Using iPlanet UDS for OS/390

iPlanet™ Unified Development Server

Version 5.0

August 2001

Contents

List of Figures

List of Tables

List of Procedures

Preface

This book describes iPlanet UDS for OS/390, which provides support for the deployment and management of iPlanet UDS server partitions on the OS/390 platform.

This preface contains the following sections:

Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

Audience for This Guide

Using iPlanet UDS for OS/390 is written for programmers who are fully conversant with OS/390-hosted COBOL applications and familiar with the iPlanet UDS environment. We assume that you:

- have TOOL programming experience
- are familiar with your particular window system
- understand the basic concepts of object-oriented programming as described in *A Guide to the iPlanet UDS Workshops*
- have used the iPlanet UDS workshops to create classes

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	**Description**
Chapter 1, "iPlanet UDS for OS/390"	Explains how you use the OS/390 system as a deployment environment for iPlanet UDS applications.
Chapter 2, "Overview of the iPlanet UDS Transaction Adapter"	Provides an overview of the iPlanet UDS Transaction Adapter for OS/390.
Chapter 3, "Using the iPlanet UDS Transaction Adapter"	Explains how you use the iPlanet UDS Transaction Adapter for OS/390 in developing an application.
Chapter 4, "Integrating IBM OS/390-Hosted COBOL Applications with iPlanet UDS"	Explains how to use Fscript extensions to create Transaction Adapters for connecting to OS/390-hosted COBOL programs.
Appendix A, "Example Applications"	Describes sample applications that demonstrate the use of the iPlanet UDS Transaction Adapter for OS/390 to interface with various transaction processing monitors.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ). Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, iPlanet UDS provides additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read "Viewing and Searching PDF Files" on page 20 to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at http://docs.iplanet.com/docs/manuals/uds.html.

The titles of the iPlanet UDS documentation are listed in the following sections.

iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Guide*
- *Fscript Reference Guide*
- *Getting Started With iPlanet UDS*
- *Integrating with External Systems*
- *iPlanet UDS Java Interoperability Guide*
- *iPlanet UDS Programming Guide*
- *iPlanet UDS System Installation Guide*
- *iPlanet UDS System Management Guide*
- *Programming with System Agents*
- *TOOL Reference Guide*
- *Using iPlanet UDS for OS/390*

Express Documentation

- *A Guide to Express*
- *Customizing Express Applications*
- *Express Installation Guide*

WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution: `FORTE_ROOT/userapp/forte/cl`*`n`*`/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help `<`*`command`*`>` for help on a specific command.

iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ To copy the documentation to a client or server

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ To view and search the documentation

1. Open the file `udsdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.

3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

   NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	**Keyboard Command**
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

Chapter 1

iPlanet UDS for OS/390

This chapter describes iPlanet UDS for OS/390, an iPlanet UDS release that provides support for the deployment of iPlanet UDS server partitions on the OS/390 platform. This chapter also supplements the documentation set provided by iPlanet UDS with information specific to the OS/390 deployment environment. This includes the following:

- integrating with C applications
- calling native MVS programs from iPlanet UDS
- setting security privileges
- performance tuning
- DB2 access
- workload management

For information on installing iPlanet UDS for OS/390, see the *iPlanet UDS System Installation Guide*.

iPlanet UDS for OS/390

Starting with release 3J of iPlanet UDS, you will be able to use the OS/390 system as a deployment environment for iPlanet UDS applications. This means that an iPlanet UDS partition installed as part of UNIX System Services on the OS/390 platform can communicate with any other iPlanet UDS partition on a platform that is running iPlanet UDS release 3.0.J or higher.

An iPlanet UDS *environment* consists of a collection of networked nodes running the iPlanet UDS runtime system. This system provides a common set of services that allows an iPlanet UDS application, or a part of such an application (a partition) to run on a number of platforms. The runtime system supports access to the local operating system as well as communication and synchronization between partitions running on other platforms.

An iPlanet UDS *deployment environment* is one in which you can deploy, run, and manage an iPlanet UDS application. A deployment environment has the iPlanet UDS runtime system and system management services installed and running and includes the Escript environment console utility, which is used to deploy, start, and manage distributed applications.

A node in an iPlanet UDS environment can be a server node, a client node, or both. iPlanet UDS for OS/390 only supports server nodes. You can deploy only server partitions on a server node. An *iPlanet UDS for OS/390 partition* refers to an iPlanet UDS server partition that is executing in the UNIX System Services environment on an OS/390 platform.

An iPlanet UDS OS/390 partition has the following characteristics:

- It can run in interpreted or compiled mode.
- It can be configured for failover and load balancing.
- It can transparently communicate with iPlanet UDS partitions deployed on other platforms that are running iPlanet UDS version 3.0.J or higher.
- It can be managed using iPlanet UDS's system management facilities.

In addition, an iPlanet UDS for OS/390 partition can be integrated with non-iPlanet UDS code. Methods in an iPlanet UDS for OS/390 partition can call encapsulated C functions residing in C projects on any iPlanet UDS partition. By way of this mechanism, iPlanet UDS applications can also call native MVS programs.

"Integrating With External Systems" on page 30 provides additional information that you need to perform this integration.

Feature Restrictions

This release of iPlanet UDS for OS/390 does not allow iPlanet UDS partitions to do the following:

- read input from or send output to terminals or printers on the OS/390 system
- run the iPlanet UDS development workshops on OS/390. (Development and distribution of iPlanet UDS applications is only possible on OS/390 using the Fscript command-line utility)
- make calls to the Display library
- make C++ callins
- build and test applications that use the display system from an OS/390 platform used as a shared development platform

Accessing DB2/MVS

iPlanet UDS DB2 Adapter for OS/390 uses the DB2 Call Level Interface (CLI) to connect directly to DB2 in cross-memory mode on the OS/390 platform. Therefore, DB2 DBSession objects must always be instantiated on OS/390 partitions.

To enable CLI use by iPlanet UDS DB2 Adapter, you must GRANT EXECUTE privileges to the iPlanet UDS for OS/390 user ID that starts the node manager. See "Security Privileges" on page 28 for more information.

DB2/MVS DBSession objects require more virtual memory than allowed by the default memory settings on OS/390. Partitions that use OS/390 DB2 require an initial heap size of 32 MB or larger. You can use either the FORTE_GC_SPECIAL environment variable or the `-fm` parameter to override default values. See "Virtual Memory" on page 37 for more information.

CAUTION Note that there are significant differences between DB2/MVS SQL syntax and DB2 SQL syntax on other platforms. For example, DB2/MVS uses different date and time formats than DB2 on other platforms, and supports shorter BLOB data items. DB2/MVS limits BLOB data items to less then 32K bytes in length. Actual BLOB size is further limited by the size of other column types in the table and by the buffer pools assigned when the tablespace was created. See DB2 for OS/390 V5 SQL Reference (SC26-8966) and DB2 for OS/390 Administration Guide (SC26-8957) for additional information.

DB2 CLI Initialization Dataset

IBM's DB2 on OS/390 allows multiple database servers for each instance of DB2 running on OS/390. Each DB2 instance is identified by the OS/390 system programmer with a
4-character subsystem name. Each DB2 subsystem may, in turn, access multiple database servers. The DB2 CLI interface reads an initialization file that identifies which DB2 instance and which database server within that instance can be accessed. The DB2 instance is selected based on the subsystem ID specified in this file; the database server is selected based on the server name, also specified in the file.

The DSNAOINI environment variable identifies the name of the dataset containing the specific DB2 CLI initialization parameters you want to use, and hence the DB2 instance and database server to be accessed. You specify the value for the DSNAOINI environment variable to the iPlanet UDS installer when it prompts for the DSNAOINI dataset name.

For example, the following response specifies the MVS dataset containing the DB2 CLI initialization parameters to be the IBM-supplied PDS member shipped with DB2 5.1.0. Note the use of quotation marks; these are required to avoid UNIX System Services treating the parenthesis as special characters. The installer accepts the entire string and assigns it including the quotation marks.

```
"DSN510.SDSNSAMP(DSNAOINI)"
```

The following response specifies a customer-named sequential dataset. The value for <qualifier> is typically the user id starting the iPlanet UDS for OS/390 node manager. In this case, there are no special characters in the string. The quotation marks are not required, but they are accepted if you decide to put them in.

```
<qualifier>.DSNAOINI
```

The installer uses your responses to set the fortedef.sh environment variable DSNAOINI.

RACF authorization to read the dataset must be given to the user id running iPlanet UDS and starting the iPlanet UDS for OS/390 node manager.

Sample Dataset

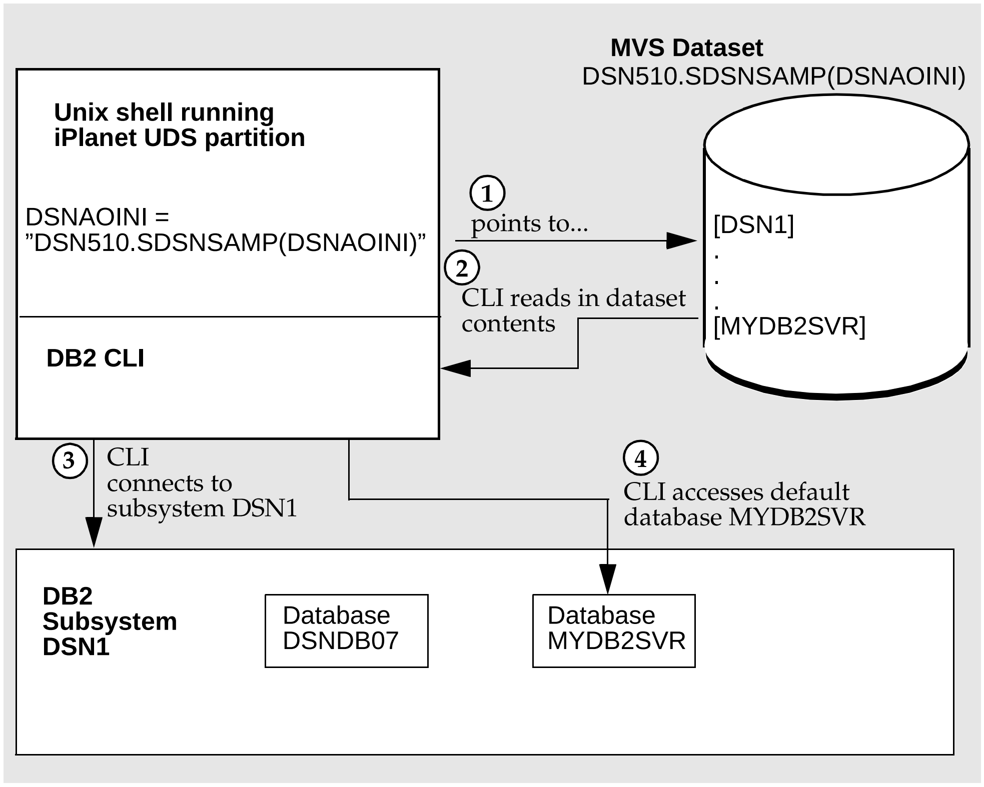
The following DB2 CLI initialization dataset defines the DSN1 instance of DB2 and the MYDB2SVR database server to be accessed. The user id that starts the iPlanet UDS for OS/390 node manager must have RACF privileges to read this dataset and also have DB2 privileges to execute the DSNACLI plan and the DSNCLI package under the MYDB2SVR database server within the DNS1 instance.

```
; This is a comment line...
; Example COMMON stanza
[COMMON]
MVSDEFAULTSSID=DSN1

; Example SUBSYSTEM stanza for DSN1 subsystem
[DSN1]
MVSATTACHTYPE=CAF
PLANNAME=DSNACLI

; Example DATA SOURCE stanza for MYDB2SVR data source
; where MYDB2SVR is one of the database servers under
; the DSN1 DB2 subsystem
[MYDB2SVR]
AUTOCOMMIT=0
CONNECTTYPE=1
CURSORHOLD=1
```

You can find an IBM-supplied example of this initialization file in the IBM DB2 install dataset member *.SDSNSAMP(DSNAOINI).

Figure 1-1 shows the process we have described so far.

Figure 1-1 DB2 Initialization Dataset


MVS Dataset
DSN510.SDSNSAMP(DSNAOINI)
Unix shell running iPlanet UDS partition
DSNAOINI = ”DSN510.SDSNSAMP(DSNAOINI)”
DB2 CLI
1 points to...
2 CLI reads in dataset contents
[DSN1]
.
.
.
[MYDB2SVR]
3 CLI connects to subsystem DSN1
4 CLI accesses default database MYDB2SVR
DB2 Subsystem DSN1
Database DSNDB07
Database MYDB2SVR

Security Privileges

Security privileges for iPlanet UDS for OS/390 partitions are controlled by the OS/390 Security Access Facility (SAF) that is running on that platform. The IBM Resource Access Control Facility (RACF) is one SAF-compliant security package; other such packages are available from vendors other than IBM. The setting of security privileges for iPlanet UDS processes running on the OS/390 platforms does not depend on the SAF-compliant package being used.

On the OS/390 platform, the privileges for an iPlanet UDS partition are the same as the privileges granted to the MVS user ID that starts the iPlanet UDS partition. Whoever starts the UNIX process executing for the partition determines the security privileges associated with that process.

For example, if user ID “sysadm” starts an iPlanet UDS node manager, the node manager will have all SAF privileges associated with the MVS “SYSADM” user ID. Furthermore, if this node manager autostarts a partition, then the started partition will also have SYSADM authority. If a partition is not autostarted but another user ID, for example, “bryanp” executes the `ftexec` command to start it, then that partition runs with “BRYANP” privileges.

This scheme affects a number of areas, principally file and database access. For example, if the partition is going to read and write to log files, repositories, or any other type of file, then the user ID starting the partition must have appropriate read and write privileges to the files that are accessed.

This scheme is enforced even if iPlanet UDS applications propagate a user ID to server partitions running on the OS/390 platform. iPlanet UDS will not use a client’s user ID to override or change any of the security privileges established for the server partition. iPlanet UDS applications can certainly take advantage of a client’s user ID to impose additional security measures, but these additional measures must be enforced programmatically by the iPlanet UDS application.

DB2 Access

The OS/390 user ID that starts the iPlanet UDS for OS/390 node manager under OS/390 Unix System Services (USS) is the user ID that must be authorized by RACF to allow access to DB2 on OS/390.

When DB2 CLI establishes a connection to DB2 from an iPlanet UDS for OS/390 partition on behalf of an iPlanet UDS client, the user ID that must be authorized to access DB2 is *not* the user ID of the client on the requesting node, but the user ID which started the node manager on USS. In other words, the user ID and password set for a DBSession object are not used to establish security privileges for DB2 access.

GRANT EXECUTE Procedure for DB2

Before using iPlanet UDS DBSession objects to access DB2, your database administrator must grant certain stored procedure privileges to the iPlanet UDS OS/390 partitions to enable use of the DB2 Call Level Interface. This procedure is documented in IBM manual *DB2 for OS/390 V5 Call Level Interface Guide* (SC26-8959). The JCL to perform the GRANTs can be found in DSN510.SDSNSAMP, member DSNTIJCL. If you cannot find this sample member, see IBM APAR PQ07001. PTF UQ08548 provides the sample JCL. See Tech Note Vantive 11598 for more information.

After doing the BINDs using the sample JCL, the following GRANTs must be run for DSNACLI as well as for any user who is to use DB2 CLI:

```
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLICS TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIC1 TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIC2 TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIF4 TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIMS TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLINC TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIQR TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIRR TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIRS TO <userid>;
GRANT EXECUTE ON PACKAGE DSNAOCLI.DSNCLIUR TO <userid>;
```

In the sample code above, <userid> is the user ID used to start the node manager. It is quite likely that the above list of GRANTs must be done several times, once for each user ID that might be starting a node manager, which, in turn, will start iPlanet UDS for OS/390 with DB2 Adapter partitions. These user IDs could easily be different for different purposes: for example, 'forte', 'frteprod', 'frtetest'.

Currently, the only way to associate different DB2/RACF controls with different users is for different node managers to be started by different user IDs (with corresponding different $FORTE_NS_ADDRESS values set) and have each iPlanet UDS client use the iPlanet UDS for OS/390 node manager with the appropriate authorizations.

Integrating With External Systems

An iPlanet UDS partition can call C functions. The process for this integration is documented in *Integrating with External Systems*. The following sections offer additional information that you need if you are using iPlanet UDS for OS/390 and need to do the following:

- integrate with C code
- monitor operating system activities
- use the ExternalConnection class

Calling External C Functions

The following list is a summary of the steps required to call C functions from an iPlanet UDS application. Any information that you need to perform these steps for an iPlanet UDS OS/390 partition is noted for each step. For detailed information on completing this process, see *Integrating with External Systems*.

1. Package your C functions in a C shared library.

   The OS/390 platform requires that C object modules be shared libraries that are compiled to be position-independent. The flag to the C89 and C++ compilers that produces position-independent code is:

   ```
   -W c, EXPORTALL
   ```

2. Create a C project definition file that maps C function names to class method names.

   When you define the `externalsharedlibs` property for this project, note that for OS/390 the library name should be the export file (sidedeck) name.

3. Import the C project definition into the Repository Workshop.
4. Partition the C project as a library.
5. Make a distribution to generate the shared library files you need and to install these on the nodes where you will need them.
6. Add the C project for the C functions as a supplier project for your TOOL project.
7. Write the TOOL application that uses the C functions.

   In this application, you instantiate the classes you defined in the C project definition file. Calling a method for such a class, invokes the C function that has the same name as the method.

8. Test, partition, and deploy your application.

Calling Native MVS Programs from iPlanet UDS

This section explains how you can call native MVS programs from iPlanet UDS. This technique involves the use of external C functions as an intermediary step, so you need to familiarize yourself with the process of calling external C functions from iPlanet UDS before you can implement this process.

An iPlanet UDS for OS/390 application can be linked with native MVS load modules to provide access to native MVS services. You can use this technique only to perform static links to MVS load modules; you cannot use this technique to link with MVS object modules nor to provide automatic support for dynamic linkage as provided by some MVS compilers.

The technique described in this section takes advantage of the MVS linkage editor automatic call feature to link in native MVS code.

MVS Program Requirements

The native MVS program must adhere to the following requirements:

- It must be reentrant.
- It must run in 31-bit addressing mode (AMODE 31).
- It must be able to reside above the 16-megabyte line in an MVS address space (RMODE ANY).

Integration Procedure

This section describes the procedure used to call a native MVS program from iPlanet UDS for OS/390 and offers a simple example of such a call.

➤ **To call a native MVS program**

1. Write a C program in which you declare your native MVS program functions.

   If your native MVS module is written in System/390 assembler and obeys standard linkage conventions, you will need to add the following directive:

   ```
   #pragma linkage(<function_name>, OS)
   ```

   The function_name parameter is the name of the native MVS load module entry point that you want to call.

2. Compile and link your native MVS program(s) into a standard MVS load library.

   The name of the load module (or one of its aliases) must be the same as the function name declared in your external C program.

3. Execute the following UNIX System Services export command in your environment:

   ```
   export FORTE_SYSLIBS="-l //<'MY.MVS.LOADLIB.NAME'>"
   ```

   The value you specify for this variable must contain valid parameters and parameter values for the UNIX c89 utility. See the UNIX System Services c89 man page for additional information.

   If you need to search multiple MVS load libraries, you can add multiple "-l" directives and library names in the $FORTE_SYSLIBS environmental variable.

   You can place this export command in your fortedef shell script if you wish to make sure that all iPlanet UDS application links will be able to link in the native MVS load module whenever necessary.

4. Integrate the external C program with your iPlanet UDS application as described in "Calling External C Functions" on page 31.

   The iPlanet UDS compcomp script will pick up the $FORTE_SYSLIBS environmental variable value and append it to the end of the c89 link command, allowing the linkage editor to search the load library for the required native MVS load module.

For example, to link the System/390 Assembler routine IEFBR14 into your external C application, include the following statement in the C source code:

```
#pragma linkage (IEFBR14, OS)
```

Set the $FORTE_SYSLIBS environmental variable to search the MVS SYS1.LINKLIB dataset:

```
export FORTE_SYSLIBS="-l //'SYS.LINKLIB'"
```

Use the standard iPlanet UDS facilities to link your external C library. The linkage editor automatic call feature will automatically find SYS1.LINKLIB(IEFBR14) and include it in the output shared library.

