TXSeries™ for Multiplatforms

Writing Encina Applications

Version 5.1

SC09-4486-02

TXSeries™ for Multiplatforms

Writing Encina Applications

Version 5.1

SC09-4486-02

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 103.

Third Edition (March 2004)

This edition replaces SC09-4486-01.

Order publications through your IBM representative or through the IBM branch office serving your locality.

Contents

Figures

Tables

About this book

This document focuses on programming in the Encina® environment. It presents an overview of the concepts of distributed transaction processing. It then provides a tutorial on how to write a simple transactional client/server application that interacts with a relational database and uses several different Encina components.

Who should read this book

This document is intended for use by programmers who want to learn how to program in the Encina environment. Basic familiarity with programming in general and with the C programming language are assumed. However, no prior knowledge of Encina or the Distributed Computing Environment (DCE) is necessary.

Document organization

This document has the following organization:

- Chapter 1, "Basic Concepts of Distributed Computing," introduces some of the concepts of distributed computing. It provides background information on client/server computing, DCE, transactions, and Encina.
- Chapter 2, "Writing the Interface for a Sample Client/Server Application," begins the development of a sample application that will be enhanced in subsequent chapters. It describes the basic steps in designing and writing the interface between the client and the server.
- Chapter 3, "Writing a Client/Server Application Using the Monitor," discusses programming in the Monitor environment. The steps involved in writing the sample application in the Monitor environment are described.
- Chapter 4, "Making the Sample Application Transactional," shows how to use Tran-C to make the sample application transactional. It discusses how to start and end (commit or abort) transactions and how to retrieve information about aborted transactions.
- Chapter 5, "Using RQS," explains how to use the Recoverable Queueing Service (RQS) with the sample application. It introduces basic RQS concepts and shows how items are queued and dequeued.
- Chapter 6, "Interacting with a Relational Database," describes how to make the sample application interact with a relational database. It describes Monitor features used in interacting with resource managers and provides an overview of how to use embedded SQL.
- Chapter 7, "Using Encina Peer-to-Peer Communications," describes how the sample application can interact with an application on a mainframe using SNA LU 6.2. It describes basic PPC concepts and shows how to use CPI-C, the interface used for PPC applications.
- Chapter 8, "Using TX," explains how to use the X/Open TX interface for managing transactions.
- Chapter 9, "Using Nested Transactions," introduces the concept of nested transactions as a way to achieve error isolation. It provides an example of doing so in the context of the sample application.
- Appendix A, "Building Encina Applications," lists the include files and libraries needed to build Encina applications.

- Appendix B, "Using Abort Codes," describes the Encina Abort Facility and shows how to use a more flexible abort mechanism than the one used in the body of the manual.
- Appendix C, "Source Code for the Example Application," contains the source code for the sample application.

Related information

For further information on the topics and software discussed in this manual, see the following documents:

- *Encina Monitor Programming Guide*
- *Encina RQS Programming Guide*
- *Encina Transactional Programming Guide*
- *Encina PPC Services Programming Guide*

Conventions used in this book

TXSeries documentation uses the following typographical and keying conventions.

