. Timestamps are taken during the flow at T1, T2, T3, and T4 on the diagrams.

- Timestamp T1 (RequestReceived) is generated as a request arrives at the Gateway daemon or Gateway classes. This data is available when the request event type is RequestEntry or ResponseExit.
- Timestamp T2 (RequestSent) is generated as the request leaves the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.
- Timestamp T3 (ResponseReceived) is generated when the reply arrives back in the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.
- Timestamp T4 (ResponseSent) is generated when the reply leaves the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.

The RequestExit object exists for the lifetime of the Gateway daemon or Gateway classes, or until it throws an exception or runtime error. When the exit is triggered, the eventFired() method is called and runs on the same thread as the caller. When the eventFired() method returns, the thread continues running as before. Processing performed by the exit on this thread affects performance and must be kept to a minimum. An example exit (com.ibm.ctg.samples.requestexit.ThreadedMonitor.java) shows you how to transfer this processing to a separate thread to reduce the impact on performance.

Controlling request monitoring user exits dynamically

You can use the /F <jobname> command to send system management commands to your request monitoring user exits so you can interact with the request monitoring user exits, to perform tasks such as dynamically starting or stopping a particular user exit.

When you issue a system management command with a RequestEvent of “Command”, the eventFired() method is driven for all request monitoring user exits that are active on the Gateway daemon. The input data is formed of a single entry in the map, with RequestData key "CommandData". The value associated with this key is a string representing the data provided via the system management command.

Sample request monitoring user exits

A simple request monitoring user exit implementation of the RequestExit interface is in the com.ibm.ctg.samples.requestexit.BasicMonitor class. The source code for request monitoring user exits samples is located in `\samples\java\com\ibm\ctg\samples\requestexit`.

Related information

Request monitoring user exit API information

Chapter 9. Creating a CICS request exit

The CICS request exit is called by CICS Transaction Gateway in remote mode, to select a CICS server name for an ECI request. The CICS request exit can be used for request retry, dynamic server selection and for rejecting non-valid requests. If the server name returned by a CICS request exit is null, the request is sent to the default CICS server if one is specified in the configuration file (ctg.ini).

Before you begin

If a request fails with a retryable error and the retry limit has not been reached, the Gateway daemon calls the CICS request exit to select an alternative CICS server. Retryable errors are:

- The specified CICS server is no longer available (ECI_ERR_CICS_DIED)
- A connectivity problem has occurred (ECI_ERR_RESOURCE_SHORTAGE)
- The specified CICS server is not available (ECI_ERR_NO_CICS)

For an XA transaction, if a request is retried using a CICS request exit, the retry must use the same protocol as the original request. For example, a request that was originally attempted over EXCI cannot be retried over IPIC. If, on retry, the exit tries to change the protocol used, the ERROR_EXIT_RETRY_INVALID return code is returned to the Client application and message CTG8468E is written to the error log.

You can pass a command to a CICS request exit dynamically using the CREXIT administration option, described in the *CICS Transaction Gateway for z/OS: Administration Guide*.

About this task

To configure and deploy a CICS request exit follow the steps given below.

Procedure

1. Create a Java class that implements the **com.ibm.ctg.ha.CICSRequestExit** interface.
2. Compile the Java class and package it into a JAR file.
3. Copy the JAR file to a location in your HFS accessible by the Gateway daemon.
4. Update the CLASSPATH environment variable in the Gateway daemon configuration to include the location of the JAR file containing your exit.

5. Specify the fully-qualified package name of your exit class by using the **cicsrequestexit** parameter in the configuration file (ctg.ini); for more information, see the *CICS Transaction Gateway for z/OS: Administration Guide*. For example, to deploy the sample RoundRobinCICSRequestExit, specify this:

   ```
   cicsrequestexit=com.ibm.ctg.samples.ha.RoundRobinCICSRequestExit
   ```

6. Start the Gateway daemon.

Related information

CICS request exit

Writing a CICS request exit

Methods implemented by the CICS request exit interface.

The CICS request exit must implement the com.ibm.ctg.ha.CICSRequestExit interface. Two methods defined by the interface must be implemented by the class:

- getRetryCount
- getCICSServer

If the CICS request exit fails to load and then initialize, the Gateway daemon fails to start. When the Gateway daemon loads the CICS request exit class, the default constructor is called, enabling any setup information to be initialized before the CICS request exit is used.

getRetryCount

If the initialization is successful; that is, no exceptions are thrown from the default constructor, the getRetryCount method is called to determine how many times a request for a new transaction can be retried following a retryable error. The getRetryCount method is called once only, so the value will be constant for the lifetime of the Gateway daemon and used for the start of every transaction.

getCICSServer

The getCICSServer method is called by the Gateway daemon at the start of each unit of work to determine the CICS server for all ECI requests within that unit of work. A unit of work is started by a SYNCONRETURN ECI request, the first ECI request in an extended LUW, or the first request in an XA transaction. If the request fails with a retryable error and the maximum number of retries has not been reached, the getCICSServer method is called again to allow a different CICS server to be used. However, if the request fails and the maximum number of retries has been reached the error from the last request is returned to the Java client application. See the Javadoc information for details of the request data available to a getCICSServer method. The retryable errors are:

- ECI_ERR_NO_CICS
- ECI_ERR_CICS_DIED
- ECI_ERR_RESOURCE_SHORTAGE

InvalidRequestException
If the getCICSServer method determines that the request is invalid; for example, an invalid user ID or the exit wants to stop retrying before the maximum is reached, it can throw a com.ibm.ctg.ha.InvalidRequestException that stops the request running and returns an ECI_INVALID_CALL_TYPE to the caller.

EventFired
The EventFired method is called if:

- A CICSRequestExit is disabled at shutdown of the Gateway daemon, on closing the JavaGateway object, or on an error.
- A Gateway daemon receives an administration request for the CICS request exit that includes a command string.

This method is called for each defined ExitEvent. The CICS request exit can selectively process these using the event parameter.

Sample CICS request exits

Two sample CICS request exits are provided: The first one returns the CICS server to use for an ECI request. The second one supports workload management using a round-robin algorithm.

Location of sample files

The source code for the CICS request exit samples is provided in the following location:<install_path>/samples/java/com/ibm/ctg/samples/ha

BasicCICSRequestExit

This sample shows you how to implement a basic CICS request exit. The getCICSServer method returns the CICS server to be used on an ECI request, based on a predefined server mapping. If the CICS server on the ECI request is defined in the server mapping, the actual CICS server that it maps to is returned. If the CICS server on the ECI request is not defined in the server mapping, the CICS server is returned unchanged.

RoundRobinCICSRequestExit

This sample shows you how to implement a CICS request exit to perform workload management. Each time that the getCICSServer method is called, it returns the next CICS server, in a threadsafe manner, from a predefined list. The CICS server specified on the ECI request by the application is ignored.

The retry count is set so that each server in the list is called at most once for each request.

Using the CICS request exit samples

Before the samples can be used, the code has to be changed to reference known CICS servers.

When these changes have been made, the class must be compiled, for example by using the **javac** command.

To configure each sample exit for use in a specific environment, the following details are provided:

BasicCICSRequestExit

The constructor for this class populates a hash table with mappings between a name that would be used by the Java client application and an actual CICS server. Change the contents of the hash table so that there is a mapping between the CICS server specified on the ECI request, by the Java client application, and an actual CICS server.

RoundRobinCICSRequestExit

The list of available CICS servers is contained in the serverList array. Change the values stored in this array to a list of actual CICS servers.

Chapter 10. Sample programs

A wide selection of sample programs for the supported programming languages are included with CICS Transaction Gateway.

The sample programs that run on z/OS are located under the Unix System Services product install `samples` directory, and in the product MVS dataset SCTGSAMP. Each sample JCL job has comments that describe how to use and customize the file. Make a copy of the SCTGSAMP library and customize the copy. The sample programs that run on non-z/OS platforms, such as C, ECIv2, and .NET, are included in the ctgredist package which is located in the <CTG install location>/deployable directory.

Test the CICS Transaction Gateway

Unix system services script `samples/ctgtest` tests the basic installation and configuration of the CICS Transaction Gateway. It is designed to be run without user input so that it can be run from batch JCL.

The ctgtest script can be run using sample JCL jobs from the SCTGSAMP library. CTGTESTR is used to test a remote mode configuration and CTGTESTL is used to test a local mode configuration.

Note: CTGTESTL can only test local mode connections to CICS using the EXCI protocol.

The script uses the EciB1 sample program, which requires the sample server program ECO1 to be installed on the CICS server. It selects the first CICS server in the list to test with. If the z/OS URL and Port are present, it requires the CICS Transaction Gateway to be running. The script is in the samples directory.