Using System Activities

iPlanet UDS provides the Rendezvous, SystemActivity, and Activity Manager classes to enable iPlanet UDS applications or wrapped C code to register for certain low-level operating system events or *system activities*. An iPlanet UDS partition running on the OS/390 platform can also use these classes to register for system events; once it does, it will be notified when the specified activity has occurred or completed. For complete information about the system activities that are supported, see *Integrating with External Systems*.

Using the External Connection Class

iPlanet UDS's ExternalConnection class facilitates network communications between two points when one of the points is an iPlanet UDS application. Using this class, an iPlanet UDS application can communicate with an external process or program that is running locally or on another host. For complete information, see *Integrating with External Systems*.

The following table shows the network transport protocols supported by the ExternalConnection class. Note that TCP/IP connections and UnixDomainSockets connections are both supported for the OS/390 platform:

Type of Connection	From	To
TCP/IP Sockets (BSD or Winsock)	Mac	any reliable TCP/IP entity
	Windows	
	NT	
	Digital Unix (Alpha OSF)	
	HP/UX	
	AIX	
	DG/UX	
	OS/390	
TCP/IP TLI Endpoint (System V)	Dynix PTX	any reliable TCP/IP entity
	Solaris	
UnixDomainSockets (UDS)	Digital Unix	any other process, running on the same machine, that can read and write UDS
	HP/UX	
	AIX	
	DG/UX	
	OS/390	
DECnet	Windows (Pathworks)	any DECnet entity
	VMS	
	Macintosh (not PowerMac)	

Performance Tuning

The following sections provide information that you might need to improve the performance of iPlanet UDS for OS/390 partitions. The areas addressed include the setting of environment variables to support a large number of users and the use of runtime options to allocate the virtual and real memory required by the iPlanet UDS runtime product.

Supporting Large Number of Users

Default values for MVS system parameters and Language Environment 370 (LE/370) runtime environment options limit the number of POSIX threads that can be created for an iPlanet UDS process. These defaults prevent iPlanet UDS for OS/390 from creating more than 50 concurrent threads. Even though users share threads, iPlanet UDS for OS/390 partitions that support large numbers of users might need to create a larger number of POSIX threads. This section explains how you can change the default value for POSIX threads and how to set runtime options so that a large number of threads can be created.

To begin, an MVS system programmer must set the MAXTHREADS parameter for the SYS1.PARMLIB data set BPXPRM*xx* member to a number that is larger than 50.

Next, you must reset several LE/370 runtime options to allow an iPlanet UDS for OS/390 process to create a larger number of threads. The sample settings shown in the table below allow an iPlanet UDS partition to create several hundred POSIX threads.

Keyword	**Value string**
all31	on
stack	48k, 0k, anywhere, free
heap	6m, 4m, anywhere, keep, 1k, 1k
anyheap	4m, 4m, anywhere, free
belowheap	256k, 256k, free

Each POSIX thread is an MVS subtask for which the system allocates a Task Control Block (TCB) and other private data areas below the 16-megabyte line. Because memory is limited below this line, you should use the options specified in the previous table to minimize memory allocation in this area.

One easy way to set these options at runtime is to define the $_CEE_RUNOPTS environmental variable as follows:

```
export _CEE_RUNOPTS ="all31(on) stack(48k,0k,anywhere,free)
heap(6m,4m,anywhere,keep,1k,1k) anyheap(4m,4m,anywhere,free)
belowheap(256k,256k,free)”
```

The $_CEE_RUNOPTS variable is set to these values by the fortedef.sh shell script, created when you installed iPlanet UDS on your system.

For additional information on LE/370 runtime options, see IBM manual number SC28-1940, *OS/390 Language Environment for OS/390 and VM Programming Reference*, Chapter 2.

Memory Allocation

This section describes iPlanet UDS’s use of real and virtual memory and explains how you can use LE/370 runtime options to allocate more virtual and real storage for iPlanet UDS for OS/390 partitions.

When iPlanet UDS for OS/390 starts up, it allocates a large heap area above the 16MB line, then uses this memory area for runtime operations. iPlanet UDS for OS/390 partitions provide their own memory management. For information on how you can fine-tune iPlanet UDS’s own memory management scheme, see *iPlanet UDS System Management Guide*.

The default settings for the iPlanet UDS memory manager on OS/390 are shown in the table below.

Keyword value	Description
n:4096	initial heap size is 4 megabytes
n:32768	if using DB2
i:1024	additional heap allocations are rounded up to 1 megabyte increments
i:4096	if using DB2
x:16384	maximum heap memory allowed is 16 megabytes
x: 65536	if using DB2

Virtual Memory

The iPlanet UDS runtime product can require significant amounts of virtual memory, depending on the size of the application, the number of concurrent users, and the operations requested by these users.

- iPlanet UDS for OS/390 executable binaries and dynamic libraries typically require 2MB - 4MB of virtual memory each.
- iPlanet UDS for OS/390 executables typically allocate 4MB of virtual memory for runtime operations.
- Each POSIX thread that is created to support user connections allocates some memory, depending on parameters established for the IBM LE/370 runtime environment. For more information, see "Supporting Large Number of Users" on page 35.

Note that these are *virtual* memory requirements. For information about real storage requirements, see "Real Storage Requirements" on page 39. For detailed information about how you change iPlanet UDS memory management values, see the *iPlanet UDS System Management Guide*.

You can often just use LE/370 runtime options to adjust virtual memory allocation. The default values set by the fortedef.sh shell script should be optimal for many applications. The use of LE/370 runtime options is documented in Chapter 2 of IBM manual SC28-1940, *OS/390 Language Environment for OS/390 and VM Programming Reference*.

iPlanet UDS parameters and MVS operating system configuration parameters can limit the amount of virtual memory available to iPlanet UDS for OS/390 partitions, leading to memory allocation failures. These are indicated by error messages like the following:

```
FATAL ERROR: Out of Memory (reason = qqOS_MM_EX_OUT_OF_MEMORY)
Class: qqsp_ResourceException with ReasonCode: SP_ER_OUTOFMEMORY
```

If you encounter such errors, you need to do the following:

- Increase the maximum amount of memory available to the iPlanet UDS partition by running the partition with the -fm parameter or by setting the $FORTE_GC_SPECIAL environment variable to increase the initial heap allocation. For DB2 to run correctly, the $FORTE_GC_SPECIAL value should provide an initial heap size of 32MB. The $FORTE_GC_SPECIAL value for partitions on OS/390 that create DB2 DBSessions should be n: 32768, x:65536.

For example, to allocate 12 MB, use n:12288. You must also increase the LE/370 heap size by changing the $_CEE_RUNOPTS environment variable value. The initial LE/370 heap size must be at least 40 bytes larger than the initial iPlanet UDS memory manager heap size. In the following example, the initial heap size is set to 13 MB, 1 MB larger than the amount allocated for the iPlanet UDS partition:

```
export _CEE_RUNOPTS ="all31(on) stack(48k,0k,anywhere,free)
heap(13m,4m,anywhere,keep,1k,1k) anyheap(4m,4m,anywhere,free)
belowheap(256k,256k,free)”
```

- Increase the MAXASSIZE parameter value in SYS1.PARMLIB, member BPXPRM*xx*.
- Check SYS1.PARMLIB, member SMFPRM*xx* to see if an IEFUSI user exit is installed. If it is, examine the assembler source code for the exit to see if the exit enforces artificially low memory allocation constraints above the 16 MB line. You can either disable the exit or change the constraints enforced by the exit.

If you get the following error message:

```
FATAL ERROR: Out of Memory (reason =
qqOS_MM_EX_OUT_OF_CONTIG_MEMORY)
Class: qqsp_ResourceException with ReasonCode: SP_ER_OUTOFMEMORY
```

Change the $_CEE_RUNOPTS environment variable setting to increase the heap size. The heap extension size must be larger than the object you are trying to allocate. For example, if your error message indicates that you’re allocating an 18 MB object, then you might change the heap extension size to 20 MB, as shown in the following code sample

```
export _CEE_RUNOPTS ="all31(on) stack(48k,0k,anywhere,free)
heap(6m,20m,anywhere,keep,1k,1k) anyheap(4m,4m,anywhere,free)
belowheap(256k,256k,free)”
```

Note that when you change a single $_CEE_RUNOPTS value, you must include all runtime parameter settings.

You might also need to increase the maximum amount of memory available to the iPlanet UDS partition by running partitions with the -fm parameter or by setting the $FORTE_GC_SPECIAL environment variable. As in the example above, you would need to set the x value to at least 22528 (22 MB) to support an initial heap size of 4 MB plus an
18-MB object.

Real Storage Requirements

The amount of real memory used by each iPlanet UDS for OS/390 process depends on usage patterns and the demand for real storage in the OS/390 environment.

iPlanet UDS recommends that you put frequently used LE/370 runtime modules into the MVS Link Pack Areas (LPA) so that iPlanet UDS processes and all users of UNIX Services share copies of these modules. Specifically, you should always place module CEEBINIT (alias CEEBLIBM) into LPA.

IBM provides a number of additional tuning recommendations on their World Wide Web site. See *http://www.s390.ibm.com/products/oe/* for more detailed information.

Workload Management

This section describes the administrative setup and user actions required to manage iPlanet UDS workloads with the OS/390 Workload Manager (WLM). Using the procedures described in the following sections, you can

- distinguish iPlanet UDS work from other UNIX work running on OS/390
- assign different priorities to iPlanet UDS production and test workloads
- prioritize a favored server partition over other iPlanet UDS partitions

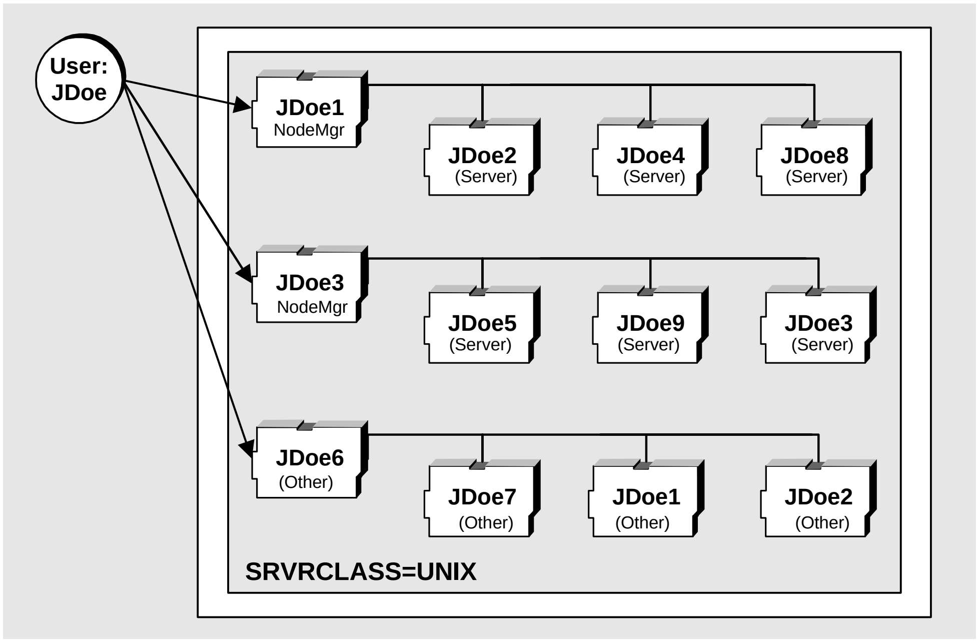
Obtaining this kind of workload management support does not require any changes at the source code level.

Overview

Workload management is a means of allocating system resources (cpu time, memory, I/O) to different processers or groups of processes. To do this, the Workload Manage, a system component, treats everything on the system as a *workload* or *service class*: for example, one service class could consist of all UNIX services, online work might comprise another service class, and CICS processing, another. The OS/390 system administrator defines policies that associate performance goals with each service class, and the allocation of MVS system resources varies as each workload meets or fails to meet the goals that are set for it.

By default, UNIX system service work is lumped together in one service class. Figure 1-2 shows how iPlanet UDS partitions are treated by default by the workload manager: all iPlanet UDS partitions, started directly or indirectly by userid JDoe, are treated the same.

Figure 1-2 Default UNIX Services Workload



When a UNIX user, for example JDoe, starts a process on MVS, the underlying address spaces are assigned job names based on the user ID as a prefix and a sequence number as a suffix. These names cannot be effectively used by the Workload Manager to distinguish work. Figure 1-2 shows how iPlanet UDS partitions might be named and how they run (indistinguishably) by default in the UNIX service class.

In order to allow the Workload Manager to distinguish one or more iPlanet UDS processes as workloads that can be managed. The iPlanet UDS administrator and the OS/390 system administrator must work together to set up policies and naming conventions that allows the Workload manager to recognize one or more iPlanet UDS processes as a specific workload. This process is described in the next subsections.

Defining Service Classes for iPlanet UDS Processes

To define a service class for one or more iPlanet UDS processes, the OS/390 system administrator must work together with the iPlanet UDS administrator to do the following:

- establish WLM job names for iPlanet UDS processes
- define WLM classification rules based on the job name; that is, map job names to a particular service class
- associate performance goals with each service class

  Performance goals vary depending on whether the WLM is in compatibility mode or goal mode. See "Compatibility and Goal Mode" on page 45 for more information.
- optionally, add a new report class or classes for iPlanet UDS processes

  For more information about report classes, see "Using Report Classes" on page 45.

In turn, the OS/390 performance administrator must ask the OS/390 security administrator to do the following:

- define the Security Access Facility (SAF) Facility Class BPX.JOBNAME
- permit the iPlanet UDS administrator user ID to have read access to this facility class

The iPlanet UDS administrator must ensure that the USS environment variable _BPX_JOBNAME is set to the appropriate iPlanet UDS process name. This can be done in one of the following ways:

- update the fortedef.sh file in the $FORTE_ROOT directory to set _BPX_JOBNAME
- create a custom shell script that sources fortedef.sh and then set _BPX_JOBNAME before manually starting a process (ftexec).
- manually set _BPX_JOBNAME and then start a process (ftexec)

The next two sections describe the effect of these techniques.

Assigning a Service Class to a Group of iPlanet UDS Partitions

To classify a group of iPlanet UDS partitions as a service class with distinct performance goals:

- The OS/390 performance administrator must edit, install, and activate a WLM policy with iPlanet UDS classification rules.

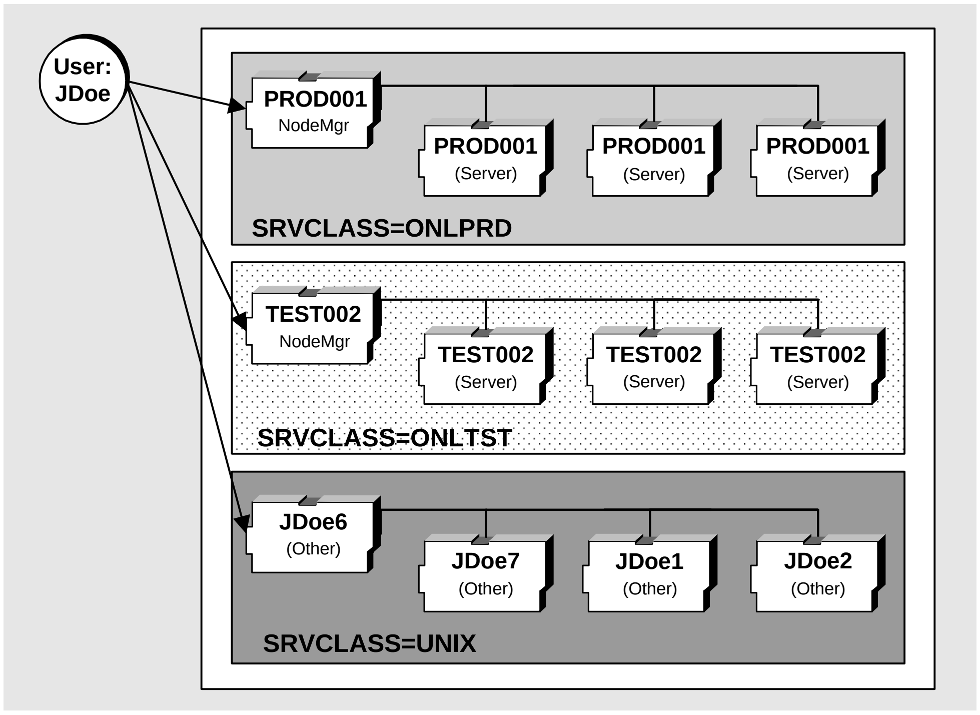
  For example, under workload type OMVS, the administrator would specify two or more rules using different qualifier Transaction Name (TN) names, map each of these names to a service class, and assign a performance goal to each class. For example, the transaction names might be “PROD*” or “TEST*”; the performance goals might be specified as ONLPRD (for production work) or ONLTST (for testing work).

- The iPlanet UDS administrator who starts iPlanet UDS processes, would then update the fortedef.sh file to set the appropriate name in the _BPX_JOBNAME environment variable.

  For example, the _BPX_JOBNAME variable might be set to “PROD*xxx*” for production work, or “TEST*yyy*” for testing work, where *xxx* and *yyy* are arbitrary numbers.

Figure 1-3 shows how the workload management picture changes as iPlanet UDS partitions are associated with different service classes (and therefore different performance goals). The partitions running as part of the ONLPRD service class might get the highest priority; the partitions running as part of the UNIX service class might get the next highest; and those running as part of ONLTST would get the lowest.

Figure 1-3 iPlanet UDS Partitions as Distinct Service Classes



Prioritizing a Single iPlanet UDS Partition

It is also possible to run a single iPlanet UDS partition with a performance goal that is different from the goals set for production and testing work. In order to do this, the partition cannot be autostarted; it must be started manually.

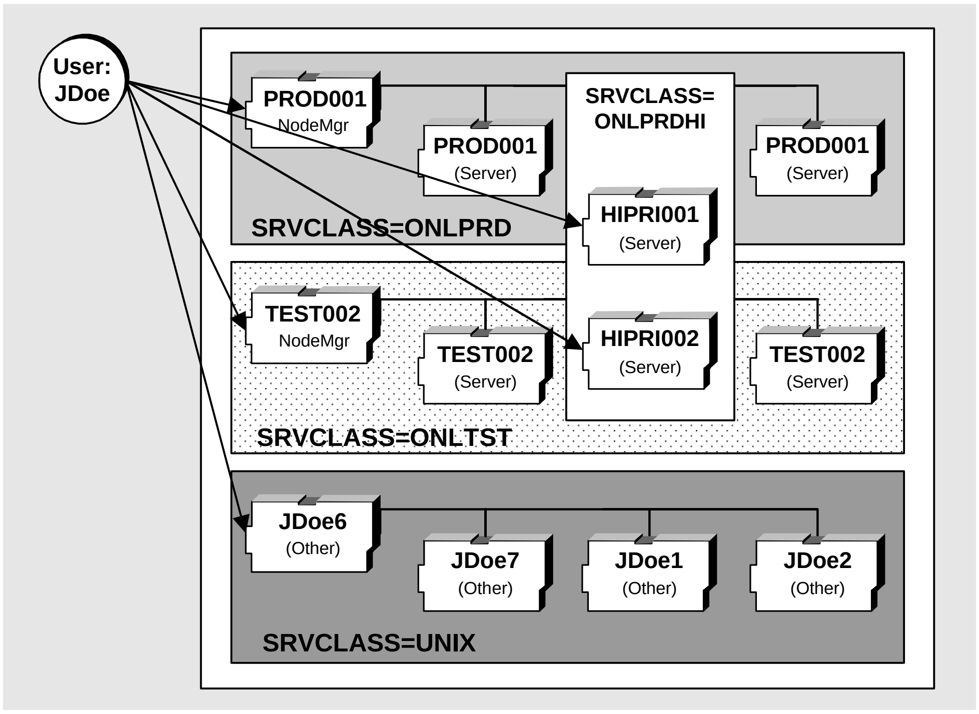
The set up required is the same as for the previous case. In addition,

- The OS/390 administrator must update the WLM policy to add a new job name, for example HIPRI*, and relate it to a new or existing service class with a different performance goal, for example ONLPRDHI.

- The iPlanet UDS administrator must create a shell script which does the following:
    - sources the fortedef.sh file for the appropriate iPlanet UDS environment (for example, test or production)
    - overrides the _BPX_JOBNAME setting (for example, with HIPRI*zzz*, where *zzz* is an arbitrary number)
    - starts the favored partition (ftexec or /xxx/yyy.exe)

The result is illustrated in Figure 1-4:

Figure 1-4 Prioritizing a Single iPlanet UDS Partition



The partitions HIPRI001 and HIPRI002 belong to the service class ONLPRDHI, and have different performance goals from service class ONLPRD and ONLTST.