Table 1. Conventions used in this book

Convention	Meaning
Bold	Indicates values you must use literally, such as commands, functions, and resource definition attributes and their values. When referring to graphical user interfaces (GUIs), bold also indicates menus, menu items, labels, buttons, icons, and folders.
`Monospace`	Indicates text you must enter at a command prompt. Monospace also indicates screen text and code examples.
Italics	Indicates variable values you must provide (for example, you supply the name of a file for *file_name*). Italics also indicates emphasis and the titles of books.
< >	Enclose the names of keys on the keyboard.
<**Ctrl-***x*>	Where *x* is the name of a key, indicates a control-character sequence. For example, <**Ctrl-c**> means hold down the **Ctrl** key while you press the **c** key.
<**Return**>	Refers to the key labeled with the word Return, the word Enter, or the left arrow.
`%`	Represents the UNIX® command-shell prompt for a command that does not require **root** privileges.
`#`	Represents the UNIX command-shell prompt for a command that requires **root** privileges.
`C:\>`	Represents the Windows® command prompt.
>	When used to describe a menu, shows a series of menu selections. For example, "Select **File > New**" means "From the **File** menu, select the **New** command."
Entering commands	When instructed to "enter" or "issue" a command, type the command and then press <**Return**>. For example, the instruction "Enter the **ls** command" means type **ls** at a command prompt and then press <**Return**>.
[ ]	Enclose optional items in syntax descriptions.
{ }	Enclose lists from which you must choose an item in syntax descriptions.
|	Separates items in a list of choices enclosed in { } (braces) in syntax descriptions.
...	Ellipses in syntax descriptions indicate that you can repeat the preceding item one or more times. Ellipses in examples indicate that information was omitted from the example for the sake of brevity.

Table 1. Conventions used in this book (continued)

Convention	Meaning
IN	In function descriptions, indicates parameters whose values are used to pass data to the function. These parameters are not used to return modified data to the calling routine. (Do *not* include the IN declaration in your code.)
OUT	In function descriptions, indicates parameters whose values are used to return modified data to the calling routine. These parameters are not used to pass data to the function. (Do *not* include the OUT declaration in your code.)
INOUT	In function descriptions, indicates parameters whose values are passed to the function, modified by the function, and returned to the calling routine. These parameters serve as both IN and OUT parameters. (Do *not* include the INOUT declaration in your code.)
$CICS	Indicates the full path name where the CICS® product is installed; for example, **C:\opt\TXSeries\cics** on Windows® or **/opt/cics** on Solaris. If the environment variable named CICS is set to the product path name, you can use the examples exactly as shown; otherwise, you must replace all instances of $CICS with the CICS product path name.
CICS on Open Systems	Refers collectively to the CICS product for all supported UNIX platforms.
TXSeries® CICS	Refers collectively to the CICS for AIX®, CICS for HP-UX, CICS for Solaris, and CICS for Windows products.
CICS	Refers generically to the CICS on Open Systems and CICS for Windows products. References to a specific version of a CICS on Open Systems product are used to highlight differences between CICS on Open Systems products. Other CICS products in the CICS Family are distinguished by their operating system (for example, CICS for OS/2® or IBM® mainframe-based CICS for the ESA, MVS™, and VSE platforms).

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TXSeries documentation, send your comments by e-mail to idrcf@hursley.ibm.com. Be sure to include the name of the book, the document number of the book, the version of TXSeries, and, if applicable, the specific location of the information you are commenting on (for example, a page number or table number).

Chapter 1. Basic concepts of distributed computing

Encina is a family of software products used to develop and manage open distributed systems. Using the underlying technology of the Open Software Foundation (OSF) Distributed Computing Environment (DCE), Encina provides the infrastructure to handle the complexities of large distributed systems and to maintain data integrity across them. Furthermore, Encina simplifies many aspects of programming distributed systems, allowing application developers to concentrate on the business logic of the program and to ignore many of the underlying details.

This manual describes how to write a basic Encina application. The chapters that follow develop a simple client/server application using a number of Encina components.

This chapter presents background information about distributed computing. It includes sections summarizing distributed software, the client/server model, and distributed transaction processing. You do not need to know all the details presented here to write the simple application that begins in Chapter 2, "Writing the interface for a sample client/server application," on page 15. If you are familiar with the concepts of client/server computing, you can skim this chapter.

Distributed computing

Encina is designed to help develop and manage open distributed systems. A *distributed computer system* consists of multiple software components on multiple computers running as a single system. The computers in a distributed system can be physically close together and connected by a local network, or they can be geographically distant and connected by a wide area network. A distributed system can comprise any number of possible configurations—mainframes, personal computers, workstations, minicomputers, and so forth. The goal of distributed computing is to make such a network act as a single computer.