The command format is:

`ctgtest` *`user_ID password z/OS_URL z/OS_port`*

All parameters are optional but, because they are positional, any that are not required must have dummy entries. If password checking is enabled the script uses the user ID and password provided. The parameters are:

user_ID
: A valid user ID on the z/OS system.

password
: A valid password for the user ID.

z/OS_URL
The TCP/IP name or address of this z/OS image. To test without knowing the real address use **localhost**.

z/OS_port
The TCP/IP port defined in the configuration file (ctg.ini) protocol definition. The default is 2006.

CICS server applications

To run the sample programs you need the correct server programs and transactions built and available on your CICS server. The sample server programs are in the <install_path>/samples/server directory.

The following COBOL programs are provided:

EC01.CCP
The first sample ECI server program. This program returns the current date and time in its COMMAREA.

EC02.CCP
The second sample ECI server program. This program returns the number of times it has been run in a unit of work in its COMMAREA.

EC03.CCP
The third sample ECI server program. This program receives CHAR container INPUTDATA and performs CICS GET CONTAINER commands to return the contents, length and CCSID of the container. This program returns the length in a BIT container and the CCSID in a CHAR container, plus the date and time on the CICS server and a message containing the input data or a failure message.

For information about how to build and install these programs, refer to your CICS server documentation.

Java client applications

Java samples are provided to illustrate the use of the ECI Request and security APIs.

To use the sample programs, you must ensure that the required server programs or transactions are installed on your CICS server. These sample programs do not demonstrate all the techniques required for a large application. They are not templates and should not be used as the basis for developing production applications.

Overview

Compiling and running the sample programs, and connecting to CICS Transaction Gateway.

Compiled sample programs

Compiled Java sample programs are provided in <install_path>/classes/ ctgsamples.jar.

The source for these sample programs is in the <install_path>/samples/java directory under the package structure, which is in the following form:

```
com.ibm.ctg.samples.type_of_sample
```

If you recompile the Java sample programs, use a supported level of the SDK; for more information, see Supported Software for CICS Transaction Gateway products.

Running the sample programs

To run the sample programs, ensure that ctgsamples.jar, ctgclient.jar, and ctgserver.jar are referenced in your class path.

These files are in the classes directory.

```
CLASSPATH=<install_path>/classes/ctgsamples.jar
:<install_path>/classes/ctgclient.jar
:<install_path>/classes/ctgserver.jar
```

Alternatively you can run the sample programs by using the Java **-classpath** option, specifying the same information.

When running a sample program, if you provide any command line parameters, you must enter them in the order specified by the usage statement of the particular sample program.

Connecting to CICS Transaction Gateway

You can provide a URL that specifies the location of the CICS Transaction Gateway to which you want to connect.

This should be of the form *protocol*://*address*. For example, for a remote mode connection using the SSL protocol to a Gateway daemon with IPv4 address "myserver.test.com":

```
ssl://myserver.ibm.com
```

If you are using IPv6, you must enclose the address in square brackets. For example, for a remote mode connection using the TCP/IP protocol to a Gateway daemon with IPv6 address "[2002:914:fc12:632:7:36:66:134]":

```
tcp://[2002:914:fc12:632:7:36:66:134]
```

If you want to use local mode, the URL is “local:”.

ECI Java base classes sample programs

CICS Transaction Gateway includes basic, intermediate, and advanced ECI Java base classes sample programs.

Basic - EciB1

This sample program lists the systems defined in the configuration file (ctg.ini) and allows you to choose the one to which an ECI request is sent. This request is then sent and the date and time are returned in ASCII by transaction EC01, alongside a representation in hexadecimal.

Usage: java com.ibm.ctg.samples.eci.EciB1 [Gateway Url] [Gateway Port Number] [SSL Keyring] [SSL Password]

If translation of the date and time to ASCII is required, a conversion template needs to be created for EC01 on the server. Refer to Configuring data conversionthe information about configuring data conversion in the *CICS Transaction Gateway for z/OS Administration Guide* for further details on conversion templates.

Basic - EciB2

This sample program can be used to test ECI requests to CICS applications and allows the user to control the values of the parameters used in the sample from the command line.

```
Usage:
  java com.ibm.ctg.samples.eci.EciB2  [jgate=gateway_URL]
                                      [jgateport=gateway_port]
                                      [clientsecurity=client_security_class]
                                      [serversecurity=server_security_class]
                                      [server=cics_server_name or IPIC_url]
                                      [userid=cics_userid]
                                      [password=cics_password]
                                      [prog<0..9>=prog_name]
                                      [commarea=comm_area]
                                      [commarealength=comm_area_length]
                                      [status]
                                      [trace]
                                      [ascii | ebcdic | asis]
```

You can specify the Gateway URL and relevant ECI request parameters as input to the application, and either call a single CICS program or call multiple CICS programs within one extended LUW. You can control the code page of the COMMAREA flowed on the ECI request as an input parameter.

Basic - EciB3

This sample program shows the basic use of the channels and containers components of the CICS Transaction Gateway API.

Usage: java com.ibm.ctg.samples.eci.EciB3 [Gateway URL] [Gateway Port Number] [SSL Keyring] [SSL Password]

When using remote mode, the sample program connects to a Gateway daemon and obtains a list of available CICS servers. It then flows an ECI request for CICS program EC03 to the selected server.

When using local mode, the sample program prompts for the URL of a CICS TCPIPSERVICE listening for IPIC requests, before flowing an ECI request for CICS program EC03 to that CICS server. This URL is of the form *protocol*://*hostname*:*port*, where *protocol* is "tcp" or "ssl".

Intermediate - EciI1

This sample program shows the use of the ECI Request classes using an asynchronous extended request and using a callbackable object.

Usage: java com.ibm.ctg.samples.eci.EciI1 [Gateway URL] [Port] [SSL keyring] [SSL password]

The sample program queries the Gateway daemon for a list of servers, then runs transaction EC02 on the selected server.

You can provide a gateway URL and port number, along with an SSL keyring and SSL password as command line parameters. If you do not provide a URL, the sample programs default to local.

When you start the Gateway daemon, ensure that the ctgsamples.jar file is referenced in the class path.

ClientCompression and ServerCompression:

This sample program also illustrates the use of the ClientCompression and ServerCompression samples to compress the data stream between the client application and the Gateway daemon.

Because the request is extended, you have the option of running the transaction again within the same logical unit of work (LUW). If you do not run the transaction again, you are prompted to back out or commit the LUW. See "Security" on page 110 for more information.

Advanced - EciA1

This sample program illustrates the use of the ECI request classes within the framework of a servlet.

To compile EciA1, the servlet packages (2.2) javax.servlet and javax.servlet.http must be referenced in the class path or added to the <install_path>/samples/java directory.

When the servlet is initialized, it reads values supplied for the Gateway URL, SSL classname and SSL password if they have been specified as initialization parameters. Otherwise the default URL is local. The initial page displays the URL of the connected Gateway daemon and a number of areas for user input: Server, Program, CommArea Size, User ID, and Password.

- Server is a combination box containing the names of all the servers listed in the configuration file (ctg.ini).
- Program is a list limited to EC01 and EC02; these must be available on the CICS Server.
- CommArea Size can be set for EC01 only; for EC02 the size is always 50.
- The user ID and password can be specified in the two remaining text areas.

The servlet takes the submitted data and runs the program, automatically backing out if the transaction terminates abnormally, or committing if it runs successfully. The results of the transaction are displayed on a new page.

You can use a servlet properties file to provide initialization parameters. The sample servlet looks for the following case-sensitive parameters:

- GatewayURL
- SSLClassname
- SSLPassword

For example:

```
servlet.EciA1.initArgs=GatewayURL=tcp://localhost:2006
```

If your J2EE application server requires Java 2 Security permissions, or if you have enabled this facility on your server, you might have to give the permissions described in "Using a Java 2 Security Manager" on page 49.

Refer to the documentation for your J2EE application server on setting servlet initialization parameters.

J2EE applications

The J2EE sample programs are provided in <install_path>/samples/java/com/ibm/ctg/samples/j2ee.

J2EE ECIDateTime sample program

This sample program uses the ECI resource adapter, and calls the CICS program EC01. The program uses an enterprise bean that makes CCI calls; a client to the enterprise bean is provided.

The ECIDateTime sample program includes the following files:

ECIDateTimeBean.java
The enterprise bean ECIDateTime implementation code

ECIDateTime.java
The enterprise bean Remote interface

ECIDateTimeHome.java
The enterprise bean Home interface

JavaStringRecord.java
The sample program record interface that wraps an ECI COMMAREA

ECIDateTimeClient.java
The client for the enterprise bean

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements (in this case, the execute() method.) The Home interface manages the life cycle of the enterprise bean.