Compatibility and Goal Mode

The Workload Manager can run in one of two modes: compatibility mode or goal mode:

- In compatibility mode, a configuration file is used to define performance metrics in terms of ranked performance groups.
- In goal mode, performance targets are defined for each service class either as a limit or as an ideal value. The targets that can be set are *discretionary*, *velocity*, and *response time*. For practical purposes, the only target that can be meaningfully set for iPlanet UDS processes is velocity.

How the OS/390 defines performance targets for iPlanet UDS service classes varies depending on the WLM's mode, as illustrated in Table 1-1.

Table 1-1 Performance Goals and WLM Modes

Mode	Rule	Service Class	Report Class	Performance Goals
Compatibility	PROD*	ONLPRD	FPROD	PGN = HI
	TEST*	ONLTST	FTST	PGN = LO
	OTHER	OTHER	OTHER	PGN = MED
Goal	PROD*	ONLPRD	FPROD	Velocity goal HI
	TEST*	ONLTST	FTST	Velocity goal LO
	OTHER	OTHER	OTHER	Velocity goal MED

Of course, all names shown in the table are arbitrary and given only as an example.

Using Report Classes

By default, Workload Management provides data for reporting based on the service class. An iPlanet UDS installation can define additional report classes to get finer granularity of reporting or to combine data across service classes.

For additional information, see *OS/390 V2R10.0 MVS Planning: Workload Management*.

Chapter 2

Overview of the iPlanet UDS Transaction Adapter

This chapter provides an overview of the iPlanet UDS Transaction Adapter for OS/390. It describes the following:

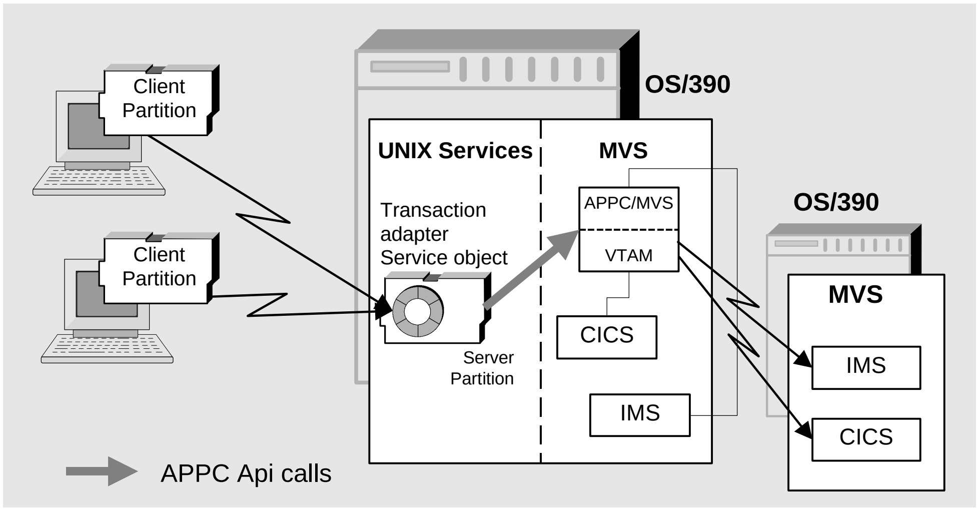
- the functionality provided by the product
- the projects and classes included in the product
- the architecture of the product

Overview

The iPlanet UDS Transaction Adapter for OS/390 software provides a simple interface to online transaction processors (OLTPs) running within an SNA network, using SNA Advanced Program-to-Program Communications (APPC). With this interface, transaction programs (TPs) running on OLTP systems such as CICS/ESA, CICS/TS, IMS/TM, and APPC/MVS can be invoked from within the iPlanet UDS Application Environment. This interface allows existing applications running in a System/390 OLTP environment to be integrated with applications running in the iPlanet UDS Application Environment. This document explains how to use the iPlanet UDS Transaction Adapter software. It assumes you have some knowledge of APPC and OLTP concepts and programming.

Figure 2-1 shows how components installed on client nodes call out, using the iPlanet UDS Transaction Adapter on an OS/390 server node, to online transaction processors running either on the same OS/390 node or on a different OS/390 node.

Figure 2-1 Using the iPlanet UDS Transaction Adapter to Call Online Transaction Processors



APPC Interfaces

The OS/390 application programming interfaces (APIs) to APPC consist of a number of routines callable from any program. These routines are provided as part of the APPC/MVS component of OS/390. There are two APIs, Common Programming Interface – Communications (CPI-C) and APPC/MVS Callable Services. The iPlanet UDS Transaction Adapter uses the APPC/MVS Callable Services API, because it provides some capabilities that are not available with the CPI-C API.

The iPlanet UDS Transaction Adapter software is provided in the APPC project, which contains the classes that implement the Transaction Adapter software. The APPC project contains classes, instances of which can eventually be deployed on client as well as on server partitions. For example, the object that implements security functions can be deployed on a client partition, whereas the Transaction Adapter, which serves as an interface to APPC, is always deployed on a server partition.

The basic interface in the iPlanet UDS Transaction Adapter is the “direct” interface, which offers a one-to-one mapping between the APPC verb set and iPlanet UDS classes and methods. The APPCApi class implements the direct interface. In addition, a “simple” interface is provided, and gives the iPlanet UDS programmer a significantly easier way to work with APPC while sacrificing little of the power of APPC. The simple interface is implemented by the APPCConversation and

APPCSecurityInfo classes and uses the direct interface for performing APPC functions. Unless your application requires direct control over the behavior of APPC, your code will normally use the simple interface. Use of the direct interface is not recommended unless it is absolutely necessary for your application to have direct access to APPC functions.

The following sections provide a brief overview of the two interfaces and the methods they provide. The classes implementing the simple interface, and their methods and attributes, are described in detail in online help. The direct interface is described in detail in Technote #12177 entitled 'Using the Direct APPC Interface of the Transaction Adapter for OS/390'.

Simple Interface

The iPlanet UDS Transaction Adapter offers the iPlanet UDS developer a powerful, high-level simple interface to APPC. Typically, in order to interact with an APPC transaction program, an application must make APPC API calls to manage the state of the APPC conversation, in addition to the calls it makes to send and receive data. The simple interface in the iPlanet UDS Transaction Adapter hides this complexity by providing a straightforward read and write interface that manages the state of the conversation internally. The APPCConversation and APPCSecurityInfo classes, along with one method in the APPCApi class, provide this interface.

- The APPCConversation class provides the actual representation of an APPC conversation.
- The APPCSecurityInfo class provides a secure mechanism for providing security parameters for the APPC conversation.
- The APPCApi class provides the underlying APPC support for the APPCConversation class, and its NewConversation method is used to obtain a new APPCConversation object that resides in the same partition as the APPCApi object.

Your application will invoke methods on the classes shown in Table 2-1 to perform high-level functions, that will in turn invoke methods on the APPCApi class. It is important to note that all applications must use the NewConversation method of the APPCApi class to create a new APPCConversation object. This is necessary to provide the appropriate setup and anchoring of the APPCConversation object.

It is strongly recommended that you use the simple interface whenever possible, as it greatly simplifies the interface between your iPlanet UDS application and OLTP programs.

Table 2-1 iPlanet UDS Transaction Adapter Simple Interface

Class	Method	Purpose
APPCApi	NewConversation	Instantiates APPCConversation and provides the object to the caller.
APPCConversation	Open	Allocates a conversation with a partner transaction program.
	Close	Deallocates a conversation with a partner transaction program.
	Read	Reads data from an open conversation with a partner transaction program.
	SetNullReplacementChar	Overrides the default replacement character (blank) that is used to replace null characters found in buffers received from a partner transaction program. Also enables null replacement if it is not enabled.
	SetReplaceNulls	Enables or disables the replacement of null characters in buffers received from a partner transaction program with a specified character.
	Write	Writes data to an open conversation with a partner transaction program.
APPCSecurityInfo	GetPassword	Returns the conversation security password.
	GetProfile	Returns the conversation security profile.
	GetUserid	Returns the conversation security userid.
	SetPassword	Sets the conversation security password.
	SetProfile	Sets the conversation security profile name (RACF group name).
	SetUserid	Sets the conversation security userid.

Direct Interface

The iPlanet UDS Transaction Adapter software's direct interface provides "iPlanet UDS-friendly" access to APPC routines while following the general design of the APPC application programming interface. This interface is contained completely within the APPCApi class, which must be deployed on an OS/390 node. If you choose to use the direct interface, your iPlanet UDS application will invoke the methods on the class shown in Table 2-2 to perform APPC operations. If you choose to use the simple interface, your iPlanet UDS application will indirectly invoke methods on this class through the methods in the APPCConversation class.

Table 2-2 iPlanet UDS Transaction Adapter Direct Interface

Class	Method	Purpose
APPCApi	Accept	Accepts an incoming conversation requested by a partner transaction program.
	Allocate	Allocates a conversation with a partner transaction program.
	Confirm	Sends a request for confirmation to the partner transaction program and waits for a response.
	Confirmed	Sends a confirmation to the partner transaction program in response to a request for confirmation.
	Deallocate	Deallocates a conversation with a partner transaction program.
	ErrorExtract	Retrieves detailed information about an error on the previous operation.
	Flush	Forces all buffered data to be sent immediately to the partner transaction program.
	GetAttributes	Retrieves information about the current conversation such as the local and partner LU names, the mode name, the partner TP name, the conversation type and state, and the security parameters.
	GetTpProperties	Retrieves the APPC properties of the local application.
	GetType	Retrieves the conversation type of the current conversation.
	PrepareToReceive	Changes the conversation from SEND state to RECEIVE state in order to receive data from the partner transaction program.

Table 2-2 iPlanet UDS Transaction Adapter Direct Interface *(Continued)*

Class	Method	Purpose
	ReceiveAndWait	Receives all available data and status information from the partner transaction program. If nothing is available, waits for data and/or status information to arrive.
	ReceiveImmediate	Receives all immediately available data and status information from the partner transaction program.
	RequestToSend	Notifies the partner transaction program that the local transaction program is requesting to enter the SEND state.
	SendData	Sends data to the partner transaction program. The data can be buffered or sent immediately.
	SendError	Sends an error indication to the partner transaction program.

Architecture

The design of the iPlanet UDS Transaction Adapter is intended to provide flexibility in how it is deployed and used by your applications. However, there is one restriction that is important to understand. Because the APPC API used by the iPlanet UDS Transaction Adapter is specific to OS/390 systems, the APPC server portion of an application built using the Adapter can be deployed only on OS/390 nodes running the iPlanet UDS for OS/390. Other parts of the server side of the application can be deployed on any iPlanet UDS node that can communicate with the OS/390 node.

The iPlanet UDS Transaction Adapter contains several classes: APPCApi, APPCConversation, APPCSecurityInfo, and APPCException. Their functions and relationships are described in the following sections.

Reference Information

You can find complete reference information for the following classes by selecting the topic `APPC Library` in the online help.

APPCApi

The APPCApi class is the server-only class, and uses the APPC/MVS Callable Services API to perform all APPC operations. The APPCApi class provides the server functionality upon which all iPlanet UDS Transaction Adapter applications rely. APPCApi methods call the entry points in the APPC/MVS Callable Services API directly, passing information from the APPCConversation object for the appropriate conversation. A new instance of the APPCConversation class is created by APPCApi for each APPC conversation requested by a client application. The APPCApi class should be deployed as a service object so that it can be shared by multiple clients.

APPCConversation

The APPCConversation class represents a single APPC conversation between an iPlanet UDS application and a partner transaction program running under an OLTP system. This class contains all of the methods necessary to operate upon the APPC conversation. There are methods to open the APPC conversation, close the APPC conversation, read data from the partner transaction program, and write data to the partner transaction program. These methods provide an abstraction of the actual APPC API; however, they are at a much higher level and are simpler to use.

An instance of the APPCConversation class is created by the NewConversation method in the APPCApi class for each new APPC conversation. The APPCConversation objects are all anchored because they must remain in the same iPlanet UDS partition for the duration of the conversation. An APPCConversation object cannot move from one partition to another. The reason for this is that APPC/MVS requires that all operations for a single APPC conversation be executed from the same address space, and each iPlanet UDS partition running under OS/390 is in a separate address space.

APPCSecurityInfo

The APPCSecurityInfo class is used to provide a secure way to supply APPC conversation security parameters to the APPC API. This class is needed only when APPC conversation security is enabled by the OLTP system. It contains methods to set up the userid, password, and profile (or group) parameters to be used for conversation security. Once these parameters are set for the APPCSecurityInfo object, they are encrypted and are decrypted only when they are about to be passed to the APPC API. The security information is guaranteed never to flow across the network in clear text form.

➤ To implement APPC conversation security

1. Use the NewConversation method to get an APPCConversation class.
2. Create an instance of the APPCSecurityInfo class.
3. Use the methods of the APPCSecurityInfo class to set security parameters for the instance you created in Step 2 on page 54.

   Both the userid and the password are required, but the profile is optional and should be used only if the OLTP system supports it. This varies with different OLTP systems and the documentation for the specific OLTP system being used should be checked for the applicability of the security parameters.
4. Pass an instance of the security object to the Open method of your APPCConversation class.

   All subsequent read and write calls you make on your conversation are now secure.

You do not have to explicitly delete the security object. It will be automatically disposed of by the garbage collector.

APPCException

The APPCException class is used to report APPC errors back to the application. A pointer to the APPCConversation object on which the error occurred is included in each APPCException object. In addition, error messages describing the error in detail are included in the object. The APPCException class is instantiated wherever the APPC error occurred, usually in one of the APPCConversation methods following an error reported by the return code from an APPC operation.

Chapter 3

Using the iPlanet UDS Transaction Adapter

This chapter provides information on how to use the iPlanet UDS Transaction Adapter for OS/390 in developing an application. It explains how you do the following:

- design an application
- use threading and partitioning
- use the Transaction Adapter classes
- use diagnostic tools to locate problems in your application

For information about known problems associated with the use of the COBOLField class, please look up Bulletin 421 at *http://www.forte.com/support/bulletins.html*.

Designing an Application

How you design an iPlanet UDS application partially depends on design decisions made when the OLTP transaction programs were written. Understanding the logical flow of the OLTP transactions will help guide the design process of the iPlanet UDS application. There are several questions you need to answer when you design your application: some concern the OLTP transaction programs and some concern the iPlanet UDS application. First, here are some questions you should answer about the OLTP transaction programs:

1. On which OLTP does the transaction program reside?

   Each OLTP system has a unique programming environment with its own capabilities and constraints. This affects how the iPlanet UDS application interfaces with the transaction program.

2. Does the transaction program already have a 3270 terminal-based user interface? If so, what OLTP facilities are used for that interface?

   This is an important issue to the application programmer on the OLTP system. The 3270 interface provided by CICS, Basic Mapping Support (BMS), does not provide automatic conversion to APPC. The CICS transaction program must be modified to support APPC communications or the CICS screen-scraping facility, Front End Programming Interface (FEPI) must be used to write an APPC front-end to the transaction. Conversely, the 3270 interface provided by IMS/TM, Message Formatting Service (MFS), provides fully automatic conversion to APPC that is completely transparent to the IMS transaction program. This is known as *implicit* APPC support. In addition, IMS transaction programs can be written solely for access via APPC using *explicit* APPC support.

3. Is the application logic separated from the user interface logic in a way that allows it to be called from another program?

   This is an issue primarily for CICS transaction programs. Many CICS applications are designed to run in a CICS multiple region environment, where there are two CICS regions known as the Terminal Owning Region (TOR) and the Application Owning Region (AOR). The TOR owns and manages all of the terminals, and the AOR owns and manages all of the data and the business logic to access that data. In such an application, the user interface logic and the business logic are split into separate programs, with the user interface programs calling the business logic programs through a Distributed Program Link (DPL). Such applications can easily be adapted to use APPC communications by writing separate user interface programs that use APPC communications, or by making the existing user interface programs capable of using either BMS or APPC for communications.

4. Is the transaction program a simple one-shot transaction with a single input and a single output, or is it a more complex interactive transaction?

   The iPlanet UDS application programmer must understand the logic of the transaction program in order to design the application to interface correctly with the transaction program. If the transaction program has multiple ways to interact with the user, the iPlanet UDS application must be programmed to handle all of them. The more complex the interactions of the transaction program with the user, the more complex the iPlanet UDS application becomes.

5. What are the input and output data for the transaction program?

   The transaction program must be examined to determine the input and output data items to be exchanged with the iPlanet UDS application. Character data (COBOL PIC X) exchanged using TextData buffers is automatically translated from EBCDIC to the appropriate character set on the client platform. However, no conversion of COBOL data types such as packed decimal, zoned decimal, binary integer or floating point is provided by this release of the Transaction Adapter. If those data types are used, the iPlanet UDS application must be designed to exchange the data in BinaryData buffers and to perform the necessary conversions itself.

Here are some questions you should answer about the iPlanet UDS application:

1. Will your iPlanet UDS application require concurrent access to multiple OLTP transaction programs, or will it require access to only one OLTP transaction program at a time?

   The Transaction Adapter allows multiple concurrent conversations with OLTP transaction programs. All that is required is that each conversation has a separate APPCConversation object. iPlanet UDS multitasking can be used to reduce the wait time for the application, and maximize throughput, by processing each conversation on a separate task.

2. Will your application be gathering information from multiple data sources that could be accessed concurrently?

   Again, iPlanet UDS multitasking can be used to access multiple data sources concurrently. This can include conversations with OLTP transaction programs.

3. Is there work your application can perform while waiting for a response from an OLTP transaction program?

   If your application can perform other work while a conversation with a transaction program is waiting for a response, iPlanet UDS multitasking is again a good solution. The APPCConversation methods that perform APPC operations are synchronous and block until the operations complete.

4. What data types are exchanged with the transaction program?

   If your application exchanges only character data with the transaction program, you may use TextData objects to handle the data so that iPlanet UDS will perform the automatic character set conversion from EBCDIC to the appropriate character set for the client platform.

Use a COBOLBuffer object if your application exchanges data with a COBOL program. In particular, COBOLBuffer has mechanisms to help you convert COBOL elementary items that are part of a group (like an 01-level record) into textual or numeric values that can be manipulated in TOOL.

If your application exchanges other data types with the transaction program, you must use BinaryData objects for handling the data. The contents of BinaryData objects do not undergo conversions of any sort. You must perform data conversions yourself, as necessary.

The answers to these questions help determine the appropriate application design approach you should follow.

Threading and Partitioning Considerations

The APPC programming interface is thread-safe, which means that the code in the APPC library can be executed safely by more than one flow of execution (thread) at the same time. A single APPC conversation, however, can be operated upon by only one thread at a time. Furthermore, each APPC operation on a single conversation must complete before another operation on that conversation can be initiated.

Because the APPC library is thread-safe, the APPCApi class is designed to be thread-safe as well. This is done by having each separate APPC conversation, or thread, be represented by a separate object, the APPCConversation object.

As discussed earlier in the architecture discussion in "APPCConversation" on page 53, the APPCConversation object is always anchored so that it remains in the same iPlanet UDS partition for the duration of its existence. The APPCConversation object will ensure that all operations on the APPC conversation are performed by the same iPlanet UDS partition. As a result, the use of Message dialog duration with an APPCApi-based service object does not produce the expected behavior, and is therefore not recommended.

An APPCApi-based service object is multi-threaded, so it would not normally be a good candidate for load balancing, because that forces the service object to be single-threaded. However, if the application must support more than 100 concurrently active threads, load balancing with a custom router can improve performance by limiting the number of threads managed by each partition replicate. The following algorithm can be used to compute the number of concurrently active threads based on the number of users and the transaction rate. (The use of this algorithm requires that you have some idea about the anticipated average response time and the anticipated average amount of time between transactions for each user (the *think time*).)

➤ **To compute the number of concurrently active threads**

1. Calculate the transaction rate in transactions per second (TPS) by dividing the number of users (USERS) by the sum of the response time in seconds (RT) and the think time in seconds (TT):

```
TPS = USERS / (RT + TT)
```

2. Calculate the number of concurrently active threads (THREADS) by multiplying the transaction rate (TPS) by the response time in seconds (RT):

```
THREADS = TPS * RT
```

If this calculation results in a number greater than 100, your application is a good candidate for a custom router. The router would have to distribute work among the partition replicates in a way that would keep each conversation on the same partition replicate for its duration, but allow the partitions to remain multi-threaded. In this case, Transaction dialog duration should be used for the APPCApi-based service object. Care must be taken in designing the application to ensure that an iPlanet UDS transaction maps properly onto the APPC conversation with the OLTP transaction program. The iPlanet UDS transaction should begin before the APPC conversation is opened, and should not end until after the APPC conversation is closed.