Distributed systems offer many benefits over centralized systems, including the following:

- Scalability: the system can easily be expanded by adding more machines as needed.
- Redundancy: several machines can provide the same services, so if one is unavailable, work does not come to a halt. Additionally, because a larger number of smaller machines can be used, this redundancy does not need to be prohibitively expensive.

Although a distributed system as just described can conceivably be implemented using proprietary hardware and software from a small number of vendors, industry trends are toward open, standards-based systems. An *open* distributed computing system is one that can run on hardware provided by multiple vendors and use a variety of standards-based software components. Such systems are independent of the underlying software and run on various operating systems using various communications protocols. For example, an open distributed system can run on a variety of hardware platforms, with some hardware using UNIX® as

the operating system, other hardware using the Windows® operating system. For intermachine communication, this hardware can use SNA or TCP/IP on Ethernet or Token Ring.

Figure 1 shows an example of such a distributed system. This system contains two local area networks (LANs)—one consisting of UNIX workstations from several different manufacturers, the other consisting primarily of PCs running several different PC operating systems, connected together. One of the LANs is also connected to a mainframe, using an SNA connection.

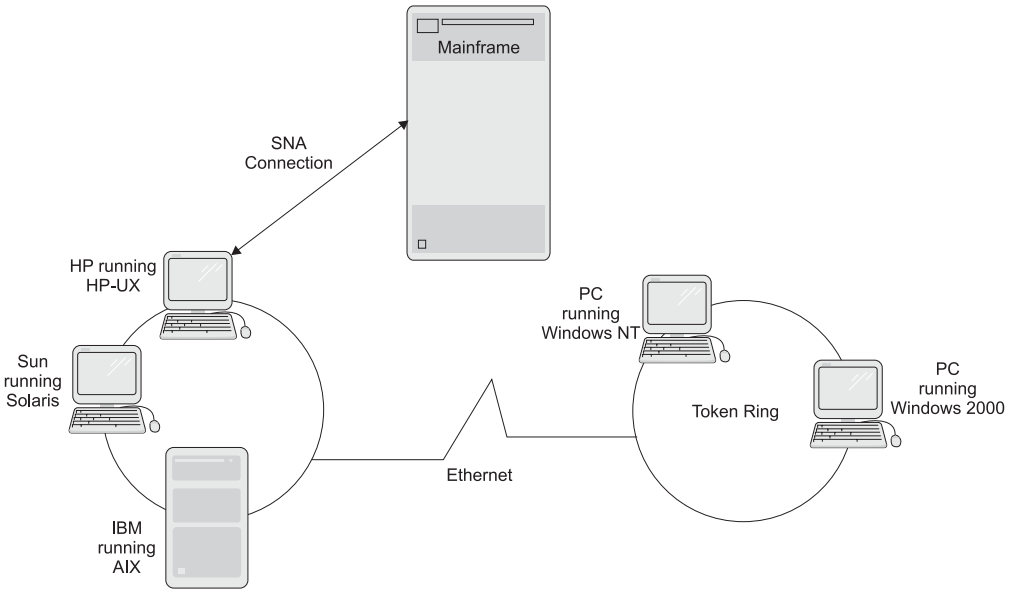

Figure 1. An example open distributed system

The client/server model

A common way of organizing software to run on distributed systems is to separate functionality into two parts—clients and servers. A *client* is a program that uses services provided by other programs called *servers*. The client makes a request for a service, and a server performs that service. Server functionality often involves some sort of resource management, in which a server synchronizes and manages access to the resource, responding to client requests with either data or status information. Client programs typically handle user interactions and often request data or initiate some data modification on behalf of a user.

For example, a client can provide a form on which a user (a person working at a data entry terminal, for example) can enter orders for a product. The client sends this order information to the server, which checks the product database and performs tasks needed for billing and shipping. A single server is typically used by multiple clients. For example, dozens or hundreds of clients can interact with a few servers that control database access.