ECIDateTimeClient looks up the enterprise bean as ECIDateTimeBean1 in Java Naming Directory Interface (JNDI), and then narrows the search to a specific object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. This remote method in turn looks up the resource adapter's connection factory (an instance of the resource adapter) under the name ECI and runs EC01 in CICS and gets the date and time back as a COMMAREA, which it then returns to the caller (the client application).

To use the sample program:

1. Deploy the CICS ECI resource adapter; this is a file called <install_path>/deployable/cicseci.rar.
2. Create a connection factory with parameters that are valid for your CICS server environment (on WebSphere Application Server, these settings are on the Custom properties tab of the J2C connection factory settings). See the information about deploying resource adapters in the *CICS Transaction Gateway Administration Guide* for more information. The connection factory must have a JNDI name of ECI for the sample program to work.
3. Deploy your enterprise bean. This automatically generates code that deals with remote method calls to your enterprise bean by the enterprise bean client. This process is specific to your J2EE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you are asked for might include:

   Transaction type
   This can be set to Container-managed or Bean-managed. This

determines whether you want to control transactions yourself. The J2EE application server manages Container-managed transactions; if prompted, select this type for the sample program.

Enterprise bean type
ECIDateTime is a stateless session bean.

JNDI name
The enterprise bean client uses JNDI to look up the enterprise bean. This allows you to find the name of the enterprise bean in the directory. The ECIDateTimeClient requires this name to be set to ECIDateTimeBean1.

Resource references
The enterprise bean refers to another resource, the ECI resource adapter. To enable this to happen, you need to:

a. Deploy a ConnectionFactory for the ECI resource adapter with a JNDI name of ECI.
b. List this ConnectionFactory as a resource reference for this enterprise bean.

4. Run the Client application. You can run it from a command line, but if using WebSphere, use the launchClient utility, which sets up the necessary parameters to allow you to talk to the JNDI directory in WebSphere to find the ECIDateTime enterprise bean. The application returns the current date and time from CICS application EC01.

J2EE EC03Channel sample program

This sample program calls the CICS program EC03 using the CICS ECI resource adapter or CICS ECI XA resource adapter. The program uses an enterprise bean that makes ECI calls; a client to the enterprise bean is provided.

The EC03Channel sample program includes the following files:

EC03ChannelBean.java
The implementation of the EC03 channel EJB

EC03Channel.java
The Remote interface for the EC03 channel EJB

EC03ChannelHome.java
The Home interface for the EC03 channel EJB

EC03ChannelClient.java
A basic client which calls the EC03 channel EJB

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements (in this case, the execute() method.) The Home interface manages the life cycle of the enterprise bean.

EC03ChannelClient looks up the enterprise bean as EC03ChannelHome in Java Naming Directory Interface (JNDI), and then narrows the search to a specific object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. This remote method in turn looks up the resource adapter's connection factory (an instance of the resource adapter) under the name ECI and runs EC03 in CICS, passing in a channel with one container. When the ECI call program returns, the containers returned from the program are enumerated and placed into a HashMap, which is then returned to the client.

To use the sample program:

1. Deploy the CICS ECI resource adapter (cicseci.rar) or CICS ECI XA resource adapter (cicseciXA.rar); these are located in the deployable directory of the CICS Transaction Gateway install path.
2. Create a connection factory with parameters that are valid for your CICS server environment (on WebSphere Application Server, these settings are on the Custom properties tab of the J2C connection factory settings). See the information about deploying resource adapters in the *CICS Transaction Gateway Administration Guide* for more information. The connection factory must have a JNDI name of "ECI" for the sample program to work.
3. Deploy your enterprise bean. This automatically generates code that deals with remote method calls to your enterprise bean by the enterprise bean client. This process is specific to your J2EE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you are asked for might include:

   Transaction type
   Can be set to container-managed or bean-managed. This determines whether you want to control transactions yourself. The J2EE application server manages Container-managed transactions; if prompted, select this type for the sample program.

   Enterprise bean type
   EC03Channel is a stateless session bean.

   JNDI name
   The enterprise bean client uses JNDI to look up the enterprise bean. This allows the enterprise client to find the name of the enterprise bean in the directory.

   Resource references
   The enterprise bean refers to a connection factory. To enabled this

to happen you need to add the connection factory defined in Step 2 on page 107 as a resource reference for this enterprise bean.

4. Run the Client application. You can run it from a command line, but if using WebSphere, use the launchClient utility, which sets up the necessary parameters to allow the enterprise client to look up the bean in the JNDI directory in WebSphere to find the EC03Channel enterprise bean. The application calls the bean, passing a string of text to the EC03 program, and displays the contents of the containers returned.

ECI Version 2 applications

The ECIv2 sample programs are written in C and can be found in the <install_path>/samples/c/eci_v2 directory. They can be built and run on any supported platform other than z/OS, but can connect to a Gateway daemon running on z/OS.

Basic - ctgecib1

The ctgecib1 basic ECIv2 sample program is written in C and can be found in the <install_path>/samples/c/eci_v2 directory.

To build the sample program on UNIX and Linux, change to this directory and issue the following command:

```
make -f samp.mak
```

To build the sample on Windows, change to this directory and run the supplied command file ctgecib1mak.cmd. The command file compiles the program for Windows using the Microsoft Visual C++ 2008 or Microsoft Visual C++ 2010 compiler.

Once compiled, the sample program can be executed using the following command:

```
ctgecib1 [host name] [port number]
```

This sample program lists the systems defined on a remote CICS Transaction Gateway, and allows you to select the one to which an ECI program call is made. This call is then made and the date and time are returned by program EC01.

Basic - ctgecib3

The ctgecib3 basic ECIv2 sample program is written in C and can be found in the <install_path>/samples/c/eci_v2 directory.

To build the sample on UNIX and Linux, change to this directory and issue the following command:

```
make -f samp.mak
```

To build the sample on Windows, change to this directory and run the supplied command file ctgecib3mak.cmd. The command file compiles the program using the Microsoft Visual C++ 2008 or Microsoft Visual C++ 2010 compiler.

Once compiled, the sample program can be executed using the following command:

```
ctgecib3 [host name] [port number]
```

The sample program lists the systems defined on a remote CICS Transaction Gateway, and allows you to select the one to which an ECI program call is made. The supplied CICS program EC03 is called with a channel and a single CHAR container. The program updates the channel by adding new containers. The sample program lists all the containers that are returned from the EC03 program.

.NET applications

The .NET sample programs are provided in C# and Visual Basic .NET.

Basic - EciB1

The EciB1 basic .NET sample program is provided in C# and Visual Basic .NET. The C# sample is in the <install_path>/samples/csharp/eci directory, and the Visual Basic .NET sample is in the <install_path>/samples/vb/eci directory.

You can compile the sample using Microsoft Visual Studio or from a Windows command prompt. Project files are provided for Visual Studio 2008.

To build the sample program from a command prompt, change to the appropriate directory and run the supplied command file EciB1mak.cmd. The file compiles the program for Windows using the C# or Visual Basic .NET compiler which are provided by the Microsoft .NET Framework.

When compiled, you can execute the sample program using the following command:

```
EciB1 [host name] [port number]
```

The sample program lists the systems defined on a remote CICS Transaction Gateway, and allows you to select the one to which an ECI program call is made. The call is made and the date and time are returned by program EC01.

Exit sample programs

The exit sample programs provide examples of user exit programs.

Security

The security sample programs compress data passed between the client application and the CICS server.

ClientCompression
Implements ClientSecurity and is referenced in the intermediate Java sample programs.

ServerCompression
Implements ServerSecurity and is referenced in the intermediate Java sample programs.

SSLServerCompression
Implements JSSEServerSecurity. This sample program shows the JSSE SSL security exit.

Request monitoring user exit sample programs

Two request monitoring user exit sample programs are supplied.

com.ibm.ctg.samples.requestexit.BasicMonitor.java

This sample program shows the basic use of the CICS Transaction Gateway request monitoring exits. The sample program writes the data available at each exit point to STDOUT or to a file specified by the Java property com.ibm.ctg.samples.requestexit.out.

To enable the sample program on the Gateway daemon you must do the following:

1. Add ctgsamples.jar to the class path used when starting the CICS Transaction Gateway.
2. Set the **requestexits** value in the configuration file (ctg.ini) to com.ibm.ctg.samples.requestexit.BasicMonitor.
3. Data is written to STDOUT by default. To capture data to a file use the Java property com.ibm.ctg.samples.requestexit.out, for example:

   ```
   CTGSTART_OPTS=-j-Dcom.ibm.ctg.samples.requestexit.out=/hfs.file
   ```

com.ibm.ctg.samples.requestexit.ThreadedMonitor.java

This sample program extends the BasicMonitor sample program. The ThreadedMonitor sample program uses a background thread to reduce the overhead for each monitored request. The sample program writes the data available at each exit point to STDOUT or to a file specified by the Java property com.ibm.ctg.samples.requestexit.out. Errors are logged to STDERR or to a file specified by the Java property com.ibm.ctg.samples.requestexit.err.