An additional consequence of the requirement that all operations of an APPC conversation be handled within the same address space is that failover of the APPCApi-based service object's partition cannot be handled automatically. When the partition fails, all of its APPC conversations are terminated abnormally by the system, and the applications at the other end of the conversations receive a return code indicating that fact.

Using the Transaction Adapter Classes

To use the classes provided by the iPlanet UDS Transaction Adapter, you must first make the APPC project a supplier plan to your TOOL project using the workshop. Once you have done this, you can access all of the classes provided by the APPC project, which includes APPCApi, APPCConversation, APPCException, and APPCSecurityInfo. This section provides a step-by-step guide for using the classes to communicate with an OLTP transaction program, or *partner transaction program*.

Define the Service Object

Decide whether you want to have a unique APPCApi-based service object for your application or whether you want to use a service object designed for sharing among applications. If you want a separate service object for your application, you must define it in your project and assign it a unique name. Make sure the service object is defined with Transaction or Session dialog duration. If you want to use a service object that is shared among applications, you must have already defined the service object, and must define a reference partition to access the service object.

Collect OLTP Information

Next, you must collect some information from the OS/390 system programmer who is in charge of the OLTP system where the partner transaction program resides. You need to know the *symbolic destination name* that represents the OLTP system. This name is used to reference a predefined set of parameters including a *partner LU name*, a *mode name*, and an optional *TP name*, known as the *side information*. Typically, there will be one set of side information defined for each OLTP system. In this case, you will also need to know the TP name of the partner transaction program. It is also possible to have a set of side information defined for each transaction program. If your system is defined this way, you do not need to know the TP name because it is already part of the side information.

The other piece of information you need from the OLTP system is the security requirement. If RACF or some other security package is used to control access to the OLTP, then a valid userid and password, and possibly a profile (or group), are needed for your application to gain access. This information is available from the OS/390 system programmer or security administrator.

Write the Application

Now you are ready to write your iPlanet UDS application. Complete the following steps to communicate with the partner transaction program:

1. Obtain a new APPCConversation object from the APPCApi-based service object. To do this, invoke the NewConversation method on the service object, which will return a pointer to your APPCConversation object. The following code fragment shows how to do this:

   ```
   myConv : APPCConversation = APPCApiSO.NewConversation();
   ```

2. If security parameters are required, create a new APPCSecurityInfo object. Use the SetUserid, SetPassword, and SetProfile methods to set the security userid, password, and profile or group name. For example:

```
mySec : APPCSecurityInfo = new;
mySec.SetUserid(userid='userid');
mySec.SetPassword(password='password');
mySec.SetProfile(profile='profile');
```

3. Open the APPC conversation with the partner transaction program by invoking the Open method on the APPCConversation object. Use the `dest` parameter to specify the symbolic destination name, and if necessary, use the `lu`, `mode`, and `tp` parameters to specify the partner LU name, the mode entry name, and the TP name, respectively. If security parameters are required, use the `security` parameter to specify the APPCSecurityInfo object that contains your security information. The following example assumes that there is one side profile defined for the OLTP system, and that security information is required:

   ```
   myConv.Open(dest='destname',tp='tpname',security=mySec);
   ```

4. Send the first input data item to the partner transaction program using the Write method on the APPCConversation object. The `dataBuffer` parameter must specify a TextData or BinaryData object that contains the data to be sent, and the `writeLength` parameter must specify the length of the data to be sent. If the data is character data and you want iPlanet UDS to perform automatic conversion of the data to the appropriate EBCDIC character set, use a TextData object to contain the data. If the data is not in a format that can be converted automatically by iPlanet UDS, use a BinaryData object and ensure that the data is in the correct format for the partner transaction program. This example shows how to send a TextData buffer:

```
inBuf : TextData = new;
inLen : integer;
inBuf.SetValue(source='This is a text message');
inLen=inBuf.ActualSize;
myConv.Write(dataBuffer=inBuf,writeLength=inLen);
```

5. Receive the output data from the partner transaction program using the Read method on the APPCConversation object. The `dataBuffer` parameter must specify a TextData, COBOLBuffer, or BinaryData object that will contain the data received, and the `readLength` parameter must specify the maximum length of the data to be received. If the data is character data and you want iPlanet UDS to perform automatic conversion of the data to the appropriate client character set, use a TextData object to contain the data.

   If the data is not in a format that can be converted automatically by iPlanet UDS, use a COBOLBuffer or BinaryData object and ensure that the data is converted by your application once it is received. Once the Read method has returned, the `readLength` parameter will contain the actual length of the data that was received. Here is code to receive a TextData buffer:

   ```
   outBuf : TextData = new;
   outLen : integer = max_length_to_receive;
   outBuf.SetAllocatedSize(n=outLen);
   myConv.Read(dataBuffer=outBuf,readLength=outLen);
   ```

   Note that the Read method is a blocking operation. If the partner transaction program has not yet sent data when the Read method is invoked, the method will wait until data is received from the partner transaction program. Care should be exercised in writing your application to ensure that a Read that never completes cannot occur.

6. Close the APPC conversation with the partner transaction program by invoking the Close method on the APPCConversation object. For example:

   `myConv.Close();`

   At this point the object can be reused for another APPC conversation by calling the Open method again with new parameters.

   Note that an APPC conversation can be deallocated only from one side of the conversation. Whichever side deallocates first, causes the other side to receive a return code on its deallocate operation indicating that the other side already deallocated. This is not treated as an error. In most cases, the OLTP side will perform its deallocate first, so the iPlanet UDS side will receive the APPC_RC_DEALLOCATED_NORMAL return code on its deallocate call. This is handled internally by the APPCConversation class and is not exposed to the client application.

Diagnostic Tools

This section describes the iPlanet UDS and VTAM traces you can use to diagnose problems.

iPlanet UDS Traces

There are high-level and low-level diagnostic traces built into the APPCApi and APPCConversation classes that you can use to locate problems with an application. The high-level trace is called the *conversation trace*: it traces entry to and exit from method calls within the APPCConversation class, along with significant events and data. The low-level trace is called the *API trace*: it traces the initiation and completion of every call to the APPC/MVS Callable Services, with pertinent completion information included.

The conversation and API traces are helpful when there is a mismatch between what the iPlanet UDS application is sending or expecting to receive and what the partner transaction program is expecting to receive or is sending.

You enable these traces by setting standard iPlanet UDS trace flags. The traces fall under the `fo` service.

➤ **To enable the conversation trace**

1. Set the following trace flag:

```
trc:fo:62
```

➤ **To enable the API trace**

1. Set the following trace flag:

```
trc:fo:63
```

You must set the trace flags on the OS/390 node where the APPCApi-based service object executes. They can be set through the FORTE_LOGGER_SETUP environment variable prior to the startup of the node manager for the OS/390 node, through the ModifyFlags method of the LogMgr object, or through the Component > Modify Log Flags command in the Environment Console.

The following example shows the output produced with both the conversation and the API traces enabled. The API trace output is shown in bold:

Code Example 3-1 Output produced with conversation and API traces enabled

```
APPCApi.NewConversation: entered
APPCApi.NewConversation: exiting normally
APPCConversation.Open: entered
APPCConversation.Open: dest = ’FRTCEXSI’,
  lu = ’’,
  mode = ’’,
  tp = ’FR02’,
  synclevel = 0
APPCConversation.Open: calling APPCApi.Allocate
APPCApi.Allocate: calling ATBALC2
APPCApi.Allocate: dest = ’FRTCEXSI’,
  lu = ’                 ’,
  mode = ’        ’,
  tp = ’FR02’
APPCApi.Allocate: synclevel = 0,
  security type = 100,
  userid = ’          ’,
  profile = ’          ’
APPCApi.Allocate: ATBALC2 returned 0
APPCConversation.Open: APPCApi.Allocate returned,
APPC return code = 0
APPCConversation.Open: calling APPCApi.GetAttributes
APPCApi.GetAttributes: calling ATBGTA2
APPCApi.GetAttributes: ATBGTA2 returned 0
APPCApi.GetAttributes: local LU = ’MVSLU03’,
  partner LU = ’NET3.MVSLU03’,
‘mode name = 'APPCHOST',
‘TP name = 'FR02'
APPCApi.GetAttributes: userid = '',
  profile = '',
  synclevel = 0,
  conversation type = 1
APPCConversation.Open: APPCApi.GetAttributes returned,
  APPC return code = 0
APPCConversation.Open: exiting normally
APPCConversation.Write: entered
APPCConversation.Write: writeLength = 6
APPCConversation.Write: calling APPCApi.SendData
APPCApi.SendData: calling ATBSEND
APPCApi.SendData: length = 6,
  send type = 0
```

Code Example 3-1 Output produced with conversation and API traces enabled *(Continued)*

```
APPCApi.SendData: ATBSEND returned 0
APPCApi.SendData: RTS = 0
APPCConversation.Write: APPCApi.SendData returned,
  APPC return code = 0
APPCConversation.Write: exiting normally
APPCConversation.Read: entered
APPCConversation.Read: readLength = 81
APPCConversation.Read: calling APPCApi.PrepareToReceive
APPCApi.PrepareToReceive: calling ATBPTR
APPCApi.PrepareToReceive: ptrType is 1
APPCApi.PrepareToReceive: ATBPTR returned 0
APPCConversation.Read: APPCApi.PrepareToReceive returned,
  APPC return code = 0
APPCConversation.Read: calling APPCApi.ReceiveAndWait
APPCApi.ReceiveAndWait: calling ATBRCVW
APPCApi.ReceiveAndWait: length = 81
APPCApi.ReceiveAndWait: ATBRCVW returned 18
APPCApi.ReceiveAndWait: length = 81,
  status = 0,
  data = 2,
  RTS = 0
APPCConversation.Read: APPCApi.ReceiveAndWait returned,
  APPC return code = 0
APPCConversation.SetState: entered
APPCConversation.SetState: state is now RESET
APPCConversation.SetState: exiting normally
APPCConversation.Read: exiting normally
APPCConversation.Read: readLength = 81
APPCConversation.Close: entered
APPCConversation.Close: exiting, conversation inactive
```

VTAM Buffer Trace

On OS/390 systems, all SNA data traffic is eventually handled by VTAM. VTAM provides extensive tracing capabilities using GTF for capturing the trace data. The most useful trace for debugging problems with the Transaction Adapter is the VTAM buffer trace. This trace provides a detailed record of the flow of data between the Transaction Adapter and the target LU. You can use it to determine whether data sent by the Transaction Adapter was received at the target LU and vice versa, as well as exactly what data was transferred. This is especially useful if the data being transferred is binary data, because the API trace will not show the contents of binary data buffers.

Typically, the OS/390 system programmer is involved in assisting with the setup and collection of VTAM traces. Here is a brief summary of the steps required to obtain a VTAM buffer trace. The VTAM buffer trace is enabled by issuing an MVS MODIFY command to the VTAM started task. The format of the MODIFY command is:

```
F net,TRACE,ID=luname,TYPE=BUF
```

In the above command, *net* is the step name by which the VTAM started task is known to the system, and *luname* is the name of either the APPC/MVS LU being used by the Transaction Adapter or the name of the target OLTP system's LU.

In order to record the trace data, GTF must be executing with the following trace parameter included:

```
TRACE=USR
```

Once the application has been executed, the VTAM buffer trace can be disabled using the following command:

```
F net,NOTRACE,ID=luname,TYPE=BUF
```

GTF must be stopped and then the trace can be viewed using IPCS.

Chapter 4

Integrating IBM OS/390-Hosted COBOL Applications with iPlanet UDS

Transaction Adapter Builder is a tool for developing transaction adapters that enable iPlanet UDS applications to share data with OS/390-hosted COBOL-based CICS and IMS transactions. It is implemented as a set of extensions to Fscript, which you use to define transactions in your COBOL code where data need to be shared with your UDS application. When you execute your script the Transaction Adapter Builder generates a `.PEX` file containing classes you use to code your transaction adapter. Your generated classes use the APPC library to manage all data transmission details, including conversions from native COBOL datatypes to TOOL datatypes and back.

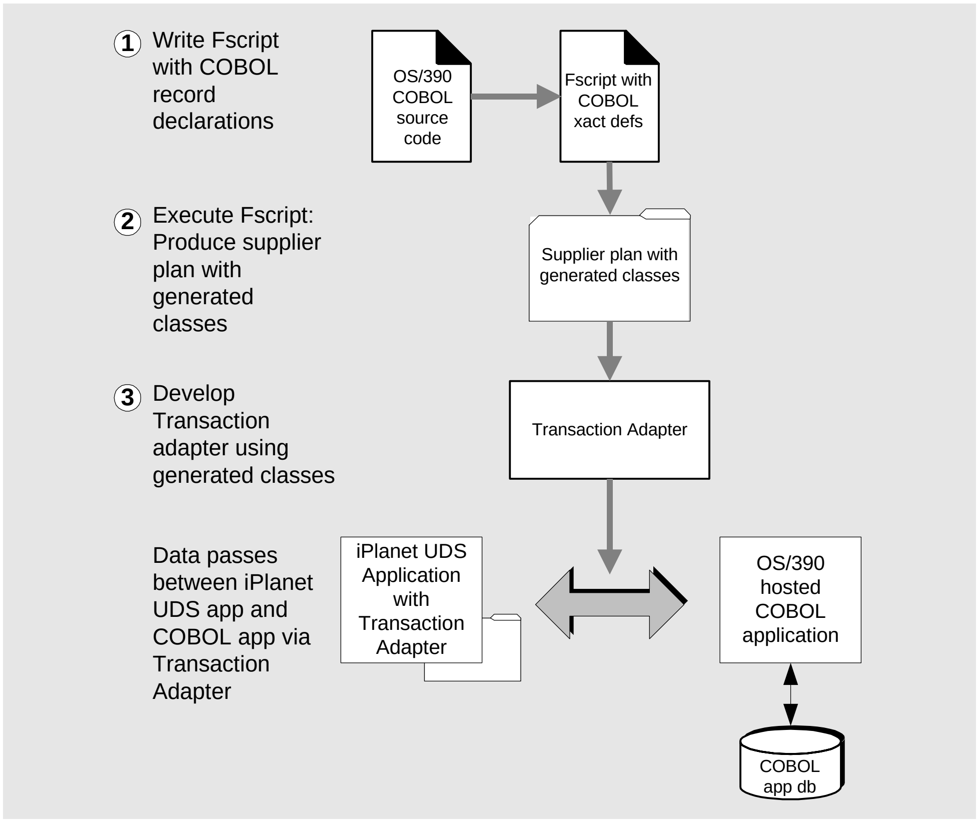
This chapter covers the following topics:

Developing a Transaction Adapter

Before you use the Transaction Adapter Builder you first identify the data exchange points in your COBOL transactions. Once you have done that you then use the Transaction Adapter Builder to perform the following tasks, which are illustrated in Figure 4-1:

1. Use Fscript extensions to define each transaction and its addressing information (see "Fscript Commands" on page 72)
2. Execute the script to generate a supplier plan (see "GenerateTransactionProxy" on page 74)
3. Import the supplier plan into the UDS application

Figure 4-1 Developing a transaction adapter with Transaction Adapter Builder


1
Write Fscript with COBOL record declarations
OS/390 COBOL source code
Fscript with COBOL xact defs
2
Execute Fscript: Produce supplier plan with generated classes
Supplier plan with generated classes
3
Develop Transaction adapter using generated classes
Transaction Adapter
Data passes between iPlanet UDS app and COBOL app via Transaction Adapter
iPlanet UDS Application with Transaction Adapter
OS/390 hosted COBOL application
COBOL app db

Identifying COBOL Exchanges

Your first task is to catalog all relevant data exchange points in your COBOL code. These are the points at which your COBOL application will pass data to and receive data from your UDS applications. Data record declarations can usually be found in *copybook* files stored in partitioned datasets on your MVS file system. For each transaction you need to collect the following information:

- All data inputs and their datatypes
- All data outputs and their datatypes
- The LU6.2 application information for your COBOL application

Once you have a full list of all your data exchanges you also need to know in which order they need to be processed.

Creating the Fscript Script

When you have identified all the data exchanges in your COBOL transactions your next step is to define these exchanges using the Fscript utility. The Transaction Adapter Builder includes an Fscript command set with which you create and remove transaction proxies, exchange methods, arguments, and records. (See "Fscript Commands" on page 72.)

- A *transaction proxy* is a TOOL class that represents the IMS or CICS transaction
- An *exchange method* represents a single data exchange point
- *Arguments* are the input and output parameters used by exchange methods
- *Records* are record declarations, which must be created before they can be passed as arguments

The Transaction Adapter Builder saves all your commands in a script file called `name.scr`, where *name* is the name of the transaction proxy you created. By default, this file is written to the directory in which your iPlanet UDS executables are, typically `FORTE_ROOT/install/bin`. (You can change the directory to which these files are written by using the Fscript `CD` command.) You can later edit this script file as you wish, and use the `GenerateTransactionProxy` command again to create your supplier plan.

NOTE You may find that as you are getting familiar with the Fscript commands it may be easier to work interactively, rather than by creating a script and executing it. This way you can correct any mistakes as you make them, rather than have your script fail to execute completely because of a mistake.

All low-level connectivity and datatype conversion between your COBOL application and your UDS application are managed by classes in the APPC library. The classes generated from your Fscript input call classes from the APPC library. For information about the APPC library see the UDS online help.

Creating and Importing a Supplier Plan

Executing your script generates two files: a `.PEX` file that contains TOOL code and an `.SCR` file that captures all the Fscript commands you used to create your transaction proxy. These files are written to the directory in which you installed your UDS application. (Typically this is `FORTE_ROOT/install/bin`.) The `.PEX` file can be imported into your UDS application as a supplier plan; it includes all the TOOL classes required for managing the connection between your UDS application and your COBOL application. You can use the `.SCR` file to recreate your transaction proxy at a later date, or simply use it as a backup.

➤ **To use the .PEX file as a supplier plan**

1. Import it into the development repository where you want to use it.
2. In the Repository Workshop select File > Include Public Plan to include the `.PEX` file in your workspace.
3. In the Project Workshop select File > Supplier Plans to import the contents of the `.PEX` file as a supplier plan for your project.

Creating a Transaction Adapter

You create the actual transaction adapter in TOOL using your generated classes and methods. (Do not edit the generated TOOL code itself.) If you require additional functionality you can create extensions to your transaction adapter using the generated classes as superclasses.

The following code fragment illustrates a simple example of TOOL code using three generated classes: GTBZ, ZipcodeRecord, and TheaterRecord.

```
-- Generates a list of movie theaters in Oakland
OaklandTheaters : Array of TextData = new;
for zip in 94601 to 94627 do
  -- Make the transaction proxy and connect to the
  -- transaction monitor.
  gtbz : GTBZ = new;
  gtbz.Open();

  -- Build the COBOL input record.
  zipcodeRec : ZipcodeRecord = new;
  zipcodeRec.value = zip;

  -- Prepare a reference to the output record.
  rec : TheaterRecord = NIL;

  -- Call a TransactionProxy method. This one retrieves
  -- a TheaterRecord when given a ZipcodeRecord.
  gtbz.GetTheaterRecord( zipcodeRec, rec );
  OaklandTheaters.AppendRow( rec.TheaterName );

  -- Close proxy connection.
  gtbz.Close();
end for;
```

The GTBZ transaction-proxy class retrieves a TheaterRecord when passed a ZipcodeRecord. It does this by invoking its GetTheaterRecord() method, which takes a single input record (a zipcode value) and returns a single output record (a theater located in that zipcode).

Both ZipcodeRecord and TheaterRecord are generated classes that represent COBOL data. Their attributes (ZipcodeRecord.value, TheaterRecord.TheaterName) are understood by the UDS application as simple data structures. The APPC library, which is referenced by methods in the generated classes, handles the conversion of COBOL datatypes into and from TOOL datatypes.

Fscript Commands

You use the Fscript commands described in Table 4-1 to develop transaction adapters.