Clients can also access several different servers, and the servers themselves can act as clients to other servers. Exactly how the functionality is distributed across servers, for example, whether a single server provides all the services a client needs or the client accesses multiple servers to perform different requests, is an application design decision. The application designer must take into account such issues as scalability, location (are both clients and servers local or is the application

spread out over a wide geographic area?), and security (for example, do the servers need to be on machines that are physically secure?). Such design decisions are outside the scope of this introduction.

Some servers are part of an application and are referred to as *application servers*. Other servers are not part of a specific application. Instead, any application can use them. DCE and Encina both provide such servers. For example, the Encina Structured File Server (SFS) provides record-oriented file access for applications.

A common design of client/server systems uses three tiers: a client that interacts with the user, an application server that contains the business logic of the application, and a resource manager that stores data. This approach is shown in Figure 2. In this way, the client is isolated from having to know anything about the actual resource manager. If you change the database you are using, the server might have to be modified, but the client does not need to be modified. Because there are usually fewer copies of the server than the client, and because the servers are often in locations that are easier to update (for example, on central machines rather than on PCs running on users' desks), the update procedure is also simplified. Furthermore, this approach provides additional security. Only the servers, not the clients, need access to the data controlled by the resource manager.

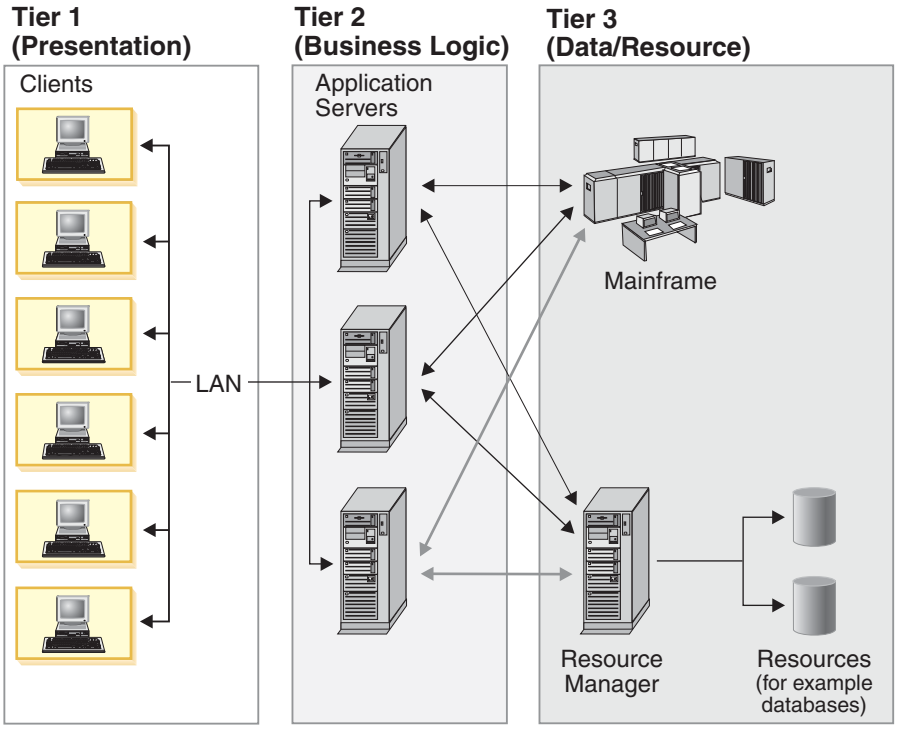

Figure 2. Three-tiered client/server architecture

Client/server applications under DCE

To implement a client/server application, a number of basic services are needed, including the following:

- A communications mechanism that enables clients and servers to interact
- A naming mechanism that enables clients to find servers that offer the desired services

- A security mechanism that enables secure communications between clients and servers

DCE, a modular collection of interfaces, provides these basic building blocks for constructing distributed client/server systems. This section provides background information about those DCE services of special interest to developers writing Encina applications. In particular, it describes the Remote Procedure Call (RPC) Facility, which provides communications between components; the Cell Directory Service (CDS), which provides a naming service that allows clients to find servers; and the Security Service, which provides for secure operation of distributed applications. For more information on DCE, see the OSF DCE documentation.