To enable the sample program on the Gateway daemon you must do the following:

1. Add ctgsamples.jar to the class path used when starting CICS Transaction Gateway.
2. Set the **requestexits** value in the configuration file to com.ibm.ctg.samples.requestexit.ThreadedMonitor.
3. Data is written to STDOUT by default. To capture data to a file use the Java property com.ibm.ctg.samples.requestexit.out, for example:

   ```
   CTGSTART_OPTS=-j-Dcom.ibm.ctg.samples.requestexit.out=/hfs.file
   ```
4. Errors are written to STDERR by default. To capture data to a file use the Java property com.ibm.ctg.samples.requestexit.err, for example:

   ```
   CTGSTART_OPTS=-j-Dcom.ibm.ctg.samples.requestexit.err=/hfs.error.file
   ```
5. An alert is logged for any transactions that take longer than 15 seconds. To change this time, use the Java property com.ibm.ctg.samples.requestexit.lrt, for example:

   ```
   CTGSTART_OPTS=-j-Dcom.ibm.ctg.samples.requestexit.lrt=5000
   ```

   (time is in milliseconds).

The sample program code details additional optional parameters that can be set.

CICS request exit samples

Two CICS request exit samples are provided as examples of how to write and structure a CICS request exit implementation.

BasicCICSRequestExit

This sample program shows you how to implement a basic CICS request exit.

The getCICSServer method returns the CICS server to be used on an ECI request, based on a predefined server mapping. If the CICS server on the ECI request is defined in the server mapping, the actual CICS server that it maps to is returned. If the CICS server on the ECI request is not defined in the server mapping, the CICS server is returned unchanged.

RoundRobinCICSRequestExit

This sample program shows you how to implement a CICS request exit to perform workload management.

Each time that the getCICSServer method is called, it returns the next CICS server from a predefined list in a threadsafe manner. The CICS server specified on the ECI request by the application is ignored. The retry count is set so that each server in the list is called at most once for each request.

Statistics sample programs

The statistics sample programs demonstrate the statistics API for C and Java and processing of SMF records.

Statistics API samples

These sample programs demonstrate how to gather and display statistics.

Statistics API for C sample

The statistics sample program is written in C and can be found in the SCTGSAMP library.

The CTGSTAT1 C sample program demonstrates the following functions:

1. Connecting to the statistical API port.
2. Querying running Gateway daemons for statistics in the connection manager resource group.
3. Obtaining values for these statistics.
4. Retrieving and displaying information about the Gateway daemon running time and the total number of requests made.

Sample JCL job SCTGSAMP(CTGSTJOB) is provided to compile, link, and run the sample program. Instructions in the JCL explain how to customize it to run it successfully.

Statistics API for Java sample

The statistics sample program is written in Java and can be found in `samples/java/com/ibm/ctg/samples/stats/Ctgstat1.java`.

The ctgstat1 Java sample program demonstrates the following functions:

1. Connecting to the statistical API port.
2. Querying running Gateway daemons for statistics in the connection manager resource group.
3. Obtaining values for these statistics.
4. Retrieving and displaying information about the Gateway daemon running time and the total number of requests made.

A pre-compiled version of com.ibm.ctg.samples.stats.Ctgstat1 is included in the Java archive file `classes/ctgsamples.jar`.

SMF viewer sample program

The SMF viewer sample program is written in C and can be found in the library member SCTGSAMP(CTGSMFRD).

This sample program shows the basic use of the CICS Transaction Gateway SMF recording facility. It demonstrates formatting and basic filtering on

statistics information written to SMF by the CICS Transaction Gateway. This sample program requires the SMF records have been extracted into a dataset by the IFASMFDP utility.

Sample JCL job SCTGSAMP(CTGSMFB) is provided to build and link the sample. Sample JCL job SCTGSAMP(CTGSMFR) is provided to run the sample program CTGSMFRD. Instructions are provided in each sample JCL job explaining the customizations required.

Request monitoring exit sample programs

CICS Transaction gateway includes sample request monitoring exits, which provide simple monitoring solutions for use with the Gateway daemon and Gateway classes.

com.ibm.ctg.samples.requestexit.BasicMonitor.java

This sample shows the basic use of the CICS Transaction Gateway request monitoring exits. The sample sends the data available at each exit point to STDOUT or to a file specified by a system property. To enable the sample on the Gateway daemon:

- Set the *requestexits* value in the configuration file to `com.ibm.ctg.samples.requestexit.BasicMonitor`.

com.ibm.ctg.samples.requestexit.ThreadedMonitor.java

This sample extends the BasicMonitor. The ThreadedMonitor uses a background thread to reduce the overhead for each monitored request. The sample sends the data available at each exit point to STDOUT or to a file specified by a system property. Errors are logged to STDERR or to a file specified by a system property. To enable the sample on the Gateway daemon:

- Set the *requestexits* value in the configuration file to `com.ibm.ctg.samples.requestexit.ThreadedMonitor.java`.

The sample code details additional optional parameters that you can set.

Product library and related literature

The CICS Transaction Gateway product library contains information on administration, messages and programming; this information is available in this information center, and is also available in PDF form. IBM Redbooks® publications provide a further source of information about working with CICS Transaction Gateway.

CICS Transaction Gateway books

The books in the library cover administration, programming and messages.

- *CICS Transaction Gateway: Messages*, SC34-7061-02 describes the error messages that can be generated by the CICS Transaction Gateway.

Sample configuration documents

Several sample configuration documents are available in portable document format (PDF).

These documents give step-by-step guidance for configuring CICS Transaction Gateway for communication with CICS servers, using various protocols. They provide detailed instructions that extend the information in the CICS Transaction Gateway library.

Visit the following Web site:

`www.ibm.com/software/cics/ctg`

and follow the **Library** link.

IBM Redbooks publications

IBM Redbook titles are available on a wide range of subjects relevant to CICS Transaction Gateway programming, installation, operation and troubleshooting.

The following International Technical Support Organization (ITSO) Redbook publication contains many examples of client/server configurations:

- *CICS Transaction Gateway V5 - The WebSphere Connector for CICS, SG24-6133* describes how to use the different protocols (TCP/IP, TCP62, APPC and EXCI) for communication with CICS, and how to securely connect a Java client application to a CICS region.

- *Revealed! Architecting Web Access to CICS, SG24-5466* is intended for IT architects who select, plan, and design SOA solutions that make use of CICS assets
- *Enterprise JavaBeans for z/OS and z/OS CICS Transaction Server V2.2, SG24-6284* describes the EJB and the way it has been implemented within the CICS architecture, also describes how to set up and configure a CICS region to support EJBs
- *Java Connectors for CICS: Featuring the J2EE Connector Architecture, SG24-6401* provides information on developing J2EE applications.
- *Systems Programmer's Guide to Resource Recovery Services (RRS), SG24-6980-00* describes how to use RRS in various scenarios.
- *Communications Server for z/OS V1R2 TCP/IP Implementation Guide, SG24-6517-00* provides information on using Communications Server for z/OS V1R2, including load balancing.
- *Redpaper: Transactions in J2EE, REDP-3659-00* discusses transactions in the J2EE environment, including one-phase commit and two-phase commit XA transactions.
- *Exploring Systems Monitoring for CICS Transaction Gateway V7.1 for z/OS, SG24-7562-00* looks at product installation and customization, and also covers systems monitoring for CICS Transaction Gateway using IBM Tivoli® OMEGAMON® XE, and statistics provided by CICS Performance Analyzer

The ITSO Redbooks are available from various sources. For the latest information, see:

`www.ibm.com/redbooks/`

Other useful information

Other sources of useful information include the CICS Transaction Server information center and associated publications.

The CICS Transaction Server for z/OS V4.1 information center is located at:

http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp

CICS Transaction Server publications

The CICS Transaction Server books on security, inter-product communication and problem determination also provide a useful source of information.

CICS Transaction Server for z/OS RACF Security Guide, SC34-7003

CICS inter-product communication

The following books describe the intercommunication facilities of the CICS server products:

- *CICS Family: Interproduct Communication*, SC34–6853
- *CICS Transaction Server for z/OS CICS External Interfaces Guide*, SC34-7019
- *CICS Transaction Server for z/OS: Intercommunication Guide*, SC34-7018
- *CICS TS for VSE: Intercommunication Guide*, SC33-0701
- *CICS Transaction Server for iSeries: Intercommunication*, SC41-5456

The first book above is a CICS family book containing a platform-independent overview of CICS inter-product communication.