To use any of these commands you first have to use the `CommandSet` command:

```
fscript > CommandSet appc
```

appc is the internal name of the Transaction Adapter Builder Fscript extensions. Once you have used `CommandSet` to enable the *appc* extensions, you can enter "help" to see a list of the *appc* commands.

Table 4-1 Fscript commands for Transaction Adapter Builder

Command	Description	See...
AddAPPCInfo	Specifies information about the connection between UDS application and COBOL application.	page 73
AddExchange	Adds a new exchange method to the current transaction proxy.	page 76
AddInputArgument	Adds an input argument to the method signature of the current exchange method.	page 78
AddOutputArgument	Adds an output argument to the method signature of the current exchange method.	page 78
AddRecord	Adds a new COBOL record declaration.	page 79
AddTransactionProxy	Adds a new transaction proxy and set it as the current transaction proxy.	page 73
FindExchange	Sets the specified exchange method to be the current exchange method.	page 77
FindTransactionProxy	Sets the specified transaction proxy to be the current transaction proxy.	page 74
GenerateTransactionProxy	Processes the current transaction proxy and generates a `.PEX` file and a `.SCR` file from it.	page 74

Table 4-1 Fscript commands for Transaction Adapter Builder *(Continued)*

Command	Description	See...
RemoveArgument	Deletes the specified argument from the method signature of the current exchange method.	page 79
RemoveExchange	Deletes the specified exchange method from the current transaction proxy.	page 77
RemoveRecord	Deletes a COBOL record declaration from memory.	page 81
RemoveTransactionProxy	Deletes the specified transaction proxy from memory.	page 75
ShowAllTransactionProxies	Displays a list of all transaction proxies currently in memory.	page 75
ShowRecords	Displays the current record.	page 81
ShowTransactionProxy	Displays the current transaction proxy and all arguments passed to its exchange methods.	page 76
SwitchTruncOption	Toggles between “standard” and “binary” modes of numeric data storage in COBOL application.	page 83
UserServiceObject	Specifies that Transaction Adapter Builder should not create a new service object when generating the current transaction proxy, but should use specified service object.	page 84

The following sections describe each of these commands in detail. The commands are organized into the following sections:

- Transaction proxies
- Exchange methods
- Arguments
- Records
- Other commands

Transaction Proxies

This section describes commands used for creating and removing transaction proxies, as well as for generating TOOL project plans from Fscript scripts.

AddTransactionProxy

```
AddTransactionProxy name
```

The `AddTransactionProxy` command adds a new transaction proxy and sets it as the current transaction proxy. The new transaction proxy is kept in memory and is written to disk only when you use the `GenerateTransactionProxy` command.

You specify a name for the transaction proxy with the *name* argument. This name is given to the TOOL class generated from your transaction proxy.

Example:

```
fscript > AddTransactionProxy GetEmpRecord
```

This command would create a new transaction proxy GetEmpRecord and set it as the current transaction proxy. When you use the `GenerateTransactionProxy` command on GetEmpRecord a TOOL class called “GetEmpRecord” is created. This class contains all the exchange methods you have defined for the transaction proxy.

FindTransactionProxy

`FindTransactionProxy` *`name`*

The `FindTransactionProxy` command makes the specified transaction proxy the current transaction proxy.

Example:

```
fscript > FindTransactionProxy GetEmpRecord
```

GetEmpRecord is now the current transaction proxy.

GenerateTransactionProxy

`GenerateTransactionProxy [`*`name`*`]`

The `GenerateTransactionProxy` command processes the current transaction proxy and creates two files:

- An .SCR script file containing the Fscript commands you used to create the transaction proxy, that you can use at a later time to recreate the transaction proxy
- A .PEX file containing the generated TOOL code that implements the transaction proxy

The base name for both files is the name you used for your transaction proxy. For example, if you had named your transaction proxy “GetEmpRecord” the `GenerateTransactionProxy` command will create the files `GetEmpRecord.scr` and `GetEmpRecord.pex`. Both files are written to the current working directory. (By default this is the directory in which your iPlanet UDS executables reside, typically `FORTE_ROOT/install/bin`. However, if you have explicitly changed directories, the files are written to the current working directory.)

If you want to use different names for these files you can specify an alternate name with the optional *name* argument.

When you execute the `GenerateTransactionProxy` command an `.SCR` file is always created. However, a `.PEX` file can be created only if you have completely described the transaction proxy. If, for example, your transaction proxy contains arguments of types you have not declared the Transaction Adapter Builder raises an error indicating that you need to use the `AddRecord` command to declare those datatypes.

For information on generated classes that are created when you generate a transaction proxy, see "Generated Classes" on page 84

RemoveTransactionProxy

`RemoveTransactionProxy` *`name`*

The `RemoveTransactionProxy` command removes a proxy while it is still in memory and before you have generated it. It takes one argument *name* that specifies which transaction proxy you want to remove.

Example:

```
fscript > RemoveTransactionProxy GetEmp
```

This command would remove the GetEmp proxy, assuming you hadn't already generated it with the `GenerateTransactionProxy` command.

ShowAllTransactionProxies

`ShowAllTransactionProxies`

The `ShowAllTransactionProxies` command displays a list of all transaction proxies currently in memory, as well as the side profile information you entered with the `AddAPPCInfo` command.

Example:

```
fscript > ShowAllTransactionProxies
GetEmp
GetEmpRecord
DeleteEmpRecord
CalcEmpSal
fscript >
```

ShowTransactionProxy

`ShowTransactionProxy`

The `ShowTransactionProxy` command displays the name of the current transaction proxy and all arguments passed to its exchange methods.

Example:

```
fscript > ShowTransactionProxy
Name: GetEmpRecord
Side Profile:
Transaction Program: Payroll
Logical Unit: HRLU
Mode: HRSNA
Sync Setting: none
1 data exchange(s)
GetEmpRecord(input number:EmpNo, input text: Name, input binary:
DeptNo, output number: EmpCode)
fscript >
```

Exchange Methods

This section describes commands for creating and removing exchange methods. An exchange is any point in your COBOL application where data must be passed to and from your UDS application, and an exchange method is used to establish an exchange.

AddExchange

`AddExchange` *`name`*

The `AddExchange` command adds a new exchange method to the current transaction proxy. `AddExchange` takes only one argument *name* with which you specify the name of the method.

Example:

```
fscript > AddExchange UpdateEmpRecord
```

This command adds the method UpdateEmpRecord() to the current transaction proxy. At this point the method has no input or output arguments: these must be added with the commands `AddInputArgument` (see "AddInputArgument" on page 78) and `AddOutputArgument` (see "AddOutputArgument" on page 78).

FindExchange

`FindExchange` *`name`*

The `FindExchange` command sets the specified exchange method to be the current exchange method.

Example:

```
fscript > FindExchange DeleteEmpRecord
```

DeleteEmpRecord is now the current exchange method.

RemoveExchange

`RemoveExchange` *`name`*

The `RemoveExchange` command removes the specified exchange method from the current transaction proxy, while it is still in memory. If the `GenerateTransactionProxy` command has already been invoked on the current transaction proxy `RemoveExchange` will not remove the exchange from either the `.SCR` or `.PEX` files.

Example:

```
fscript > RemoveExchange UpdateEmpRekord
```

This command removes the UpdateEmpRekord method from the current transaction proxy. You could then use `AddExchange` to add the correctly spelled exchange:

```
fscript > AddExchange UpdateEmpRecord
```

Arguments

This section describes commands for adding and deleting arguments for method signatures.

AddInputArgument

`AddInputArgument` *`name datatype`*

The `AddInputArgument` command adds an input argument to the method signature of the current exchange method. The first parameter *name* specifies an arbitrary name for the input argument. Your UDS application will reference this argument by this name in transactions with your COBOL application. The second parameter *datatype* is the datatype of the new argument. Datatypes are declared with the `AddRecord` command (see "AddRecord" on page 79). The type names used with `AddInputArgument` must match those declared with `AddRecord`.

Example:

```
fscript > FindExchange Query
fscript > AddInputArgument EmpNo EmployeeNumber
```

This command adds an input argument of type EmployeeNumber to the current exchange:

```
Query(input EmpNo: EmployeeNumber)
```

AddOutputArgument

`AddOutputArgument` *`name datatype`*

The `AddOutputArgument` command adds an output argument to the method signature of the current exchange method. The first parameter *name* specifies an arbitrary name for the output argument. Note that the name specified here will be the name by which your UDS application will reference this argument in transactions with your COBOL application. The second parameter *datatype* specifies the datatype of the new output argument. Datatypes are declared with the `AddRecord` command (see "AddRecord" on page 79). Note that the type names used with `AddOutputArgument` must match those declared with `AddRecord`.

Example:

```
fscript > FindExchange Query
fscript > AddOutputArgument EmpRec EmployeeRecord
```

This command adds an output argument of type *record* to the current method signature:

```
Query (input EmpNo: EmployeeNumber, output EmpRec:
EmployeeRecord)
```

RemoveArgument

`RemoveArgument` *`argument`*

The `RemoveArgument` command deletes the specified argument from the current exchange method while it is still in memory. This can be useful if you have mistyped an argument name and want to replace it with a correctly spelled one. (Once you have used the `GenerateTransactionProxy` command you cannot remove an argument from the `.SCR` or `.PEX` files with the `RemoveArgument` command.)

Example:

```
fscript > FindExchange Query
fscript > RemoveArgument EmppRec
```

This command would remove the *EmppRec* argument from the Query exchange method. You could then use the `AddOutputArgument` command to add the correct argument name:

```
fscript > AddOutputArgument EmpRec EmployeeRecord
```

Your method signature would now be:

```
Query (input EmpNo: EmployeeNumber, output EmpRec:
EmployeeRecord)
```

Records

This section describes commands you use to add and remove COBOL record declarations.

AddRecord

`AddRecord` *`name`*

The `AddRecord` command adds a new COBOL record declaration named with the value you give to the *name* argument. You need to declare a COBOL record before you can add arguments of its type. (Be sure to spell and capitalize the record names correctly when you add arguments of each type. The Fscript parser does not recognize misspelled or mis-capitalized record types.)

NOTE The UDS Transaction Adapter Builder supports only textual data.

When you use the `AddRecord` command Fscript switches to "line-entry" mode so you can enter COBOL source code declarations. If you are using Fscript interactively you terminate line-entry mode by entering the keyword "endrecord." If Fscript is reading the declarations from a script include-file line-entry mode is terminated when Fscript reaches the end of the include file.

NOTE When you enter records interactively you need to terminate *each* record with the endrecord keyword. If you forget to do this the Fscript parser becomes confused and does not know whether the following lines are meant as part of the record or not.

Example:

```
fscript > AddRecord FirstName
fscript > 01 FName PIC x(12).
fscript > endrecord
fscript > AddRecord LastName
fscript > 01 LName PIC x(24).
fscript > endrecord
fscript >
```

For records with multiple datatypes you only use the endrecord keyword after you have entered the entire record with all its types:

```
fscript > AddRecord EmpRec
fscript > 01 EmpRec
fscript > 02 EmpNo PIC x(5).
fscript > 02 EmpLName PIC x(24).
fscript > 02 EmpFName PIC x(12).
fscript > 02 DeptNo PIC x(5).
fscript > 02 Mgr PIC x(24).
fscript > 02 HireDate PIC x(8).
fscript > 02 Salary PIC x(6).
fscript > 03 BonusRate PIC x(5).
fscript > 03 Commission PIC x(4).
fscript > endrecord
fscript >
```

If you have a record with a lot of types you may want to create an *include* file containing the declaration instead of entering it interactively.

RemoveRecord

`RemoveRecord` *`name`*

The `RemoveRecord` command deletes a previously added COBOL record declaration from memory. You specify the record to be deleted with the *name* argument.

Example:

```
fscript > RemoveRecord EmpRec
```

This command would delete the EmpRec record from memory.

ShowRecords

`ShowRecords`

The ShowRecords command displays the current record.

Example:

```
fscript > ShowRecords
EmpRec
fscript >
```

Other Commands

This section describes other useful Fscript commands that don’t fit neatly into any of the previous categories.

AddAPPCInfo

`AddAPPCInfo [profile=`*profile_name*` | tp=`*tp_name*` | lu=`*lu_value*` | mode=`*mode_value*`]`

You use the `AddAPPCInfo` command to specify information about the connection between your UDS application and your COBOL application.

On OS/390 systems information about connecting to OLTP programs is stored in a “side profile,” which is contained in a system file called `SYS1.APPCSI`. The side profile contains information about connecting to the entire OLTP system as well as to specific programs within the overall system.

Transaction adapters connect to OS/390 systems by calling the APPC library Open() method, which uses the information you specify with the `AddAPPCInfo` command to locate the specific program to which you want to connect.

If you want to use the default connection information stored in the side profile you need use only the *profile* argument. If you do not use the *profile* argument you *must use all three of the other arguments* to specify connection information.

`AddAPPCInfo` takes the four following arguments:

- *profile* specifies the name of a side profile from which your UDS application should get connection information. If you want to use the default values in the side profile use *only* the profile argument. You can override specific defaults by using the other three arguments.
- *tp* specifies the name of a particular transaction program within the larger OLTP system. A transaction program name specified with the *tp* argument overrides the default transaction program name from the *profile* argument.
- *lu* specifies the LU6.2 logical unit name of the OS/390 OLTP system, and is used to override the default value specified with the *profile* argument.
- *mode* specifies the SNA mode entry name, and is used to override the default value specified with the *profile* argument.

Arguments and their values are separated from other arguments with a single blank space. You need only list those arguments for which you have values.

Example:

```
fscript > AddAPPCInfo profile=TOOLPROF
```

This command would specify that your UDS application should connect to the OS/390 system using the default information in the `TOOLPROF` side profile.

Example:

```
fscript > AddAPPCInfo tp=STOCKTP1 lu=STOCKLU mode=STOCKSNA
```

This command would specify that your UDS application should connect to the OS/390 system by overriding the default values in the side profile for *tp*, *lu*, and *mode* with the values STOCKTP1, STOCKLU, and STOCKSNA respectively.

CommandSet

`CommandSet` *`name`*

The `CommandSet` command enables your current Fscript session to use the Transaction Adapter Builder commands. (It also makes online help on those commands available in Fscript, which you invoke by typing “help” at the Fscript prompt.)

NOTE Until you have used the CommandSet command with “appc” as the value of the *name* argument you cannot use or see any of the Transaction Adapter Builder commands.

Example:

```
fscript > CommandSet appc
fscript >
```

This command makes the APPC commands available in your current Fscript session.

SwitchTruncOption

`SwitchTruncOption`

The `SwitchTruncOption` command toggles between the standard (“TRUNC(STD)”) and binary (“TRUNC(BIN)”) modes of numeric data storage in your COBOL programs.

Depending on how the COBOL compiler’s TRUNC option was set when your COBOL program code was compiled, numeric data are stored in either “standard” mode or “binary” mode. (“Standard” mode is the default.) The `SwitchTruncOption` command lets you toggle between these two modes.

It is likely that your COBOL code was compiled in standard mode, in which case you need not use the `SwitchTruncOption` command. However, if you either know for sure that your code was compiled in binary mode or you get errors raised at runtime when using a generated transaction proxy class, try using the `SwitchTruncOption` command.

Example:

```
fscript > SwitchTruncOption
fscript >
```

With this command the current mode will be switched to the other mode.

UseServiceObject

`UserServiceObject` *`name`*

The `UseServiceObject` enables you to specify that the Transaction Adapter Builder should not generate a new service object for your transaction adapter, but instead should use an existing one specified with the *name* argument.

By default, the Transaction Adapter Builder creates a new service object when you generate a transaction proxy, and names it “appcSO.” If you want to use an existing service object from another iPlanet UDS project plan, and do not want to create a new one for your generated classes, use the `UseServiceObject` command to specify the existing service object you want to use.

Example:

```
fscript > UseServiceObject SomeOtherPlan.SomeOtherServiceObject
```

This command would specify that the classes generated for your transaction proxy will use your service object “SomeOtherPlan.SomeOtherServiceObject” instead of having Transaction Adapter Builder create a new service object.

Generated Classes

When you use the `GenerateTransactionProxy` command two kinds of generated classes are created: one for each transaction proxy, and one for each COBOL record.

Transaction proxy classes contain methods for each exchange you create, and each of these methods takes the COBOL record classes you created as its attributes.

When you create an exchange you specify its input and output parameters. The methods generated from your script take these parameters as their attributes. Suppose you created an exchange called “GetEmpNo” with the input parameter datatypes EmpLName and EmpFName and the output parameter datatype EmpNo. From this exchange a method would be generated, with the signature:

```
...
has public method GetEmpNo(input text1: EmpLName,input text2:
EmpFName,output number: EmpNo);
...
```

Each of these parameter datatypes must have already been declared as a COBOL record in your script.

Supporting Classes

The TOOL code generated from your script uses the classes and methods from two iPlanet UDS libraries for low-level functionality:

- The Framework library
- The APPC library

Refer to the iPlanet UDS online help for information on these libraries.

Sample Script and Generated Code

The following sample script illustrates a simple transaction proxy called EmployeeRecord that contains one exchange GetEmpRec. This exchange takes one input argument EmpRec and returns one output argument EmpNo. Following this script is an output sample illustrating the TOOL code that would be generated by executing this script.

Code Example 4-1 Sample script illustrating a simple transaction proxy

```
# Generated by Mainframe Transaction Adapter version 1.0
# Generated script file that re-creates 'EmployeeRecord'
AddTransactionProxy EmployeeRecord
AddAPPCInfo profile=PAYROLL mode=HRSNA tp=HR03 lu=HRLU
AddExchange GetEmpRec
AddInputArgument EmpRec EmployeeNumber
AddOutputArgument EmpNo EmployeeRecord
UseServiceObject appcSO
AddRecord EmployeeNumber
1 EmployeeNumber.
2 EMPNO PIC X(5).
endrecord
AddRecord EmployeeRecord
1 EmployeeRecord.
2 LNAME PIC X(24).
2 FNAME PIC X(12).
2 NUM PIC X(5).
2 PNUM PIC X(5).
2 EMPLOC PIC X(9).
2 DEPT PIC X(12).
2 MGR PIC X(36).
2 STARTDATE PIC X(8).
2 TITLE PIC X(20).
2 SALARY PIC X(6).
2 BONUS PIC X(5).
2 COMRATE PIC X(2).
endrecord
```

The output produced by using the `GenerateTransactionProxy` command on the script is illustrated below.

.