In DCE, a group of machines that work together and are administered as a unit is called a *cell*. Each cell provides the services needed for a distributed environment. The application we develop in this manual runs in a single DCE cell. DCE also provides support for intercell communications.

Remote procedure calls

Many different communications models can be used for clients to interact with servers. For example, a client can queue requests to be processed by a server, or the client and server can interact by using shared memory. However, the most common way for clients to communicate with servers (and the approach used by DCE and Encina) is by using *remote procedure calls* (RPCs).

The RPC mechanism makes the details of network communications transparent to applications. A client program invokes an RPC in the same way as it invokes a local procedure call. However, the procedure is not implemented in the same process that calls the procedure, as is the case with local procedure calls. Instead, the underlying RPC run time transmits the procedure call to the server process. The server receives the request and executes the procedure, returning the results to the client. Like local procedures, a remote procedure call can be passed both input and output parameters and can return a value. Figure 3 contrasts local and remote procedure calls.

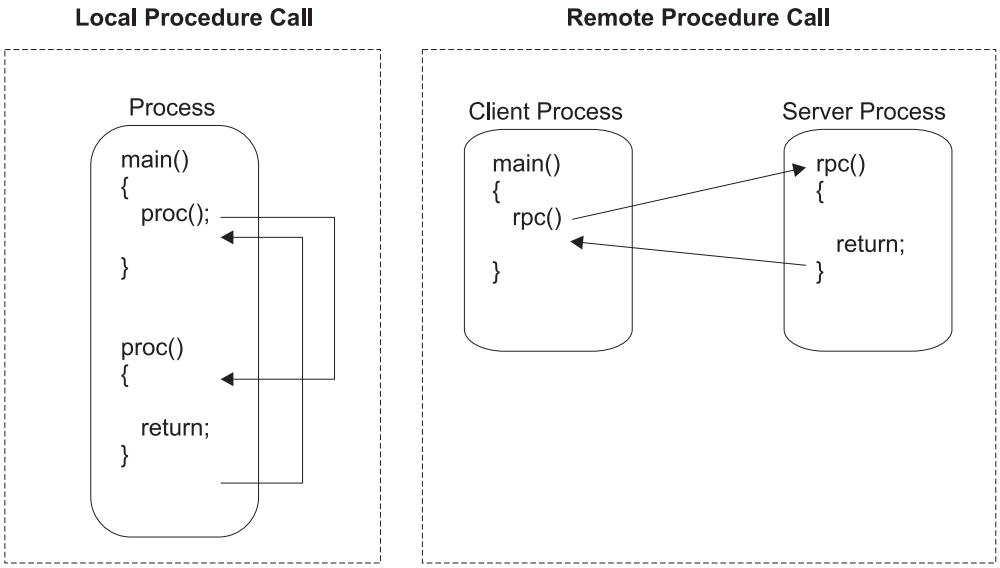

Figure 3. Local and remote procedure calls

In the case of local procedure calls, the called procedure is part of the same process and thus shares the same memory space as the calling procedure. Therefore, the called procedure can easily access any parameters passed to it because these parameters are stored in the process's memory. This is not possible for RPCs because the called procedure does *not* have access to the caller's memory, which is in a different process and often on a different machine. RPCs thus provide a mechanism for passing parameters between clients and servers. When a client program makes an RPC to a server, the procedure's parameters are automatically packed into a request message, which is sent to the remote program. Packing procedure parameters in this way is called *marshaling*. When the message is received by the remote program, the RPC facility unpacks (*unmarshals*) the message and makes the actual procedure call. The results of the call are packed into a reply message, which is returned to the calling program and unmarshalled there by the RPC system.