CICS problem determination

The following books describe the problem determination facilities of the CICS server products:

- *Transaction Server for Windows Problem Determination*, GC34-6210
- *CICS Transaction Server for z/OS Problem Determination Guide*, SC34-7034
- *CICS TS for VSE 2.3 Problem Determination Guide*, SC33-0716
- *CICS Transaction Server for iSeries: Problem Determination*, SC41-5453
- *TXSeries for Multiplatforms: Problem Determination Guide*, SC34-6636

Accessibility

Accessibility features help users with a physical disability, for example restricted mobility or limited vision, to use information technology products successfully. CICS Transaction Gateway provides accessibility by enabling keyboard-only operation.

For more information about the IBM commitment to accessibility, visit the IBM Accessibility Center.

Glossary

This glossary defines the terms and abbreviations used in CICS Transaction Gateway and in the information centers.

A

abnormal end of task (abend)
The termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve.

Advanced program-to-program communication (APPC)
An implementation of the SNA/SDLC LU 6.2 protocol that allows interconnected systems to communicate and share the processing of programs. The Client daemon uses APPC to communicate with CICS server systems.

APAR See *Authorized program analysis report*.

API See *application programming interface*.

APPC See *Advanced program-to-program communication*.

application programming interface (API)
A functional interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

APPLID

1. On CICS Transaction Gateway: The application identifier that is used to identify CICS Transaction Gateway connections on the CICS server and tasks in a CICSplex. See also *APPLID qualifier* and *fully-qualified APPLID*.
2. On CICS Transaction Server: The name by which a CICS system is known in a network of interconnected CICS systems. CICS Transaction Gateway application identifiers do not need to be defined in SYS1.VTAMLST. The CICS APPLID is specified in the APPLID system initialization parameter.

APPLID qualifier
Optionally used as a high-level qualifier for the APPLID to form a fully-qualified APPLID. See also *APPLID* and *fully-qualified APPLID*.

ARM See *automatic restart manager*.

Authorized program analysis report (APAR)
A request for correction of a defect in a current release of an IBM-supplied program.

ATI See *automatic transaction initiation*.

attach In SNA, the request unit that flows on a session to initiate a conversation.

Attach Manager
The component of APPC that matches attaches received from remote computers to accepts issued by local programs.

autoinstall
A method of creating and installing resources dynamically as terminals log on, and deleting them at logoff.

automatic restart manager (ARM)
A z/OS recovery function that can improve the availability of specific batch jobs or started tasks, and therefore result in faster resumption of productive work.

automatic transaction initiation (ATI)
The initiation of a CICS transaction by an internally generated request, for example, the issue of an EXEC CICS START command or the reaching of a transient data trigger level. CICS resource definition can associate a trigger level and a transaction with a transient data destination. When the number of records written to the destination reaches the trigger level, the specified transaction is automatically initiated.

B

bean A definition or instance of a JavaBeans component. See also *JavaBeans*.

bean-managed transaction
A transaction where the J2EE bean itself is responsible for administering transaction tasks such as committal or rollback. See also *container-managed transaction*.

BIND command
In SNA, a request to activate a session between two logical units (LUs).

business logic
The part of a distributed application that is concerned with the application logic rather than the user interface of the application. Compare with *presentation logic*.

C

CA See *certificate authority*.

callback
A way for one thread to notify another application thread that an event has happened.

certificate authority (CA)
In computer security, an organization that issues certificates. The certificate authority authenticates the certificate owner's identity and the services that the owner is authorized to use. It issues new certificates and revokes certificates from users who are no longer authorized to use them.

change-number-of-sessions (CNOS)
An internal transaction program that regulates the number of parallel sessions between the partner LUs with specific characteristics.

channel
A channel is a set of containers, grouped together to pass data to CICS. There is no limit to the number of containers that can be added to a channel, and the size of individual containers is limited only by the amount of storage that you have available.

CICS connectivity components
A generic reference to the Client daemon, EXCI, and the IPIC protocol.

CICS connectivity components
The Client daemon, the EXCI (External CICS Interface), and the IPIC (IP Interconnectivity) protocol are collectively called the 'CICS connectivity components'. The Client daemon handles the TCP/IP and the SNA protocols.

CICS Request Exit
An exit that is invoked by the CICS Transaction Gateway for z/OS at run time to determine which CICS server to use.

CICS server name
A defined server known to CICS Transaction Gateway.

CICS TS
Abbreviation of CICS Transaction Server.

class In object-oriented programming, a model or template that can be instantiated to create objects with a common definition and therefore, common properties, operations, and behavior. An object is an instance of a class.

CLASSPATH
In the execution environment, an environment variable keyword that specifies the directories in which to look for class and resource files.

Client API
The Client API is the interface used by Client applications to interact with CICS using the Client daemon. See External Call Interface, External Presentation Interface, and External Security Interface.

Client application
The client application is a user application written in a supported programming language that uses one or more of the CICS Transaction Gateways APIs.

Client daemon
The Client daemon manages TCP/IP, SNA, and, on Windows, named pipe connections to CICS servers on UNIX, Linux, and Windows. It processes ECI, EPI, and ESI requests, sending and receiving the appropriate flows to and from the CICS server to satisfy Client application requests. It can support concurrent requests to one or more CICS servers. The CICS Transaction Gateway initialization file defines the operation of the Client daemon and the servers and protocols used for communication.

client/server
Pertaining to the model of interaction in distributed data processing in which a program on one computer sends a request to a program on another computer and awaits a response. The requesting program is called a client; the answering program is called a server.

CNOS See *Change-Number-of-Sessions*.

code page
An assignment of hexadecimal identifiers (code points) to graphic characters. Within a given code page, a code point can have only one meaning.

color mapping file
A file that is used to customize the 3270 screen color attributes on client workstations.

COMMAREA
See *communication area*.

commit phase
The second phase in a XA process. If all participants acknowledge that they are prepared to commit , the transaction manager issues the commit request. If any participant is not prepared to commit the transaction manager issues a back-out request to all participants.

communication area (COMMAREA)
A communication area that is used for passing data both between programs within a transaction and between transactions.

configuration file
A file that specifies the characteristics of a program, system device, server or network.

connection
In data communication, an association established between functional units for conveying information.

In Open Systems Interconnection architecture, an association established by a given layer between two or more entities of the next higher layer for the purpose of data transfer.

In TCP/IP, the path between two protocol application that provides reliable data stream delivery service.

In Internet, a connection extends from a TCP application on one system to a TCP application on another system.

container
A container is a named block of data designed for passing information between programs. A container is a "named COMMAREA" that is not limited to 32KB. Containers are grouped together in sets called channels.

container-managed transaction
A transaction where the EJB container is responsible for administration of tasks such as committal or rollback. See also *bean-managed transaction*.

control table
In CICS, a storage area used to describe or define the configuration or operation of the system.

conversation
A connection between two programs over a session that allows them to communicate with each other while processing a transaction.

conversation security
In APPC, a process that allows validation of a user ID or group ID and password before establishing a connection.

D

daemon
A program that runs unattended to perform continuous or periodic systemwide functions, such as network control. A daemon can be launched automatically, such as when the operating system is started, or manually.

data link control (DLC)
A set of rules used by nodes on a data link (such as an SDLC link or a token ring) to accomplish an orderly exchange of information.

DBCS See *double-byte character set*.

default CICS server
The CICS server that is used if a server name is not specified on an ECI, EPI, or ESI request. The default CICS server name is defined as a product wide setting in the configuration file (ctg.ini).

dependent logical unit
A logical unit that requires assistance from a system services control point (SSCP) to instantiate an LU-to-LU session.

deprecated
Pertaining to an entity, such as a programming element or feature, that is supported but no longer recommended, and that might become obsolete.

digital certificate
An electronic document used to identify an individual, server, company, or some other entity, and to associate a public key with the entity. A digital certificate is issued by a certificate authority and is digitally signed by that authority.

digital signature
Information that is encrypted with an entity's private key and is appended to a message to assure the recipient of the authenticity and integrity of the message. The digital signature proves that the message was signed by the entity that owns, or has access to, the private key or shared secret symmetric key.

distinguished name
The name that uniquely identifies an entry in a directory. A distinguished name is made up of attribute:value pairs, separated by commas. The format of a distinguished name is defined by RFC4514. For more information, see `http://www.ietf.org/rfc/rfc4514.txt`. See also *realm name* and *identity propagation*.

distributed application
An application for which the component application programs are distributed between two or more interconnected processors.

distributed identity
User identity information that originates from a remote system. The distributed identity is created in one system and is passed to one or more other systems over a network. See also *distinguished name* and *realm name*.

distributed processing
The processing of different parts of the same application in different systems, on one or more processors.

distributed program link (DPL)
A link that enables an application program running on one CICS system to link to another application program running in another CICS system.