Code Example 4-2 Output from sample script *(page 1 of 9)*

```
-- Generated by Mainframe Transaction Adapter version 1.0
begin TOOL EmployeeRecord;
-- GENERATED FILE - DO NOT MODIFY

includes Framework;
includes APPC;

HAS PROPERTY IsLibrary = FALSE;

-- START SERVICE OBJECT DEFINITIONS
service appcSO:APPC.APPCApi = (DialogDuration = SESSION,
  Visibility = environment,
  FailOver = FALSE,
  LoadBalance = FALSE);
-- END SERVICE OBJECT DEFINITIONS

-- START FORWARD CLASS DECLARATIONS

forward EmployeeRecord;
forward EmployeeNumber;
forward EmployeeNumber_Impl;
forward EmployeeRecord;
forward EmployeeRecord_Impl;
-- END FORWARD CLASS DECLARATIONS


-- START CLASS DEFINITIONS

class EmployeeRecord inherits from Framework.Object
has property
  shared=(allow=off, override=on);
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=on, override=on);
has public method Open(input dest:Framework.string='PAYROLL',input
lu:Framework.string='HRLU',input tp:Framework.string='HR03',input
mode:Framework.string='HRSNA',input security:APPC.APPCSecurityInfo=NIL);
has public method Init();
has public method GetEmpRec(input EmpRec:EmployeeNumber,output
EmpNo:EmployeeRecord);
has public method Close(input normal:Framework.Boolean=TRUE);
has public attribute Conv:APPC.APPCConversation;
has public attribute TraceActive:Framework.Boolean;
has public attribute Log:Framework.LogMgr;
has public attribute IsOpen:Framework.Boolean;
end class;
class EmployeeNumber inherits from Framework.Object
has property
  shared=(allow=off, override=on);
```

Code Example 4-2 Output from sample script *(page 2 of 9)*

```
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=on, override=on);
has public method Init();
has public attribute _private:EmployeeNumber_Impl has property extended =
(XMLServerIgnore='true');
has public attribute EMPNO:TextData;
end class;
class EmployeeNumber_Impl inherits from Framework.Object
has property
  shared=(allow=off, override=on);
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=on, override=on);
has public method SetValues();
has public method Init();
has public method GetValues();
has public method Construct(input
implementedRecord:EmployeeNumber):EmployeeNumber_Impl;
has public method BinaryData():Framework.BinaryData;
has public attribute EmployeeNumber:EmployeeNumber;
has public attribute _buffer:COBOLBuffer;
has public attribute EMPNO:COBOLField;
end class;
class EmployeeRecord inherits from Framework.Object
has property
  shared=(allow=off, override=on);
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=on, override=on);
has public method Init();
has public attribute _private:EmployeeRecord_Impl has property extended =
(XMLServerIgnore='true');
has public attribute LNAME:TextData;
has public attribute FNAME:TextData;
has public attribute NUM:TextData;
has public attribute PNUM:TextData;
has public attribute EMPLOC:TextData;
has public attribute DEPT:TextData;
has public attribute MGR:TextData;
has public attribute STARTDATE:TextData;
has public attribute TITLE:TextData;
has public attribute SALARY:TextData;
has public attribute BONUS:TextData;
has public attribute COMRATE:TextData;
end class;
class EmployeeRecord_Impl inherits from Framework.Object
has property
  shared=(allow=off, override=on);
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=on, override=on);
has public method SetValues();
```

Code Example 4-2 Output from sample script *(page 3 of 9)*

```
has public method Init();
has public method GetValues();
has public method Construct(input
implementedRecord:EmployeeRecord):EmployeeRecord_Impl;
has public method BinaryData():Framework.BinaryData;
has public attribute EmployeeRecord:EmployeeRecord;
has public attribute _buffer:COBOLBuffer;
has public attribute LNAME:COBOLField;
has public attribute FNAME:COBOLField;
has public attribute NUM:COBOLField;
has public attribute PNUM:COBOLField;
has public attribute EMPLOC:COBOLField;
has public attribute DEPT:COBOLField;
has public attribute MGR:COBOLField;
has public attribute STARTDATE:COBOLField;
has public attribute TITLE:COBOLField;
has public attribute SALARY:COBOLField;
has public attribute BONUS:COBOLField;
has public attribute COMRATE:COBOLField;
end class;
method EmployeeRecord.Open(input dest:Framework.string='PAYROLL',input
lu:Framework.string='HRLU',input tp:Framework.string='HR03',input
mode:Framework.string='HRSNA',input security:APPC.APPCSecurityInfo=NIL)
begin
self.Conv = appcSO.NewConversation();
if self.TraceActive then
  self.Log.Put( 'Opening conversation.\n');
end if;
self.Conv.Open(
  dest=dest,
  lu=lu,
  mode=mode,
  tp=tp,
  security=security,
  synclevel=0
 );
self.IsOpen = true;
if self.TraceActive then
  self.Log.Put( 'Conversation opened.\n' );
end if;
end method;
method EmployeeRecord.Init()
begin
super.Init();
self.Conv = NIL;
self.TraceActive = FALSE;
self.Log = task.Part.LogMgr;
self.IsOpen = false;
end method;
method EmployeeRecord.GetEmpRec(input EmpRec:EmployeeNumber,output
EmpNo:EmployeeRecord)
begin
bits : BinaryData;
```

Code Example 4-2 Output from sample script *(page 4 of 9)*

```
len : Integer;
begin

-- This code handles the ’EmpRec’ parameter.
  EmpRec._private.SetValues();
  bits = EmpRec._private.BinaryData();
  len = bits.ActualSize;
  if self.TraceActive then
    self.Log.Put(’Sending ’);
    self.Log.Put(len);
    self.Log.Put(’ bytes\n’);
  end if;
  self.Conv.Write(dataBuffer=bits,writeLength=len);

-- This code handles the ’EmpNo’ parameter.
  EmpNo = new;
  bits = EmpNo._private.BinaryData();
  len = bits.ActualSize;
  if self.TraceActive then
    self.Log.Put(’Receiving ’);
    self.Log.Put(len);
    self.Log.Put(’ bytes\n’);
  end if;
  self.Conv.Read(dataBuffer=bits,readLength=len);
  if self.TraceActive then
    self.Log.Put(’Received ’);
    self.Log.Put(len);
    self.Log.Put(’ bytes: "’);
  self.Log.Put(bits);
    self.Log.Put(’"\n’);
  end if;
  EmpNo._private.GetValues();

  exception when e:GenericException do begin
    self.Close( normal=false );
    raise;
  end;

end;
end method;
method EmployeeRecord.Close(input normal:Framework.Boolean=TRUE)
begin
if not self.IsOpen then
  return;
end if;
if normal then
  self.Conv.Close( type=APPC.APPC_CLOSE_NORMAL );
  if self.TraceActive then
    self.Log.Put( ’Conversation closed normally.\n’ );
  end if;
else
  self.Conv.Close( type=APPC.APPC_CLOSE_ABNORMAL );
  if self.TraceActive then
    self.Log.Put( ’Conversation closed abnormally.\n’ );
```

Code Example 4-2 Output from sample script *(page 5 of 9)*

```
  end if;
end if;
self.IsOpen = false;
end method;
method EmployeeNumber.Init()
begin
super.Init();
_private = EmployeeNumber_Impl().Construct( implementedRecord=self );
EMPNO = new;
end method;
method EmployeeNumber_Impl.SetValues()
begin
EMPNO.PutValue( buffer=_buffer, source=EmployeeNumber.EMPNO );
end method;
method EmployeeNumber_Impl.Init()
begin
super.Init();
_buffer = new;
_buffer.SetBufferAttribute(
  serverCodeset=APPC.COB_CODESET_EBCDIC,
  serverPlatform=APPC.COB_PLATFORM_S390,
  compileTruncOption=APPC.COB_COMPILER_TRUNC_STD
 );
_buffer.SetAllocatedSize(5);
_buffer.ActualSize = 5;
EMPNO = new;
EMPNO.Define(
  location=0,
  picture='X(5)',
  usage='',
  sync=APPC.COB_NO_SYNC,
  flags=0,
  maxOccurs=1
 );
end method;
method EmployeeNumber_Impl.GetValues()
begin
EMPNO.GetValue( buffer=_buffer, target=EmployeeNumber.EMPNO );
end method;
method EmployeeNumber_Impl.Construct(input
implementedRecord:EmployeeNumber):EmployeeNumber_Impl
begin
EmployeeNumber = implementedRecord;
return self;
end method;
method EmployeeNumber_Impl.BinaryData():Framework.BinaryData
begin
return _buffer;
end method;
method EmployeeRecord.Init()
begin
super.Init();
_private = EmployeeRecord_Impl().Construct( implementedRecord=self );
LNAME = new;
```

Code Example 4-2 Output from sample script *(page 6 of 9)*

```
FNAME = new;
NUM = new;
PNUM = new;
EMPLOC = new;
DEPT = new;
MGR = new;
STARTDATE = new;
TITLE = new;
SALARY = new;
BONUS = new;
COMRATE = new;
end method;
method EmployeeRecord_Impl.SetValues()
begin
LNAME.PutValue( buffer=_buffer, source=EmployeeRecord.LNAME );
FNAME.PutValue( buffer=_buffer, source=EmployeeRecord.FNAME );
NUM.PutValue( buffer=_buffer, source=EmployeeRecord.NUM );
PNUM.PutValue( buffer=_buffer, source=EmployeeRecord.PNUM );
EMPLOC.PutValue( buffer=_buffer, source=EmployeeRecord.EMPLOC );
DEPT.PutValue( buffer=_buffer, source=EmployeeRecord.DEPT );
MGR.PutValue( buffer=_buffer, source=EmployeeRecord.MGR );
STARTDATE.PutValue( buffer=_buffer, source=EmployeeRecord.STARTDATE );
TITLE.PutValue( buffer=_buffer, source=EmployeeRecord.TITLE );
SALARY.PutValue( buffer=_buffer, source=EmployeeRecord.SALARY );
BONUS.PutValue( buffer=_buffer, source=EmployeeRecord.BONUS );
COMRATE.PutValue( buffer=_buffer, source=EmployeeRecord.COMRATE );
end method;
method EmployeeRecord_Impl.Init()
begin
super.Init();
_buffer = new;
_buffer.SetBufferAttribute(
  serverCodeset=APPC.COB_CODESET_EBCDIC,
  serverPlatform=APPC.COB_PLATFORM_S390,
  compileTruncOption=APPC.COB_COMPILER_TRUNC_STD
 );
_buffer.SetAllocatedSize(144);
_buffer.ActualSize = 144;
LNAME = new;
LNAME.Define(
  location=0,
  picture='X(24)',
  usage='',
  sync=APPC.COB_NO_SYNC,
  flags=0,
  maxOccurs=1
 );
FNAME = new;
FNAME.Define(
  location=24,
  picture='X(12)',
  usage='',
  sync=APPC.COB_NO_SYNC,
  flags=0,
```

Code Example 4-2 Output from sample script *(page 7 of 9)*

```
    maxOccurs=1
   );
  NUM = new;
  NUM.Define(
    location=36,
    picture='X(5)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  PNUM = new;
  PNUM.Define(
    location=41,
    picture='X(5)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  EMPLOC = new;
  EMPLOC.Define(
    location=46,
    picture='X(9)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  DEPT = new;
  DEPT.Define(
    location=55,
    picture='X(12)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  MGR = new;
  MGR.Define(
    location=67,
    picture='X(36)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  STARTDATE = new;
  STARTDATE.Define(
    location=103,
    picture='X(8)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
```

Code Example 4-2 Output from sample script *(page 8 of 9)*

```
    maxOccurs=1
   );
  TITLE = new;
  TITLE.Define(
    location=111,
    picture='X(20)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  SALARY = new;
  SALARY.Define(
    location=131,
    picture='X(6)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  BONUS = new;
  BONUS.Define(
    location=137,
    picture='X(5)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  COMRATE = new;
  COMRATE.Define(
    location=142,
    picture='X(2)',
    usage='',
    sync=APPC.COB_NO_SYNC,
    flags=0,
    maxOccurs=1
   );
  end method;
  method EmployeeRecord_Impl.GetValues()
  begin
  LNAME.GetValue( buffer=_buffer, target=EmployeeRecord.LNAME );
  FNAME.GetValue( buffer=_buffer, target=EmployeeRecord.FNAME );
  NUM.GetValue( buffer=_buffer, target=EmployeeRecord.NUM );
  PNUM.GetValue( buffer=_buffer, target=EmployeeRecord.PNUM );
  EMPLOC.GetValue( buffer=_buffer, target=EmployeeRecord.EMPLOC );
  DEPT.GetValue( buffer=_buffer, target=EmployeeRecord.DEPT );
  MGR.GetValue( buffer=_buffer, target=EmployeeRecord.MGR );
  STARTDATE.GetValue( buffer=_buffer, target=EmployeeRecord.STARTDATE );
  TITLE.GetValue( buffer=_buffer, target=EmployeeRecord.TITLE );
  SALARY.GetValue( buffer=_buffer, target=EmployeeRecord.SALARY );
  BONUS.GetValue( buffer=_buffer, target=EmployeeRecord.BONUS );
  COMRATE.GetValue( buffer=_buffer, target=EmployeeRecord.COMRATE );
  end method;
```

Code Example 4-2 Output from sample script *(page 9 of 9)*

```
method EmployeeRecord_Impl.Construct(input
implementedRecord:EmployeeRecord):EmployeeRecord_Impl
begin
EmployeeRecord = implementedRecord;
return self;
end method;
method EmployeeRecord_Impl.BinaryData():Framework.BinaryData
begin
return _buffer;
end method;
end EmployeeRecord;
```

Example Applications

This appendix describes the example applications that are provided with the iPlanet UDS Transaction Adapter for OS/390. Included are the following examples:

- CICS VSAM query transaction with an APPC-only interface
- CICS DB2 query transaction with both BMS 3270 and APPC interfaces using the AOR/TOR concept
- IMS DLI query transaction with an MFS 3270 interface
- IMS DLI query transaction with an explicit APPC interface using CPI-C

The example applications consist of two sets of files, one set for the client system and one set for the OLTP systems. Client example application files are imported into your iPlanet UDS environment using the developer workshop. The client example files are in the FORTE_ROOT/install/examples/extsys/txadapt directory of the iPlanet UDS installation system. The OLTP example application files are installed into two partitioned datasets on the OS/390 system. The source files are stored as members in the FORTE.V30M1.SOURCE dataset, and the JCL and assorted other files are stored as members in the FORTE.V30M1.JCL dataset.

CICS VSAM Query Example

The CICS VSAM query example application shows the use of the Transaction Adapter to invoke a simple CICS transaction that takes a customer number as input, queries a VSAM file, and returns a customer record as output. This application uses the CICS sample VSAM file, FILEA, provided by IBM with the CICS product set. The CICS transaction is designed to be invoked only across an APPC conversation. The purpose of this example is to show how simple the CICS APPC interface is to use.

The application consists of the FRTCEX02 CICS program and the CICSvsamQuery iPlanet UDS project. The FRTCEX02 program is a COBOL program that implements the CICS transaction FR02 and uses the EXEC CICS interface for its APPC operations. The CICSvsamQuery project provides the iPlanet UDS client interface to the CICS FR02 transaction.

Installing the Example Application

CICS Transaction

Table 4-2 lists the sources and JCLs you will need for the FRTCEX02 program.

Table 4-2 Sources and JCLs for FRTCEX02 program

Sources and JCLs	Location
Source for the FRTCEX02 program	FORTE.V30M1.SOURCE(FRTCEX02)
JCL to compile and link program	FORTE.V30M1.JCL(FRTCEX02)
JCL to build the CICS definitions for the transaction	FORTE.V30M1.JCL(CSDCEX02)

The JCL files listed in Table 4-2 contain comments that detail the modifications that must be made to tailor the JCL for your site.

At many sites, the definition of new CICS transactions requires the involvement of the CICS system administrator. If your site requires this, have the CICS system administrator perform the definition of the CICS transaction and the installation of the CICS sample VSAM file.

To define the CICS transaction, first tailor the JCL in CSDCEX02 and submit the job to define the FRTCEX02 program and FR02 transaction to CICS. This job uses the CICS CSD update utility to build the definitions. Alternatively, you can manually enter these definitions into CICS using the CEDA transaction. After the definitions have been built, use CEDA to install the group to which the definitions were added.

Next, make sure that the CICS sample VSAM file, FILEA, has been installed and activated on your CICS system. Refer to the IBM CICS manuals for information on the installation of FILEA.

To prepare the FRTCEX02 program, compile and link the program by tailoring the JCL in FRTCEX02 and submitting the job. Once the job has successfully completed, the CICS transaction is ready to use.

iPlanet UDS Application

The project export of the CICSvsamQuery project is in the cicsvsam.pex file. This project should be imported into your workspace using the workshop, and then deployed as a client on the machines where you wish to run the application. The project deploys into two partitions, a client partition and a server partition containing the APPCApiSO service object. Make sure that the server partition is deployed on the OS/390 node where you installed the Transaction Adapter.

The CICSvsamQuery project contains three classes: FILEA describes the customer record as an object, QueryCics is the class that interfaces with the CICS transaction using the Transaction Adapter classes, and QueryWindow is the window class that provides the user interface.

This example assumes that the symbolic destination name, FRTCEXSI, has been defined on your system and accesses the CICS system where you have installed the CICS transaction. It also assumes that the CICS transaction name is FR02. If either of these names has been changed, you must modify the RunQuery method of the QueryCics class to specify the correct names. The code under the topic "Establishing the Conversation" on page 99 shows where the changes should be made.

Running the Example Application

➤ **To execute this example application**

1. Bring up the iPlanet UDS client GUI window.
2. Type a 6-digit customer number into the input area, and click the Query button.

   The following are some of the valid customer numbers for which records exist: 100000, 111111, 200000, 222222, 300000, 333333, 400000, 444444, 500000, 555555, 600000, 666666, 700000, 777777, 800000, 888888, 900000, and 999999. The IBM CICS documentation provides more complete information on the valid customer numbers for the FILEA sample file.
3. To terminate the application, close the GUI window.

Understanding the Application Logic

This section describes the logic for the communications between the iPlanet UDS client application and the CICS transaction. Each step in the execution is covered, with code fragments from both sides of the application included to show how the Transaction Adapter is used to interface with the CICS transaction program. The client application code fragments shown here are extracted from the RunQuery method of the QueryCics class. The CICS application code fragments shown here are extracted from the FRTCEX02 COBOL program.

Creating the APPCConversation Object

The first step in the iPlanet UDS application is to instantiate a new APPCConversation object:

```
conv = APPCApiSO.NewConversation();
```

Creating the COBOLBuffer and COBOLField Objects

The next step is to instantiate the COBOLBuffer object used to hold the VSAM data, and the COBOLField objects used to convert the fields into iPlanet UDS data:

```
buf : COBOLBuffer = new;
cobAddress: COBOLField = new;
cobAmount: COBOLField = new;
cobComment: COBOLField = new;
cobDate: COBOLField = new;
cobName: COBOLField = new;
cobNum: COBOLField = new;
cobPhone: COBOLField = new;
cobRecStat: COBOLField = new;
cobStatus: COBOLField = new;
```

Defining the Layout of the COBOL Record

Next, define the individual fields of the COBOL record using the COBOLField objects and define the characteristics of the COBOL program using the COBOLBuffer object as shown in the following code:

```
loc = cobStatus.Define(location=0,picture='X',usage='DISPLAY');
loc = cobNum.Define(location=loc,picture='X(6)',usage='DISPLAY');
loc = cobName.Define(location=loc,picture='X(20)',usage='DISPLAY');
loc = cobAddress.Define(location=loc,picture='X(20)', usage='DISPLAY');
loc = cobPhone.Define(location=loc,picture='X(8)',usage='DISPLAY');
loc = cobDate.Define(location=loc,picture='X(8)',usage='DISPLAY');
loc = cobAmount.Define(location=loc,picture='X(8)', usage='DISPLAY');
loc = cobComment.Define(location=loc,picture='X(9)', usage='DISPLAY');
loc = cobRecStat.Define(location=loc,picture='X',usage='DISPLAY');
reclen = loc;

buf.SetBufferAttribute(serverCodeset=COB_CODESET_EBCDIC,
       compileTruncOption= COB_COMPILER_TRUNC_STD,
        serverPlatform=COB_PLATFORM_S390);
```

Establishing the Conversation

The next step is to establish the APPC conversation between the iPlanet UDS application and the CICS transaction program. The following TOOL code shows how the client application requests a conversation with the CICS transaction:

```
dest = 'FRTCEXSI';     -- must match your symbolic destination name
tp = 'FR02';           -- must match your CICS transaction name
conv.Open(dest=dest,tp=tp);
```

The following CICS COBOL code accepts the conversation request and checks for errors:

```
EXEC CICS ASSIGN FACILITY(APPC-CONVID) RESP(CICS-RESP)
END-EXEC.
IF CICS-RESP NOT = DFHRESP(NORMAL) THEN
   MOVE 'ASSIGN FACILITY FAILED' TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

If an error is found, a message is displayed on the OS/390 console, because there is no way for a message to be sent back to the iPlanet UDS application without an APPC conversation.

Sending the Input Data

Now the input data must be sent from the iPlanet UDS application to the CICS transaction. In this application, the input data is a 6-digit customer number in character format. The following TOOL code shows how the client application sends the input data:

```
len = custRec.NUM_LENGTH;
conv.Write(dataBuffer=custRec.Num,writeLength=len);
```

The following CICS COBOL code receives the input data and checks for errors:

```
EXEC CICS IGNORE CONDITION LENGERR
END-EXEC.
MOVE LENGTH OF CUST-NUM TO IN-LENGTH.
EXEC CICS RECEIVE CONVID(APPC-CONVID)
                  INTO(IN-AREA)
                  LENGTH(IN-LENGTH)
                  STATE(APPC-STATE)
                  RESP(CICS-RESP)
END-EXEC.
IF (CICS-RESP NOT = DFHRESP(NORMAL) AND
    CICS-RESP NOT = DFHRESP(EOC)) OR
   (EIBERR = HIGH-VALUES) THEN
   MOVE ’FAILURE RECEIVING CUST-NUM’ TO DIAG-MSG
   PERFORM DIAG
ELSE
   IF EIBNODAT = HIGH-VALUES THEN
      MOVE ’RECEIVED NO CUST-NUM’ TO DIAG-MSG
      PERFORM DIAG
   END-IF
END-IF.
IF IN-LENGTH = LENGTH OF CUST-NUM THEN
   MOVE IN-AREA TO CUST-NUM
ELSE
   MOVE ’RECEIVED INCORRECT LENGTH FOR CUST-NUM’ TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