Marshaling and unmarshaling are handled for a program by code called a stub. A *stub* translates the local procedure call into a remote procedure call, marshaling and unmarshaling the arguments. The client stub communicates with the server stub by using the DCE RPC run-time library. This is shown in Figure 4. These implementation details do not need to be considered when writing client/server applications; the details are handled by DCE.

An *interface* is a group of remote procedure calls that a server makes available to clients. Interfaces are described by using the Interface Definition Language. The *Interface Definition Language* (*IDL*) is a high-level language with a declaration syntax similar to that of the C programming language, with additional attributes needed for defining remote procedures. An IDL file contains definitions, similar to ANSI C function prototypes, of each procedure, including the procedure name and descriptions of return values and argument types. This file is compiled with the DCE **idl** compiler, which produces the client and server stubs as well as an interface header file that can be included in the client and server program.

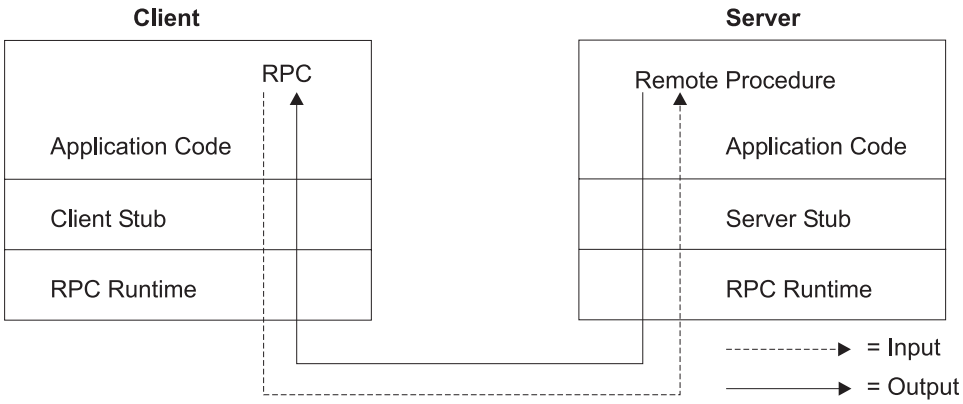

Figure 4. A DCE RPC

Locating resources

Before a client can make an RPC to a server, it must first identify a server that supports the desired interface and make the connection to the server. The server must advertise itself, making available the information the client needs to make this connection. This information is called *binding information*, and the relationship between the client and the server is called *binding*. Binding information includes the following:

- The communications protocols (for example, TCP/IP) the server can use

- Protocol-specific address information, which typically includes the hostname of the machine on which the server runs and a process address on that machine
- The interfaces the server supports

DCE stores this information in two places. The protocol information, the host identifier, and the interface information are stored in the DCE Cell Directory Service (CDS). The process address at which the server listens for RPCs (called an *endpoint*) is stored in the *endpoint map* on the host on which the server is running. The endpoint map is a simple database that relates interfaces to the process addresses of servers that support those interfaces.

When a server is initialized, it *exports* binding information to CDS and registers its endpoint and the interfaces it supports with the endpoint map. The client *imports* the binding information exported by the server. Importing binding information can be explicit (the client calls DCE functions to bind to the server) or implicit (DCE automatically makes the connection on the client's behalf).

The process of binding is described in more detail in "An overview of the binding process" on page 7. First, however, the following section describes CDS in more detail.

The DCE Cell Directory Service

The DCE Cell Directory Service provides a consistent way of identifying resources in a DCE cell. CDS maintains a database of names and attributes (including locations) of servers in a cell. A server exports its name, its location, and the interfaces it supports to CDS; clients locate the server by looking up the name or interface in CDS.

CDS is comparable to a telephone book. By providing CDS with the name of a resource, a user can obtain the location of the resource. For example, because a server exports its name and supported interfaces to CDS, users can refer to this server by name. They do not need to know the server's network address to locate the server.