DLC See *data link control*.

DLL See *dynamic link library*.

domain
In the Internet, a part of a naming hierarchy in which the domain name consists of a sequence of names (labels) separated by periods (dots).

domain name
In TCP/IP, a name of a host system in a network.

domain name server
In TCP/IP, a server program that supplies name-to-address translation by mapping domain names to IP addresses. Synonymous with name server.

dotted decimal notation
The syntactical representation for a 32-bit integer that consists of four 8-bit numbers written in base 10 with periods (dots) separating them. It is used to represent IP addresses.

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. Contrast with *single-byte character set*.

DPL See *distributed program link*.

dynamic link library (DLL)
A collection of runtime routines made available to applications as required.

dynamic server selection
The mapping of a logical server name to an actual server name at run time using logical server definitions, default CICS server definitions, or custom function in a CICS request exit. Previously known as *server name remapping*.

E

EBCDIC
See *extended binary-coded decimal interchange code*.

ECI See *external call interface*.

EJB See *Enterprise JavaBeans*.

emulation program
A program that allows a host system to communicate with a workstation in the same way as it would with the emulated terminal.

emulator
A program that causes a computer to act as a workstation attached to another system.

encryption
The process of transforming data into an unintelligible form in such a way that the original data can be obtained only by using a decryption process.

enterprise bean
A Java component that can be combined with other resources to create J2EE applications. There are three types of enterprise beans: entity beans, session beans, and message-driven beans.

Enterprise JavaBeans (EJB)
A component architecture defined by Sun Microsystems for the development and deployment of object-oriented, distributed, enterprise-level applications (J2EE).

environment variable
A variable that specifies the operating environment for a process. For example, environment variables can describe the home directory, the command search path, the terminal in use, and the current time zone.

EPI See *external presentation interface*.

ESI See *external security interface*.

Ethernet
A local area network that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and transmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

EXCI See *external CICS interface*.

extended binary-coded decimal interchange code (EBCDIC)
A coded character set of 256 8-bit characters developed for the representation of textual data.

extended logical unit of work (extended LUW)
A logical unit of work that is extended across successive ECI requests to the same CICS server.

external call interface (ECI)
A facility that allows a non-CICS program to run a CICS program. Data is exchanged in a COMMAREA or a channel as for usual CICS interprogram communication.

external CICS interface (EXCI)
An MVS application programming interface provided by CICS Transaction Server for z/OS that enables a non-CICS program to call a CICS program and to pass and receive data using a COMMAREA. The CICS application program is started as if linked-to by another CICS application program.

external presentation interface (EPI)
A facility that allows a non-CICS program to appear to CICS as one or more standard 3270 terminals. 3270 data can be presented to the user by emulating a 3270 terminal or by using a graphical user interface.

external security interface (ESI)
A facility that enables client applications to verify and change passwords for user IDs on CICS servers.

F

firewall
A configuration of software that prevents unauthorized traffic between a trusted network and an untrusted network.

FMH See *function management header*.

fully-qualified APPLID
Used to identify CICS Transaction Gateway connections on the CICS server and tasks in a CICSplex. It is composed of an APPLID with an optional network qualifier. See also *APPLID* and *APPLID qualifier*.

function management header (FMH)
One or more headers, optionally present in the leading request units (RUs) of an RU chain, that allow one LU to (a) select a transaction program or device at the session partner and control the way in which the end-user data it sends is handled at the destination, (b) change the destination or the characteristics of the data during the session, and (c) transmit between session partners status or user

information about the destination (for example, a program or device). Function management headers can be used with LU type 1, 4, and 6.2 protocols.

G

gateway
A device or program used to connect two systems or networks.

gateway classes
The gateway classes provide APIs for ECI, EPI, and ESI that allow communication between Java client applications and the Gateway daemon.

Gateway daemon
A long-running Java process that listens for network requests from remote Client applications. It issues these requests to CICS servers using the CICS connectivity components. The Gateway daemon on z/OS processes ECI requests and on UNIX, Windows, and Linux platforms it process EPI and ESI requests as well. The Gateway daemon uses the GATEWAY section of ctg.ini for its configuration.

Gateway group
A set of Gateway daemons that share an APPLID qualifier, and where each Gateway daemon has a unique APPLID within the Gateway group.

gateway token
A token that represents a specific Gateway daemon, when a connection is established successfully. Gateway tokens are used in the C language statistics and ECI V2 APIs.

global transaction
A recoverable unit of work performed by one or more resource managers in a distributed transaction processing environment and coordinated by an external transaction manager.

H

HA group
See *highly available gateway group*.

highly available gateway group (HA group)
A Gateway group that utilizes TCP/IP load balancing, and can be viewed as a single logical Gateway daemon. A Gateway daemon instance in a HA group can recover indoubt XA transactions on behalf of another Gateway daemon within the HA group

host A computer that is connected to a network (such as the Internet or an

SNA network) and provides an access point to that network. The host can be any system; it does not have to be a mainframe.

host address
An IP address that is used to identify a host on a network.

host ID
In TCP/IP, that part of the IP address that defines the host on the network. The length of the host ID depends on the type of network or network class (A, B, or C).

host name
In the Internet suite of protocols, the name given to a computer. Sometimes, host name is used to mean the fully qualified domain name; other times, it is used to mean the most specific subname of a fully qualified domain name. For example, if mycomputer.city.company.com is the fully qualified domain name, either of the following can be considered the host name: mycomputer.city.company.com, mycomputer.

hover help
Information that can be viewed by holding a mouse over an item such as an icon in the user interface.

HTTP See *Hypertext Transfer Protocol*.

HTTPS
See *Hypertext Transfer Protocol Secure*.

Hypertext Transfer Protocol (HTTP)
In the Internet suite of protocols, the protocol that is used to transfer and display hypertext and XML documents.

Hypertext Transfer Protocol Secure (HTTPS)
A TCP/IP protocol that is used by World Wide Web servers and Web browsers to transfer and display hypermedia documents securely across the Internet.

I

ID data
An ID data structure holds an individual result from a statistical API function.

identity propagation
The concept of preserving a user's security identity information (the distributed identity) independent of where the identity information has been created, for use during authorization and for auditing purposes. The distributed identity is carried with a request from the distributed client application to the CICS server, and is incorporated in the access control of the server as part of the authorization process,

for example, using RACF. CICS Transaction Gateway flows the distributed identity to CICS. See also *distributed identity*.

identity propagation login module
A code component that provides support for identity propagation. The identity propagation login module is included with the CICS Transaction Gateway ECI resource adapters (cicseci.rar and cicseciXA.rar), conforms to the JAAS specification and is contained in a single Java class within the resource adapter. See also *identity propagation*.

iKeyman
A tool for maintaining digital certificates for JSSE.

in doubt
The state of a transaction that has completed the prepare phase of the two-phase commit process and is waiting to be completed.

in flight
The state of a transaction that has not yet completed the prepare phase of the two-phase commit process.

independent logical unit
A logical unit (LU) that can both send and receive a BIND, and which supports single, parallel, and multiple sessions. See *BIND*.

<install_path>
This term is used in file paths to represent the directory where you installed the product.

Internet Architecture Board
The technical body that oversees the development of the internet suite of protocols known as TCP/IP.

Internet Protocol (IP)
In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment.

interoperability
The capability to communicate, run programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

IP Internet Protocol.

IPIC See *IP interconnectivity*.

IP address
A unique address for a device or logical unit on a network that uses the IP standard.

IP interconnectivity (IPIC)
The IPIC protocol enables Distributed Program Link (DPL) access from a non-CICS program to a CICS program over TCP/IP, using the External Call Interface (ECI). IPIC passes and receives data using COMMAREAs, or containers.

J

J2EE See *Java 2 Platform Enterprise Edition*

J2EE Connector architecture (JCA)
A standard architecture for connecting the J2EE platform to heterogeneous enterprise information systems (EIS).

Java An object-oriented programming language for portable interpretive code that supports interaction among remote objects.

Java 2 Platform Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications, defined by Sun Microsystems Inc. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that allow multi-tiered, Web-based applications to be developed.

JavaBeans
As defined for Java by Sun Microsystems, a portable, platform-independent, reusable component model.

Java Client application
The Java client application is a user application written in Java, including servlets and enterprise beans, that uses the Gateway classes.

Java Development Kit (JDK)
The name of the software development kit that Sun Microsystems provided for the Java platform, up to and including v 1.1.x. Sometimes used erroneously to mean the Java platform or as a generic term for any software developer kits for Java.