Receiving the Output Data

Now the output data must be received from the CICS transaction by the iPlanet UDS application. In this application, the output data is an 80-byte customer record in character format, plus a 1-byte record status code. The following TOOL code shows how the client application receives the output data:

```
len = reclen;
conv.Read(dataBuffer=buf,readLength=len);
```

The following CICS COBOL code sends the output data and checks for errors:

```
EXEC CICS SEND CONVID(APPC-CONVID)
               STATE(APPC-STATE)
               RESP(CICS-RESP)
               FROM(FILEA)
               LENGTH(LENGTH OF FILEA)
END-EXEC.
IF (CICS-RESP NOT = DFHRESP(NORMAL) AND
    CICS-RESP NOT = DFHRESP(EOC)) OR
   (EIBERR = HIGH-VALUES) THEN
   MOVE ’ERROR SENDING CUST-REC’ TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

Terminating the Conversation

The conversation with the CICS transaction is no longer needed by the iPlanet UDS application. The following TOOL code shows how the client application terminates the conversation:

```
conv.Close(type=CLOSE_NORMAL);
```

The following CICS COBOL code closes the APPC conversation from the CICS side:

```
EXEC CICS FREE CONVID(APPC-CONVID)
               STATE(APPC-STATE)
               RESP(CICS-RESP)
END-EXEC.
```

Processing the Data

The iPlanet UDS application can now use the data received from the CICS transaction. The following TOOL code shows how the client application extracts individual fields from the buffer:

```
cobStatus.GetValue(buffer=buf,target=custRec.Status);
cobName.GetValue(buffer=buf,target=custRec.Name);
cobAddress.GetValue(buffer=buf,target=custRec.Address);
cobPhone.GetValue(buffer=buf,target=custRec.Phone);
cobDate.GetValue(buffer=buf,target=custRec.Date);
cobAmount.GetValue(buffer=buf,target=custRec.Amount);
cobComment.GetValue(buffer=buf,target=custRec.Comment);
cobRecStat.GetValue(buffer=buf,target=custRec.RecStat);
```

In this example, the extraction of the individual fields is accomplished in the application by calling the COBOLField.GetValue method to extract each field into a TextData object. The necessary character set conversions are performed automatically.

CICS DB2 Query Example

The CICS DB2 query example application shows the use of the Transaction Adapter to invoke a CICS transaction that takes an employee number as input, queries a DB2 table, and returns an employee row as output. This application uses the DB2 sample employee table, EMP, provided by IBM with the DB2 product set.

This application is written using the CICS AOR/TOR concept, where the application logic and the user interface logic are split into separate programs. This example provides two versions of the user interface, one using CICS BMS 3270 support and one using CICS APPC support. The 3270 version of the application is provided to show how the transaction runs in a CICS-only environment. This section will not describe the 3270 version in detail. The purpose of this example is to show how simple it is to add an APPC front-end to an existing application that is already written for the AOR/TOR environment.

The application consists of the FRTCEX03, FRTCEX04, and FRTCEX05 CICS programs, and the CICSdb2Query iPlanet UDS project. The FRTCEX03 program is a COBOL program that implements the CICS transaction FR03 and uses the CICS BMS interface for communicating directly with a 3270 terminal. The FRTCEX04 program is a COBOL program that implements the application logic to query the DB2 EMP table. The FRTCEX05 program is a COBOL program that implements the CICS transaction FR05 and uses the CICS APPC interface for communicating with the Transaction Adapter. Both the FRTCEX03 and the FRTCEX05 programs call the FRTCEX04 program using a CICS DPL. The CICSdb2Query project provides the iPlanet UDS client interface to the FR05 CICS transaction.

Installing the Example Application

The DB2 system administrator should be involved in the steps for setting up DB2 to run this example application.

Make sure that the DB2 sample employee table, EMP, has been installed on your DB2 system and is accessible to your CICS transaction. Refer to the IBM DB2 manuals for information on the installation of the sample table.

The source for the FRTCEX04 program is shipped ready to use with DB2 Version 5. If you are using a different version of DB2, then you must modify the FRTCEX04 program. Specifically, the schema name for the DB2 sample EMP table is hard-coded in the source as DSN8510. For other versions of DB2 this should be changed to the correct schema name. The DB2 system administrator can provide the correct schema name. The actual compilation of the FRTCEX04 program is covered in the section, "CICS Transaction" on page 96.

The CICS DB2 interface requires a special table called a Resource Control Table (RCT) that contains entries for all CICS transactions that access DB2. An entry must be added to the RCT for each CICS transaction in this example application. In FORTE.V30M1.SOURCE(RCTCEX03), there is a sample entry for the FRTCEX03 program. In FORTE.V30M1.SOURCE(RCTCEX05), there is a sample entry for the FRTCEX05 program. These entries should be added to the RCT that is used for your CICS system, the RCT should be reassembled and relinked, and the CICS DB2 interface should be stopped and restarted before the example application is used. The DB2 or CICS system administrator usually does this.

CICS Transactions

Table 4-3 lists the sources and JCLs you will need for the FRTCEX03 program.

Table 4-3 Sources and JCLs for FRTCEX03 program

Source and JCLs	Location
Source for FRTCEX03 program	FORTE.V30M1.SOURCE(FRTCEX03)
JCL to compile and link program	FORTE.V30M1.JCL(FRTCEX03)
Source for BMS map for program	FORTE.V30M1.SOURCE(FRTCEM03)
JCL to assemble and link BMS map	FORTE.V30M1.JCL(FRTCEM03)
JCL to build CICS definitions for the transaction	FORTE.V30M1.JCL(CSDCEX03)

Table 4-4 lists the sources and JCLs you will need for the FRTCEX04 program.

Table 4-4 Sources and JCLs for FRTCEX04 program

Source and JCLs	Location
Source for FRTCEX04 program	FORTE.V30M1.SOURCE(FRTCEX04)
JCL to compile and link program	FORTE.V30M1.JCL(FRTCEX04)
JCL to build CICS definitions for the transaction	FORTE.V30M1.JCL(CSDCEX04)

Table 4-5 lists the sources and JCLs you will need for the FRTCEX05 program.

Table 4-5 Sources and JCLs for FRTCEX05 program

Source and JCLs	Location
Source for FRTCEX05 program	FORTE.V30M1.SOURCE(FRTCEX05)
JCL to compile and link program	FORTE.V30M1.JCL(FRTCEX05)
JCL to build CICS definitions for the transaction	FORTE.V30M1.JCL(CSDCEX05)

The JCL files for this example contain comments that detail the modifications that you must make to tailor the JCL for your site.

At many sites, the definition of new CICS transactions requires the involvement of the CICS system administrator. If your site requires this, have the CICS system administrator perform the definition of the CICS transactions.

To build the CICS BMS 3270 user interface transaction, first tailor the JCL in CSDCEX03 and submit the job to define the FRTCEX03 program, FR03 transaction, and FRTCEM03 BMS map to CICS. This job uses the CICS CSD update utility to build the definitions. Alternatively, you can manually enter these definitions into CICS using the CEDA transaction. Next, compile and link the FRTCEX03 program and FRTCEM03 BMS map by tailoring the JCL in FRTCEX03 and FRTCEM03, respectively, and submitting the jobs.

To build the CICS DB2 query application logic program, first tailor the JCL in CSDCEX04 and submit the job to define the FRTCEX04 program to CICS. This job uses the CICS CSD update utility to build the definition. Alternatively, you can manually enter this definition into CICS using the CEDA transaction. Next, compile and link the FRTCEX04 program by tailoring the JCL in FRTCEX04 and submitting the job. In addition to compiling and linking the program, this JCL contains a step to perform the necessary DB2 BIND and GRANT processing to allow the program to be used by both the FRTCEX03 and FRTCEX05 programs.

To build the CICS APPC user interface transaction, first tailor the JCL in CSDCEX05 and submit the job to define the FRTCEX05 program and FR05 transaction to CICS. This job uses the CICS CSD update utility to build the definitions. Alternatively, you can manually enter these definitions into CICS using the CEDA transaction. Next, compile and link the FRTCEX05 program by tailoring the JCL in FRTCEX05 and submitting the job.

After all of the CICS components of this example application have been built, use CEDA to install the group into which the CICS definitions were added. Once this has been done, the CICS transactions are ready to use.

iPlanet UDS Application

The project export of the CICSdb2Query project is in the cicsdb2.pex file. This project should be imported into your workspace using the workshop, and then deployed as a client on the machines where you wish to run the application. The project deploys into two partitions, a client partition and a server partition containing the APPCApiSO service object. Make sure that the server partition is deployed on the OS/390 node where you installed the Transaction Adapter.

The CICSdb2Query project contains three classes: EmpRec describes the employee table row as an object, QueryCics is the class that interfaces with the CICS transaction using the Transaction Adapter classes, and QueryWindow is the window class that provides the user interface.

This example assumes that the symbolic destination name FRTCEXSI has been defined on your system and accesses the CICS system where you have installed the CICS transaction. It also assumes that the CICS transaction name is FR05. If either of these names has been changed, you must modify the RunQuery method of the QueryCics class to specify the correct names. The code under the topic "Establishing the Conversation" on page 99 shows where the changes should be made.

Running the Example Application

To execute either of the application versions, you need to enter employee numbers.The following are some of the valid employee numbers for which rows exist: 000100, 000110, 000120, 000130, 000140, 000150, 000160, 000170, 000180, and 000190. The IBM DB2 documentation provides more complete information on the valid employee numbers for the EMP sample DB2 table.

➤ **To execute the CICS BMS version of this example application**

1. Log on to your CICS system.
2. Press the Clear key.
3. Type the following text and press Enter.

   ```
   FR03
   ```

   A formatted screen will appear.
4. Type in a 6-digit employee number and press Enter.

➤ **To execute the iPlanet UDS version of this example application**

1. Bring up the iPlanet UDS client GUI window.
2. Type a 6-digit employee number into the input area, and click the Query button.
3. To terminate the application, close the GUI window.

Understanding the Application Logic

This section describes the logic for the communications between the iPlanet UDS client application and the CICS transaction. Each step in the execution is covered, with code fragments from both sides of the application included to show how the Transaction Adapter is used to interface with the CICS transaction program. The client application code fragments shown here are extracted from the RunDb2q method of the QueryCics class. The CICS application code fragments shown here are extracted from the FRTCEX05 COBOL program.

Creating the APPCConversation Object

The first step in the iPlanet UDS application is to instantiate a new APPCConversation object:

```
conv = APPCApiSO.NewConversation();
```

Creating the COBOLBuffer and COBOLField Objects

The next step is to instantiate the COBOLBuffer object used to hold the DB2 record, and the COBOLField objects used to convert the fields into iPlanet UDS data:

```
buf : COBOLBuffer = new;
cobBirthDate : COBOLField = new;
cobBonus : COBOLField = new;
cobCommission : COBOLField = new;
cobDept : COBOLField = new;
cobEdLevel : COBOLField = new;
cobEmpNo : COBOLField = new;
cobHireDate : COBOLField = new;
cobJob : COBOLField = new;
cobMessage1 : COBOLField = new;
cobMessage2 : COBOLField = new;
cobName : COBOLField = new;
cobPhone : COBOLField = new;
cobSalary : COBOLField = new;
cobSex : COBOLField = new;
```

Defining the Layout of the COBOL Record

Now you need to define the individual fields of the COBOL record using the COBOLField objects and the characteristics of the COBOL program using the COBOLBuffer object. These definitions are shown in the following code sample:

```
loc = cobEmpNo.Define(location=0,picture='X(6)',usage='DISPLAY');
loc = cobName.Define(location=loc,picture='X(30)',usage='DISPLAY');
loc = cobDept.Define(location=loc,picture='X(3)',usage='DISPLAY');
loc = cobPhone.Define(location=loc,picture='X(4)',usage='DISPLAY');
loc = cobHireDate.Define(location=loc,picture='X(10)',usage='DISPLAY');
loc = cobJob.Define(location=loc,picture='X(8)',usage='DISPLAY');
loc = cobEdLevel.Define(location=loc,picture='X(4)',usage='DISPLAY');
loc = cobSex.Define(location=loc,picture='X(1)',usage='DISPLAY');
loc =  cobBirthDate.Define(location=loc,picture='X(10)',usage='DISPLAY');
loc = cobSalary.Define(location=loc,picture='X(11)',usage='DISPLAY');
loc = cobBonus.Define(location=loc,picture='X(11)',usage='DISPLAY');
loc = cobCommission.Define(location=loc,picture='X(11)',usage='DISPLAY');
loc = cobMessage1.Define(location=loc,picture='X(40)',usage='DISPLAY');
loc =  cobMessage2.Define(location=loc,picture='X(240)',usage='DISPLAY');
reclen = loc;

buf.SetBufferAttribute(serverCodeset=COB_CODESET_EBCDIC,
      compileTruncOption=COB_COMPILER_TRUNC_STD,
      serverPlatform=COB_PLATFORM_S390);
```

Establishing the Conversation

The next step is to establish the APPC conversation between the iPlanet UDS application and the CICS transaction program. The following TOOL code shows how the client application requests a conversation with the CICS transaction:

```
dest = 'FRTCEXSI';
tp = 'FR05';
conv.Open(dest=dest,tp=tp);
```

The following CICS COBOL code accepts the conversation request and checks for errors:

```
EXEC CICS ASSIGN FACILITY(APPC-CONVID) RESP(CICS-RESP)
END-EXEC.
IF CICS-RESP NOT = DFHRESP(NORMAL) THEN
   MOVE 'ASSIGN FACILITY FAILED' TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

If an error is found, a message is displayed upon the OS/390 console, because there is no way for a message to be sent back to the iPlanet UDS application without an APPC conversation.

Sending the Input Data

Now the input data must be sent from the iPlanet UDS application to the CICS transaction. In this application, the input data is a 6-digit employee number in character format. The following TOOL code shows how the client application sends the input data:

```
len = empRec.EMPNO_LENGTH;
conv.Write(dataBuffer=empRec.Empno,writeLength=len);
```

The following CICS COBOL code receives the input data and checks for errors:

```
EXEC CICS IGNORE CONDITION LENGERR
END-EXEC.
MOVE LENGTH OF EX04-EMPNO TO IN-LENGTH.
EXEC CICS RECEIVE CONVID(APPC-CONVID)
                  INTO(IN-AREA)
                  LENGTH(IN-LENGTH)
                  STATE(APPC-STATE)
                  RESP(CICS-RESP)
END-EXEC.
IF (CICS-RESP NOT = DFHRESP(NORMAL) AND
    CICS-RESP NOT = DFHRESP(EOC)) OR
   (EIBERR = HIGH-VALUES) THEN
   MOVE 'FAILURE RECEIVING EMPNO' TO DIAG-MSG
   PERFORM DIAG
ELSE
```

```
   IF EIBNODAT = HIGH-VALUES THEN
      MOVE ’RECEIVED NO EMPNO’ TO DIAG-MSG
      PERFORM DIAG
   END-IF
END-IF.
MOVE SPACES TO EX04-COMMAREA.
IF IN-LENGTH = LENGTH OF EX04-EMPNO THEN
   MOVE IN-AREA TO EX04-EMPNO
ELSE
   MOVE ’RECEIVED INCORRECT LENGTH FOR EMPNO’ TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

Receiving the Output Data

Now the output data must be received from the CICS transaction by the iPlanet UDS application. In this application, the output data is a 109-byte employee record in character format, plus a 40-byte message field and a 240-byte DB2 diagnostic area. The following TOOL code shows how the client application receives the output data:

```
len = reclen;
conv.Read(dataBuffer=buf,readLength=len);
```

The following CICS COBOL code sends the output data and checks for errors:

```
EXEC CICS SEND CONVID(APPC-CONVID)
               STATE(APPC-STATE)
               RESP(CICS-RESP)
               FROM(EX04-COMMAREA)
               LENGTH(LENGTH OF EX04-COMMAREA)
END-EXEC.
IF (CICS-RESP NOT = DFHRESP(NORMAL) AND
    CICS-RESP NOT = DFHRESP(EOC)) OR
   (EIBERR = HIGH-VALUES) THEN
   MOVE ’ERROR SENDING RESPONSE’ TO DIAG-MSG
   PERFORM DIAG
END-IF.
```

Terminating the Conversation

The conversation with the CICS transaction is no longer needed by the iPlanet UDS application. The following TOOL code shows how the client application terminates the conversation:

```
conv.Close(type=CLOSE_NORMAL);
```

The following CICS COBOL code closes the APPC conversation from the CICS side:

```
EXEC CICS FREE CONVID(APPC-CONVID)
               STATE(APPC-STATE)
               RESP(CICS-RESP)
END-EXEC.
```

Processing the Data

The iPlanet UDS application can now use the data received from the CICS transaction. The following TOOL code shows how the client application extracts individual fields from the buffer:

```
cobName.GetValue(buffer=buf,target=empRec.Name);
cobDept.GetValue(buffer=buf,target=empRec.Dept);
cobPhone.GetValue(buffer=buf,target=empRec.Phone);
cobHireDate.GetValue(buffer=buf,target=empRec.HireDate);
cobJob.GetValue(buffer=buf,target=empRec.Job);
cobEdLevel.GetValue(buffer=buf,target=empRec.EdLevel);
cobSex.GetValue(buffer=buf,target=empRec.Sex);
cobBirthDate.GetValue(buffer=buf,target=empRec.BirthDate);
cobSalary.GetValue(buffer=buf,target=empRec.Salary);
cobBonus.GetValue(buffer=buf,target=empRec.Bonus);
cobCommission.GetValue(buffer=buf,target=empRec.Commission);
cobMessage1.GetValue(buffer=buf,target=empRec.Message1);
cobMessage2.GetValue(buffer=buf,target=empRec.Message2);
```

The extraction of the individual fields is accomplished by using the COBOLField.GetValue method to extract the fields directly from the COBOLBuffer object. All necessary character set conversions are performed automatically.

IMS DLI Query MFS Example

The IMS DLI query MFS example application shows the use of the Transaction Adapter to invoke an IMS transaction that takes a part number as input, queries an IMS database, and returns part information as output. This application uses the IMS sample parts database, DI21PART, provided by IBM with the IMS product set. The IMS transaction uses the Message Formatting Service (MFS) for its user interface, and can be run directly from an IMS terminal. The purpose of this example is to show how IMS implicit APPC support allows the Transaction Adapter to execute a transaction that is written for interaction with a 3270 terminal, with no changes to the IMS transaction program required.

The application consists of the FRTIEX02 IMS program and the IMSdliQuery iPlanet UDS project. The FRTIEX02 program is a COBOL program that implements the IMS transaction FRTIEX02 and uses the IMS MFS interface for communicating directly with a 3270 terminal. The IMSdliQuery project provides the iPlanet UDS client interface to the FRTIEX02 IMS transaction.

Installing the Example Application

With IMS, a stage 1 and stage 2 system generation must be performed to define the example application to IMS/TM. This must be completed before the application can be fully installed. Sample IMS definitions for the transaction are in FORTE.V30M1.SOURCE(DEFIEX02). These definitions should be added to your IMS stage 1 source and then a system generation should be performed. At some installations, the IMS online change utility can be used to install the updates into the running IMS system without a restart of IMS. Your IMS system administrator should know whether or not this is allowed and should be involved in the process.

In addition to the system generation, IMS requires that a PSB and an ACB be built for a transaction. The source for the PSB for the transaction is in FORTE.V30M1.SOURCE(FRTIEP02). The JCL to perform the PSBGEN and the ACBGEN is in FORTE.V30M1.JCL(GENIEX02). This JCL should be tailored and then submitted to build the PSB and ACB for the application. At some installations, the IMS online change utility can be used to install these into the running IMS system without a restart of IMS. Your IMS system administrator should know whether or not this is allowed and should be involved in the process.

Make sure that the IMS sample parts database, DI21PART, has been installed on your IMS system and is accessible to your IMS transaction. Refer to the IBM IMS manuals for information on the installation of the sample database.

IMS Transaction

Table 4-6 lists the sources and JCLs needed for the FRTIEX02 program.

Table 4-6 Source and JCLs for the FRTIEX02 program

Source and JCLs	Location
Source for FRTIEX02 program	FORTE.V30M1.SOURCE(FRTIEX02)
JCL to compile and link program	FORTE.V30M1.JCL(FRTIEX02)
Source for message format	FORTE.V30M1.SOURCE(FRTIEF02)
JCL to assemble and link message format	FORTE.V30M1.JCL(FRTIEF02)

To build the IMS transaction, first tailor the JCL in FRTIEX02 and submit the job to compile and link the FRTIEX02 program. Next, tailor the JCL in FRTIEF02 and submit the job to assemble and link the FRTIEF02 message format.

iPlanet UDS Application

The project export of the IMSdliQuery project is in the imsdli.pex file. This project should be imported into your workspace using the workshop, and then deployed as a client on the machines where you wish to run the application. The project deploys into two partitions, a client partition and a server partition containing the APPCApiSO service object. Make sure that the server partition is deployed on the OS/390 node where you installed the Transaction Adapter.

The IMSdliQuery project contains three classes: Part describes the parts database record as an object, QueryIms is the class that interfaces with the IMS transaction using the Transaction Adapter classes, and QueryWindow is the window class that provides the user interface.

This example assumes that the symbolic destination name, FRTIEXSI, has been defined on your system and accesses the IMS system where you have installed the IMS transaction. It also assumes that the IMS transaction name is FRTIEX02. If either of these names has been changed, you must modify the RunQuery method of the QueryIms class to specify the correct names. The code under the topic "Establishing the Conversation" on page 99 shows where the changes should be made.

Running the Example Application

Complete the following steps for the version you want to execute. For either version, the following are some of the valid part numbers for which records exist: 02AN960C10, 023003806, 023007228, 023013412, 02652799, 027438995P002, and 027618032P101. The IBM IMS documentation provides more complete information on the valid part numbers for the DI21PART sample database

➤ **To execute the IMS MFS version of this example application**

1. Log on to your IMS system.
2. Type the following text and press Enter.

   ```
   /for frtm02
   ```

   A formatted screen will appear.
3. Type in a 1- to 17-character part number and press Enter.

➤ **To execute the iPlanet UDS version of this example application**

1. Bring up the iPlanet UDS client GUI window.
2. Type a 1- to 17-character part number into the input area, and click the Query button.
3. To terminate the application, close the GUI window.

Understanding the Application Logic

This section describes the logic for the communications between the iPlanet UDS client application and the IMS transaction. Each step in the execution is covered, with code fragments from both sides of the application included to show how the Transaction Adapter is used to interface with the IMS transaction program. The client application code fragments shown here are extracted from the RunQuery method of the QueryIms class. The IMS application code fragments shown here are extracted from the FRTIEX02 COBOL program.

Creating the APPCConversation Object

The first step in the iPlanet UDS application is to instantiate a new APPCConversation object:

```
conv = APPCApiSO.NewConversation();
```

Creating the COBOLBuffer and COBOLField Objects

Next, you must instantiate the COBOLBuffer object used to hold the IMS record and the COBOLField objects used to convert the fields into iPlanet UDS data:

```
buf : COBOLBuffer = new;
cPartNumber : COBOLField = new;
cDescription : COBOLField = new;
cProcurementCode : COBOLField = new;
cInventoryCode : COBOLField = new;
cMakeDept : COBOLField = new;
cMakeCostCenter : COBOLField = new;
cMakeTime : COBOLField = new;
cPlanningRevisionNumber : COBOLField = new;
cCommodityCode : COBOLField = new;
cMessage : COBOLField = new;
cSegmentNumber : COBOLField = new;
```

Defining the Layout of the COBOL Record

Now you need to define the individual fields of the COBOL record using the COBOLField objects, and the characteristics of the COBOL program using the COBOLBuffer object:

```
loc = cPartNumber.Define(location=0,picture='X(17)',usage='DISPLAY');
loc =  cDescription.Define(location=loc,picture='X(20)',usage='DISPLAY');
loc = cProcurementCode.Define(location=loc,picture='X(2)',usage='DISPLAY');
loc = cInventoryCode.Define(location=loc,picture='X(1)',usage='DISPLAY');
loc = cMakeDept.Define(location=loc,picture='X(2)',usage='DISPLAY');
loc = cMakeCostCenter.Define(location=loc,picture='X(2)',usage='DISPLAY');
loc = cMakeTime.Define(location=loc,picture='X(3)',usage='DISPLAY');
loc = cPlanningRevisionNumber.Define(location=loc,picture='X(2)',
       usage='DISPLAY');
loc = cCommodityCode.Define(location=loc,picture='X(4)',usage='DISPLAY');
loc = cMessage.Define(location=loc,picture='X(70)',usage='DISPLAY');
loc = cSegmentNumber.Define(location=loc,picture='X(4)',usage='DISPLAY');
reclen = loc;

buf.SetBufferAttribute(serverCodeset=COB_CODESET_EBCDIC,
      compileTruncOption=COB_COMPILER_TRUNC_STD,
      serverPlatform=COB_PLATFORM_S390);
```

Establishing the Conversation

The next step is to establish the APPC conversation between the iPlanet UDS application and the IMS transaction program. The following TOOL code shows how the client application requests a conversation with the IMS transaction:

```
dest = ’FRTIEXSI’;
tp = ’FRTIEX02’;
conv.SetReplaceNulls();
conv.Open(dest=dest,tp=tp);
```

Note the use of the SetReplaceNulls method. This is necessary with this IMS transaction to force the replacement of null characters (binary zeros) sent by IMS with blanks. When the IMS transaction places data in output buffer, fields are padded with null characters, and those must be replaced with blanks before the data is placed into a TextData buffer to prevent truncation of the data.

IMS implicitly accepts the conversation request and no code is required to handle this in the IMS transaction.

Sending the Input Data

Now the input data must be sent from the iPlanet UDS application to the IMS transaction. In this application, the input data is a 17-character part number. The following TOOL code shows how the client application prepares and sends the input data:

```
len = partInfo.PART_NUMBER_LENGTH;
buf.SetValue(source=partInfo.PartNumber);
while buf.ActualSize < len do
    buf.Concat(source=’ ’);
end while;
conv.Write(dataBuffer=buf,writeLength=len);
```

The following IMS COBOL code receives the input data and checks for errors:

```
CALL ’CBLTDLI’ USING DLI-GETUNIQUE,
                     IO-PCB,
                     MID.
IF IO-STATUS = DLI-OK THEN
   MOVE MID-PARTNO TO PART-KEY
   SET PARTNO-OK TO TRUE
ELSE
   MOVE DLI-GETUNIQUE TO MSG-TERM-FUNCTION
   MOVE IO-LTERM TO MSG-TERM-NAME
   MOVE IO-STATUS TO MSG-TERM-STATUS
   DISPLAY MSG-TERM-ERROR UPON CONSOLE
   SET PARTNO-ERROR TO TRUE
END-IF.
```

If an error is found, a message is displayed upon the OS/390 console, because there is no way for a message to be sent back to the iPlanet UDS application without an APPC conversation. Note that the IMS Get Unique DLI call here is the exact same call that reads from the 3270 terminal when the transaction is executed from a 3270 terminal. There is no change to the code to support APPC.

Receiving the Output Data

Now the output data must be received from the IMS transaction by the iPlanet UDS application. In this application, the output data is a 57-byte part record in character format, plus a 70-byte message field. The following TOOL code shows how the client application receives the output data:

```
len = reclen;
conv.Read(dataBuffer=buf,readLength=len);
```

The following IMS COBOL code sends the output data and checks for errors:

```
CALL ’CBLTDLI’ USING DLI-INSERT,
               IO-PCB,
               MOD,
               MOD-NAME.
IF IO-STATUS NOT = DLI-OK THEN
   MOVE DLI-INSERT TO MSG-TERM-FUNCTION
   MOVE IO-LTERM TO MSG-TERM-NAME
   MOVE IO-STATUS TO MSG-TERM-STATUS
   DISPLAY MSG-TERM-ERROR UPON CONSOLE
END-IF.
```

Note that the IMS Insert DLI call here is the exact same call that writes to the 3270 terminal when the transaction is executed from a 3270 terminal. There is no change to the code to support APPC.

Terminating the Conversation

The conversation with the CICS transaction is no longer needed by the iPlanet UDS application. The following TOOL code shows how the client application terminates the conversation:

```
conv.Close(type=CLOSE_NORMAL);
```

IMS implicitly deallocates the conversation when the transaction program terminates and no code is required to handle this in the IMS transaction.

Processing the Data

The iPlanet UDS application can now use the data received from the IMS transaction. The following TOOL code shows how the client application extracts individual fields from the buffer:

```
cDescription.GetValue(buffer=buf,target=partInfo.Description);
cProcurementCode.GetValue(buffer=buf,
              target=partInfo.ProcurementCode);
cInventoryCode.GetValue(buffer=buf,target=partInfo.InventoryCode);
cMakeDept.GetValue(buffer=buf,target=partInfo.MakeDept);
cMakeCostCenter.GetValue(buffer=buf,
              target=partInfo.MakeCostCenter);
cMakeTime.GetValue(buffer=buf,target=partInfo.MakeTime);
cPlanningRevisionNumber.GetValue(buffer=buf,
              target=partInfo.PlanningRevisionNumber);
cCommodityCode.GetValue(buffer=buf,target=partInfo.CommodityCode);
cMessage.GetValue(buffer=buf,target=partInfo.Message);
```

In this application, the individual attributes of the part information object are of the string data type. The COBOLField.GetValue method is used to extract each field into the corresponding attribute in the part information object.

IMS DLI Query Explicit APPC Example

The IMS DLI query explicit APPC example application shows the use of the Transaction Adapter to invoke an IMS transaction that takes a part number as input, queries an IMS database, and returns part information as output. This application uses the IMS sample parts database, DI21PART, provided by IBM with the IMS product set. The IMS transaction uses CPI-C for its user interface, and can be run only across an APPC conversation. The purpose of this example is to show how IMS explicit APPC support (IMS/APPC) can be used to write transactions that can be called by the Transaction Adapter. This example also shows a persistent transaction; that is, one that continues receiving requests and processing them until the client application deallocates the conversation.

The application consists of the FRTIEX03 IMS program and the IMScpicQuery iPlanet UDS project. The FRTIEX03 program is a COBOL program that implements the IMS transaction FRTIEX03 and uses the explicit APPC interface. The IMScpicQuery project provides the iPlanet UDS client interface to the FRTIEX03 IMS transaction.

Installing the Example Application

IMS System Preparation

With IMS, a stage 1 and stage 2 system generation must be performed to define the example application to IMS/TM. This must be completed before the application can be fully installed. Sample IMS definitions for the transaction are in FORTE.V30M1.JCL(DEFIEX03). These definitions should be added to your IMS stage 1 source and then a system generation should be performed. At some installations, the IMS online change utility can be used to install the updates into the running IMS system without a restart of IMS. Your IMS system administrator should know whether or not this is allowed and should be involved in the process.

In addition to the system generation, IMS requires that a PSB and an ACB be built for the transaction.

- The source for the PSB for the transaction is in FORTE.V30M1.SOURCE(FRTIEP03).
- The JCL to perform the PSBGEN and the ACBGEN is in FORTE.V30M1.JCL(GENIEX03).

  This JCL should be tailored and then submitted to build the PSB and ACB for the application. At some installations, the IMS online change utility can be used to install these into the running IMS system without a restart of IMS. Your IMS system administrator should know whether or not this is allowed and should be involved in the process.

Make sure that the IMS sample parts database, DI21PART, has been installed on your IMS system and is accessible to your IMS transaction. Refer to the IBM IMS manuals for information on the installation of the sample database.

IMS Transaction

The source for the FRTIEX03 program is in FORTE.V30M1.SOURCE(FRTIEX03). The JCL to compile and link this program is in FORTE.V30M1.JCL(FRTIEX03).

To build the IMS transaction, tailor the JCL in FRTIEX03 and submit the job to compile and link the FRTIEX03 program.

APPC/MVS Definition

IMS/APPC uses APPC/MVS for its APPC interface. Any IMS transactions that use IMS/APPC must therefore be defined to APPC/MVS in a TP profile. The JCL to define this IMS transaction program to APPC/MVS is in FORTE.V30M1.JCL(FRTITP03). Tailor the JCL in FRTITP03 and submit the job to define the FRTIEX03 transaction program to APPC/MVS.

iPlanet UDS Application

The project export of the IMScpicQuery project is in the imscpic.pex file. This project should be imported into your workspace using the workshop, and then deployed as a client on the machines where you wish to run the application. The project deploys into two partitions, a client partition and a server partition containing the APPCApiSO service object. Make sure that the server partition is deployed on the OS/390 node where you installed the Transaction Adapter.

The IMScpicQuery project contains three classes: Part describes the parts database record as an object, QueryIms is the class that interfaces with the IMS transaction using the Transaction Adapter classes, and QueryWindow is the window class that provides the user interface.

This example assumes that the symbolic destination name, FRTIEXSI, has been defined on your system and accesses the IMS system where you have installed the IMS transaction. It also assumes that the IMS transaction name is FRTIEX03. If either of these names has been changed, you must modify the StartUp method of the QueryIms class to specify the correct names. The code under the topic 'Establishing the Conversation' shows where the changes should be made.

Running the Example Application

➤ **To execute this example application**

1. Bring up the iPlanet UDS client GUI window.
2. Type a 1- to 17-character part number into the input area, and click the Query button.

   The following are some of the valid part numbers for which records exist: 02AN960C10, 023003806, 023007228, 023013412, 02652799, 027438995P002, and 027618032P101. The IBM IMS documentation provides more complete information on the valid part numbers for the DI21PART sample database.
3. To terminate the application, close the GUI window.

Application Logic

This section describes the logic for the communications between the iPlanet UDS client application and the IMS transaction. Each step in the execution is covered, with code fragments from both sides of the application included to show how the Transaction Adapter is used to interface with the IMS transaction program. The client application code fragments shown here are extracted from the StartUp, RunQuery, and ShutDown methods of the QueryIms class. The IMS application code fragments shown here are extracted from the FRTIEX03 COBOL program.

Creating the APPCConversation Object

The first step in the StartUp method of the iPlanet UDS application is to instantiate a new APPCConversation object:

```
self.Conv = APPCApiSO.NewConversation();
```

Establishing the Conversation

The next step is to establish the APPC conversation between the iPlanet UDS application and the IMS transaction program. The following TOOL code from the StartUp method shows how the client application requests a conversation with the IMS transaction:

```
dest = ’FRTIEXSI’;
tp = ’FRTIEX03’;
self.Conv.Open(dest=dest,tp=tp);
```

The following IMS COBOL code accepts the incoming conversation request and checks for errors:

```
OVE LOW-VALUES TO CONVERSATION-ID.
CALL ’CMACCP’ USING CONVERSATION-ID,
                    CM-RETCODE.
IF NOT CM-OK THEN
   MOVE ’CMACCP’ MSG-CPIC-OPERATION
   MOVE CM-RETCODE TO MSG-CPIC-RETURN
DISPLAY MSG-CPIC-ERROR UPON CONSOLE
   GO TO 10-INIT-CPIC-EXIT
END-IF.
```

Sending the Input Data

Now the input data must be sent from the iPlanet UDS application to the IMS transaction. In this application, the input data is a 17-character part number. The following TOOL code from the RunQuery method shows how the client application prepares and sends the input data:

```
len = partInfo.PART_NUMBER_LENGTH;
buf.SetValue(source=partInfo.PartNumber);
while buf.ActualSize < len do
    buf.Concat(source=’ ’);
end while;
self.Conv.Write(dataBuffer=buf,writeLength=len);
```

The following IMS COBOL code receives the input data and checks for errors:

```
MOVE LENGTH OF RECEIVE-BUFFER TO REQUESTED-LENGTH.
CALL ’CMRCV’ USING CONVERSATION-ID,
                   RECEIVE-BUFFER,
                   REQUESTED-LENGTH,
                   DATA-RECEIVED,
                   RECEIVED-LENGTH,
                   STATUS-RECEIVED,
                   REQUEST-TO-SEND-RECEIVED,
                   CM-RETCODE.
IF CM-DEALLOCATED-NORMAL THEN
   SET PARTNO-DONE TO TRUE
   GO TO 40-RECEIVE-PARTNO-EXIT
END-IF.
IF NOT CM-OK THEN
   MOVE ’CMRCV’ TO MSG-CPIC-OPERATION
   MOVE CM-RETCODE TO MSG-CPIC-RETURN
   DISPLAY MSG-CPIC-ERROR UPON CONSOLE
   PERFORM 70-DEAL-ABEND
   SET PARTNO-ERROR TO TRUE
   GO TO 40-RECEIVE-PARTNO-EXIT
END-IF.
IF RECEIVED-LENGTH NOT = LENGTH OF PART-KEY THEN
   MOVE RECEIVED-LENGTH TO MSG-BAD-RECV-LENGTH-LL
   MOVE MSG-BAD-RECV-LENGTH TO RESP-MSG
   SET PARTNO-ERROR TO TRUE
   GO TO 40-RECEIVE-PARTNO-EXIT
END-IF.
MOVE RECEIVE-BUFFER TO PART-KEY.
SET PARTNO-OK TO TRUE.
```

If the receive fails, a message is displayed upon the OS/390 console, because there is no way for a message to be sent back to the iPlanet UDS application without an APPC conversation. Note that if the iPlanet UDS application has deallocated the conversation, flags are set for normal termination of the transaction.

Receiving the Output Data

Now the output data must be received from the IMS transaction by the iPlanet UDS application. In this application, the output data is a 1-byte status code, a 37-byte part record in character format, plus a 70-byte message field. The following TOOL code from the RunQuery method shows how the client application receives the output data:

```
len = partInfo.PART_LENGTH;
buf.SetAllocatedSize(n=len);
self.Conv.Read(dataBuffer=buf,readLength=len);
```

The following IMS COBOL code sends the output data and checks for errors:

```
SET CM-SEND-AND-PREP-TO-RECEIVE TO TRUE.
CALL 'CMSST' USING CONVERSATION-ID,
                   SEND-TYPE,
                   CM-RETCODE.
IF NOT CM-OK THEN
   MOVE 'CMSST ' TO MSG-CPIC-OPERATION
   MOVE CM-RETCODE TO MSG-CPIC-RETURN
   DISPLAY MSG-CPIC-ERROR UPON CONSOLE
   PERFORM 70-DEAL-ABEND
   GO TO 60-SEND-RESPONSE-EXIT
END-IF.
MOVE LENGTH OF RESP-BUFFER TO SEND-LENGTH.
CALL 'CMSEND' USING CONVERSATION-ID,
                    RESP-BUFFER,
                    SEND-LENGTH,
                    REQUEST-TO-SEND-RECEIVED,
                    CM-RETCODE.
IF NOT CM-OK THEN
   MOVE 'CMSEND' TO MSG-CPIC-OPERATION
   MOVE CM-RETCODE TO MSG-CPIC-RETURN
   DISPLAY MSG-CPIC-ERROR UPON CONSOLE
   PERFORM 70-DEAL-ABEND
END-IF.
```

Terminating the Conversation

The conversation with the CICS transaction is no longer needed by the iPlanet UDS application. The following TOOL code from the ShutDown method shows how the client application terminates the conversation:

```
self.Conv.Close(type=CLOSE_NORMAL);
```

The IMS transaction gets a CM-DEALLOCATED-NORMAL return code on its CMRCV call when the client has deallocated the conversation. This causes the IMS transaction to terminate normally.

Processing the Data

The iPlanet UDS application can now use the data received from the IMS transaction. The following TOOL code from the RunQuery method shows how the client application extracts individual fields from the buffer:

```
buf.Offset = 0;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.STATUS_LENGTH);
partResp.Status = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.MSG_LENGTH);
partResp.Msg = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.DESCRIPTION_LENGTH);
partResp.Description = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.PROCUREMENT_CODE_LENGTH);
partResp.ProcurementCode = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.INVENTORY_CODE_LENGTH);
partResp.InventoryCode = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.MAKE_DEPT_LENGTH);
partResp.MakeDept = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.MAKE_COST_CENTER_LENGTH);
partResp.MakeCostCenter = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.MAKE_TIME_LENGTH);
partResp.MakeTime = work.Value;
work = buf.CutRange(startOffset=0,

endOffset=partResp.PLANNING_REVISION_NUMBER_LENGTH);
partResp.PlaningRevisionNumber = work.Value;
work = buf.CutRange(startOffset=0,
                    endOffset=partResp.COMMODITY_CODE_LENGTH);
partResp.CommodityCode = work.Value;
```

In this application, the individual attributes of the part information object are of the string data type. The TextData.CutRange method is used on the buffer to extract each field into a temporary TextData object, and the string value of that object is then assigned to the corresponding attribute in the part information object.

Index