JavaGateway
The URL of the CICS Transaction Gateway with which the Java Client application will communicate. The JavaGateway takes the form `protocol://address:port`. These protocols are supported: `tcp://`, `ssl://`, and `local:`. The CICS Transaction Gateway runs with the default port value of 2006. This parameter is not relevant if you are using the protocol `local:`. For example, you might specify a JavaGateway of `tcp://ctg.business.com:2006`. If you specify the protocol as `local:` you will connect directly to the CICS server, bypassing any CICS Transaction Gateway servers.

Java Native Interface (JNI)
A programming interface that allows Java code running in a Java virtual machine to work with functions that are written in other programming languages.

Java Runtime Environment (JRE)
A subset of the Java Software Development Kit (SDK) that supports the execution, but not the development, of Java applications. The JRE comprises the Java Virtual Machine (JVM), the core classes, and supporting files.

Java Secure Socket Extension (JSSE)
A Java package that enables secure Internet communications. It implements a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TSL) protocols and supports data encryption, server authentication, message integrity, and optionally client authentication.

Java virtual machine (JVM)
A software implementation of a processor that runs compiled Java code (applets and applications).

JDK See *Java development kit*.

JCA See *J2EE Connector Architecture* .

JNI See *Java Native Interface*.

JRE See *Java Runtime Environment*

JSSE See *Java Secure Socket Extension*.

JVM See *Java Virtual Machine*.

K

keyboard mapping
A list that establishes a correspondence between keys on the keyboard and characters displayed on a display screen, or action taken by a program, when that key is pressed.

key ring
In the JSSE protocol, a file that contains public keys, private keys, trusted roots, and certificates.

L

local mode
Local mode describes the use of the CICS Transaction Gateway *local* protocol. The Gateway daemon is not used in local mode.

local transaction
A recoverable unit of work managed by a resource manager and not coordinated by an external transaction manager.

logical CICS server
A logical CICS server name is an alias that can be passed on an ECI request when running in remote mode to CICS Transaction Gateway for z/OS. The alias name is mapped to an actual CICS server name by a LOGICALSERVER definition in the configuration file (ctg.ini).

logical end of day
The local time of day on the 24-hour clock to which a Gateway daemon aligns statistics intervals. If the statistics interval is 24 hours, this is the local time at which interval statistics will be reset and, on z/OS, optionally recorded to SMF. This time is set using the **stateod** parameter in the configuration file (ctg.ini).

logical unit (LU)
In SNA, a port through which an end user accesses the SNA network to communicate with another end user and through which the end user accesses the functions provided by system services control points (SSCP). An LU can support at least two sessions, one with an SSCP and one with another LU, and might be capable of supporting many sessions with other logical units. See also *network addressable unit, primary logical unit, secondary logical unit*.

logical unit 6.2 (LU 6.2)
A type of logical unit that supports general communications between programs in a distributed processing environment.

The LU type that supports sessions between two applications using APPC.

logical unit of work (LUW)
The processing that a program performs between synchronization points

LU See *logical unit*.

LU 6.2 See *logical unit 6.2*.

LU-LU session
In SNA, a session between two logical units (LUs) in an SNA network. It provides communication between two end users, or between an end user and an LU services component.

LU-LU session type 6.2
In SNA, a type of session for communication between peer systems. Synonymous with APPC protocol.

LUW See *logical unit of work*.

M

managed mode
Describes an environment in which connections are obtained from connection factories that the J2EE server has set up. Such connections are owned by the J2EE server.

media access control (MAC) sublayer
One of two sublayers of the ISO Open Systems Interconnection data link layer proposed for local area networks by the IEEE Project 802 Committee on Local Area Networks and the European Computer Manufacturers Association (ECMA). It provides functions that depend on the topology of the network and uses services of the physical layer to provide services to the logical link control (LLC) sublayer. The OSI data link layer corresponds to the SNA data link control layer.

method
In object-oriented programming, an operation that an object can perform. An object can have many methods.

mode In SNA, a set of parameters that defines the characteristics of a session between two LUs.

N

name server
In TCP/IP, synonym for Domain Name Server. In Internet communications, a host that translates symbolic names assigned to networks and hosts into IP addresses.

NAU See *network addressable unit*.

network address
In SNA, an address, consisting of subarea and element fields, that identifies a link, link station, or network addressable unit (NAU). Subarea nodes use network addresses; peripheral nodes use local addresses. The boundary function in the subarea node to which a peripheral node is attached transforms local addresses to network addresses and vice versa. See also *network name*.

network addressable unit (NAU)
In SNA, a logical unit, a physical unit, or a system services control point. The NAU is the origin or the destination of information transmitted by the path control network. See also *logical unit, network address, network name*.

network name
In SNA, the symbolic identifier by which end users refer to a network addressable unit (NAU), link station, or link. See also *network address*.

node type
In SNA, a designation of a node according to the protocols it supports and the network addressable units (NAUs) it can contain. Four types are defined: 1, 2, 4, and 5. Type 1 and type 2 nodes are peripheral nodes; type 4 and type 5 nodes are subarea nodes.

nonextended logical unit of work
See *SYNCONRETURN*.

nonmanaged mode
An environment in which the application is responsible for generating and configuring connection factories. The J2EE server does not own or know about these connection factories and therefore provides no Quality of Service facilities.

O

object In object-oriented programming, a concrete realization of a class that consists of data and the operations associated with that data.

object-oriented (OO)
Describing a computer system or programming language that supports objects.

one-phase commit
A protocol with a single commit phase, that is used for the coordination of changes to recoverable resources when a single resource manager is involved.

OO See *object-oriented*.

P

pacing
A technique by which a receiving station controls the rate of transmission of a sending station to prevent overrun.

parallel session
In SNA, two or more concurrently active sessions between the same two LUs using different pairs of network addresses. Each session can have independent session parameters.

PING In Internet communications, a program used in TCP/IP networks to test the ability to reach destinations by sending the destinations an Internet Control Message Protocol (ICMP) echo request and waiting for a reply.

partner logical unit (PLU)
In SNA, the remote participant in a session.

partner transaction program
The transaction program engaged in an APPC conversation with a local transaction program.

PLU See *primary logical unit* and *partner logical unit*.

port An endpoint for communication between devices, generally referring to a logical connection. A 16-bit number identifying a particular Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) resource within a given TCP/IP node.

port sharing
A way of load balancing TCP/IP connections across a group of servers running in the same z/OS image.

prepare phase
The first phase of a XA process in which all participants are requested to confirm readiness to commit.

presentation logic
The part of a distributed application that is concerned with the user interface of the application. Compare with *business logic*.

primary logical unit (PLU)
In SNA, the logical unit that contains the primary half-session for a particular logical unit-to-logical unit (LU-to-LU) session. See also *secondary logical unit*.

protocol boundary
The signals and rules governing interactions between two components within a node.

Q

Query strings
Query strings are used in the statistical data API. A query string is an input parameter, specifying the statistical data to be retrieved.

R

RACF See *Resource Access Control Facility*.

realm A named collection of users and groups that can be used in a specific security context. See also *distinguished name* and *identity propagation*.

Recoverable resource management services (RRMS)
The registration services, context services, and resource recovery services provided by the z/OS sync point manager that enable consistent changes to be made to multiple protected resources.

Resource Access Control Facility (RACF)
An IBM licensed program that provides access control by identifying

users to the system; verifying users of the system; authorizing access to protected resources; logging detected unauthorized attempts to enter the system; and logging detected accesses to protected resources.

region In workload management on CICS Transaction Gateway for Windows, an instance of a CICS server.

remote mode
Remote mode describes the use of one of the supported CICS Transaction Gateway network protocols to connect to the Gateway daemon.

remote procedure call (RPC)
A protocol that allows a program on a client computer to run a program on a server.

Request monitoring exits
Exits that provide information about individual requests as they are processed by the CICS Transaction Gateway.

request unit (RU)
In SNA, a message unit that contains control information such as a request code, or function management (FM) headers, end-user data, or both.

request/response unit
A generic term for a request unit or a response unit. See also *request unit* and *response unit*.

response file
A file that contains predefined values that is used instead of someone having to enter those values one at a time. See also *CID methodology*.

response unit (RU)
A message unit that acknowledges a request unit; it can contain prefix information received in a request unit.

Resource adapter
A system-level software driver that is used by an EJB container or an application client to connect to an enterprise information system (EIS). A resource adapter plugs in to a container; the application components deployed on the container then use the client API (exposed by adapter) or tool-generated, high-level abstractions to access the underlying EIS.

resource group ID
A resource group ID is a logical grouping of resources, grouped for statistical purposes. A resource group ID is associated with a number of resource group statistics, each identified by a statistic ID.

resource ID
A resource ID refers to a specific resource. Information about the resource is included in resource-specific statistics. Each statistic is identified by a statistic ID.

resource manager
The participant in a transaction responsible for controlling access to recoverable resources. In terms of the CICS resource adapters this is represented by an instance of a ConnectionFactory.

Resource Recovery Services (RRS)
A z/OS facility that provides two-phase sync point support across participating resource managers.

Result set
A result set is a set of data calculated or recorded by a statistical API function.

Result set token
A result set token is a reference to the set of results returned by a statistical API function.

rollback
An operation in a transaction that reverses all the changes made during the unit of work. After the operation is complete, the unit of work is finished. Also known as a backout.

RU See *Request unit* and *Response unit*.

RPC See *remote procedure call*.

RRMS
See *Recoverable resource management services*.

RRS See *Resource Recovery Services*.

S

SBCS See *single-byte character set*.

secondary logical unit (SLU)
In SNA, the logical unit (LU) that contains the secondary half-session for a particular LU-LU session. Contrast with primary logical unit. See also *logical unit*.

Secure Sockets Layer (SSL)
A security protocol that provides communication privacy. SSL enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. SSL applies only to internet protocols, and is not applicable to SNA.

server name remapping
See *dynamic server selection*.

servlet
A Java program that runs on a Web server and extends the server's functionality by generating dynamic content in response to Web client requests. Servlets are commonly used to connect databases to the Web.

session limit
In SNA, the maximum number of concurrently active logical unit to logical unit (LU-to-LU) sessions that a particular logical unit (LU) can support.

silent installation
Installation that does not display messages or windows during its progress. Silent installation is not a synonym of "unattended installation", although it is often improperly used as such.

single-byte character set (SBCS)
A character set in which each character is represented by 1 byte. Contrast with double-byte character set.

sign-on capable terminal
A sign-on capable terminal allows sign-on transactions, either CICS-supplied (CESN) or user-written, to be run. Contrast with sign-on incapable terminal.

SIT See *system initialization table*.

SLU See *secondary logical unit*.

SMIT See *System Management Interface Tool*.

SNA See *Systems Network Architecture*.

SNA sense data
An SNA-defined encoding of error information In SNA, the data sent with a negative response, indicating the reason for the response.

SNASVCMG mode name
The SNA service manager mode name. This is the architecturally-defined mode name identifying sessions on which CNOS is exchanged. Most APPC-providing products predefine SNASVCMG sessions.

socket A network communication concept, typically representing a point of connection between a client and a server. A TCP/IP socket will normally combine a host name or IP address, and a port number.

SSL See *Secure Sockets Layer*.

SSLight
An implementation of SSL, written in Java, and no longer supported by CICS Transaction Gateway.

statistic data
A statistic data structure holds individual statistical result returned after calling a statistical API function.

statistic group
A generic term for a collection of statistic IDs.

statistic ID
A label referring to a specific statistic. A statistic ID is used to retrieve specific statistical data, and always has a direct relationship with a statistic group.

standard error
In many workstation-based operating systems, the output stream to which error messages or diagnostic messages are sent.

subnet
An interconnected, but independent segment of a network that is identified by its Internet Protocol (IP) address.

subnet address
In Internet communications, an extension to the basic IP addressing scheme where a portion of the host address is interpreted as the local network address.

sync point
Synchronization point. During transaction processing, a reference point to which protected resources can be restored if a failure occurs.

SYNCONRETURN
A request where the CICS server takes a sync point on successful completion of the server program. Changes to recoverable resources made by the server program are committed or rolled-back independently of changes to recoverable resources made by the client program issuing the ECI request, or changes made by the server in any subsequent ECI request. Also referred to as a *nonextended logical unit of work*.

system initialization table (SIT)
A table containing parameters used to start a CICS control region.

System Management Command
An administrative request received by a Gateway daemon (or Gateway daemon address space on z/OS) from the **ctgadmin** command (on UNIX, Linux, or Windows) or the z/OS console. The request might be made to retrieve information about the Gateway daemon, or to alter some aspect of Gateway daemon behavior.

Typically, a **ctgadmin** command in the form **ctgadmin** <command string> is entered by an operator using the command line interface, or a modify command in the form /F <job name>,APPL=<command string> is entered by an operator on the z/OS console.

System Management Interface Tool (SMIT)
An interface tool of the AIX® operating system for installing, maintaining, configuring, and diagnosing tasks.

Systems Network Architecture (SNA)
An architecture that describes the logical structure, formats, protocols, and operational sequences for transmitting information units through the networks and also the operational sequences for controlling the configuration and operation of networks.

System SSL
An implementation of SSL, no longer supported by CICS Transaction Gateway on z/OS.

T

TCP/IP
See *Transmission Control Protocol/Internet Protocol*.

TCP/IP load balancing
The ability to distribute TCP/IP connections across target servers.

terminal emulation
The capability of a personal computer to operate as if it were a particular type of terminal linked to a processing unit and to access data. See also *emulator, emulation program*.

thread A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

timeout
A time interval that is allotted for an event to occur or complete before operation is interrupted.

TLS See *Transport Layer Security*.

token-ring network
A local area network that connects devices in a ring topology and allows unidirectional data transmission between devices by a token-passing procedure. A device must receive a token before it can transmit data.

trace A record of the processing of a computer program. It exhibits the sequences in which the instructions were processed.

transaction manager
A software unit that coordinates the activities of resource managers by managing global transactions and coordinating the decision to commit them or roll them back.

transaction program
A program that uses the Advanced Program-to-Program Communications (APPC) application programming interface (API) to communicate with a partner application program on a remote system.

Transmission Control Protocol/Internet Protocol (TCP/IP)
An industry-standard, nonproprietary set of communications protocols that provide reliable end-to-end connections between applications over interconnected networks of different types.

Transport Layer Security (TLS)
A security protocol that provides communication privacy. TLS enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. TLS applies only to internet protocols, and is not applicable to SNA. TLS is also known as SSL 3.1.

two-phase commit
A protocol with both a prepare and a commit phase, that is used for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction.

type 2.0 node
A node that attaches to a subarea network as a peripheral node and provides a range of end-user services but no intermediate routing services.

type 2.1 node
An SNA node that can be configured as an endpoint or intermediate routing node in a network, or as a peripheral node attached to a subarea network.

U

unattended installation
Unattended installation is installation performed without user interaction during its progress, or, with no user present at all, except for the initial launch of the process. -

Uniform Resource Locator (URL)
A sequence of characters that represent information resources on a computer or in a network such as the Internet. This sequence of characters includes (a) the abbreviated name of the protocol used to access the information resource and (b) the information used by the protocol to locate the information resource.

unit of recovery (UR)
A defined package of work to be performed by the RRS.

unit of work (UOW)
A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or at a user-requested sync point. It ends either at a user-requested sync point or at the end of a transaction.

UOW See *unit of work*.

UR See *unit of recovery*.

URL See *Uniform Resource Locator*.

user registry
The location where the distinguished name of a user is defined and authenticated. See also *distinguished name*.

user session
Any APPC session other than a SNASVCMG session.

V

verb A reserved word that expresses an action to be taken by an application programming interface (API), a compiler, or an object program.

In SNA, the general name for a transaction program's request for communication services.

version string
A character string containing version information about the statistical data API.

W

WAN See *wide area network*.

Web browser
A software program that sends requests to a Web server and displays the information that the server returns.

Web server
A software program that responds to information requests generated by Web browsers.

wide area network (WAN)
A network that provides communication services to a geographic area larger than that served by a local area network or a metropolitan area network, and that can use or provide public communication facilities.

wrapping trace
On Windows, UNIX, and Linux, a configuration in which the **Maximum Client wrap size** setting is greater than 0. The total size of Client daemon binary trace files is limited to the value specified in the **Maximum Client wrap size** setting. With standard I/O tracing, two files, called `cicscli.bin` and `cicscli.wrp`, are used; each can be up to half the size of the **Maximum Client wrap size**.

X

XA requests
An XA request is any request sent or received by the CICS Transaction Gateway in support of an XA transaction. These requests include the XA commands commit, complete, end, forget, prepare, recover, rollback, and start.

XA transaction
A global transaction that adheres to the X/Open standard for distributed transaction processing (DTP.)

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Sending your comments to IBM

If you especially like or dislike anything about this book, use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Limit your comments to the information in this book and the way in which the information is presented.

To ask questions, make comments about the functions of IBM products or systems, or to request additional publications, contact your IBM representative or your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - +44 1962 842327 (if you are outside the UK)
  - 01962 842327 (if you are in the UK)
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Product Number: 5655-W10

SC34-7059-02

Spine information:

CICS Transaction Gateway

CICS Transaction Gateway for z/OS: Programming Guide

Version 8.0