CDS is actually one part of the DCE Directory Service, which also supports a global name service for identifying resources outside a cell. For the remainder of this manual, we use a single DCE cell; thus, when we refer to the DCE Directory Service, we are referring to the CDS component.

To enable users to identify resources, CDS stores objects that represent machines and resources. Each machine or resource must have an entry in CDS. CDS stores attributes of each object (for example, a network address). Object names are independent of attributes so that only the object's attributes, not its name, need to change if the program or resource it represents is moved to a different node in the cell. Users can continue to locate the moved object by using the same name.

The names and attributes of client/server applications in a DCE cell are stored together in an area called the CDS *namespace*. This namespace has a directory structure, with the root directory at the top and one or more directories beneath the root directory. For example, the Encina Monitor cell for an order entry system might be called **order_cell** and have a root directory, **/.:/order_cell**. Under this root directory, there is a directory called *server*, under which the servers that are used in the application are registered. Figure 5 on page 7 shows an example in which two example servers—**orderServer** and **billingServer**—are registered.

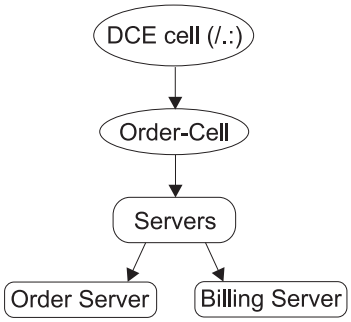

Figure 5. Encina directory structure

After servers have entries in the namespace, clients can use CDS to find them. A resource has a path-like name that identifies the resource relative to the DCE cell in which it exists. The path includes each directory in the directory structure under which the resource name is registered, as in **/.:/order_cell/server/orderServer**. The **/.:** prefix used when specifying CDS names is an abbreviation for **/.../***dce_cell_name*.

An overview of the binding process

For a client and a server to establish a binding, both must perform certain steps. Depending upon the programmatic interface used, some of these steps might be performed automatically for the application. For example, as described in Chapter 3, “Writing a Monitor client/server application,” on page 23, the Monitor performs most of these steps for both the client and the server.

To enable itself to be contacted by clients, a server performs the following steps:

1. Register each interface it supports with the RPC run-time library.
2. Publish its binding information, which includes the following steps:
   a. Select the protocol sequences it can use. Most servers support TCP/IP, UDP/IP, or both.
   b. Register its server address information in the endpoint map.
   c. Export its binding information to CDS.
3. Listen for incoming RPCs.

To contact a server, a client performs the following steps:

1. Import server binding information from CDS. This tells the client on which host it can find the desired interface exported by the server.
2. Look in the endpoint map on that host to find the endpoint of the desired server. This lookup is performed by the DCE daemon (**dced**) process, working on behalf of the client.

After the client has obtained the endpoint information, it communicates directly with the server. That is, it no longer uses CDS or the endpoint map when making RPCs.

Protecting resources

Most computer systems use some method of protecting resources. This can be a special concern in a distributed environment, where many different systems, often with thousands of users, are communicating. Users of distributed systems typically must identify themselves to the system by specifying a user name and a password. The system maintains a record of the password for each user and checks the password it has recorded against that specified by the user. The system also must

determine which resources the user is permitted to access and to what degree. *Permissions* give specific users a specified level of access to files and resources. For instance, a particular user can be allowed to read data from a particular file but not to modify the file.

The DCE Security Service provides the following security features to protect DCE resources:

- *Authentication* ensures that a *principal* (the DCE term for a user or server) is who he or she claims to be.
- *Protection levels* for RPCs control the frequency of authentication and the degree of encryption for RPCs from clients to servers.
- *Authorization* determines whether an authenticated principal is authorized (that is, has the required permissions) to make a particular request.

Authentication and protection levels are closely related. When a client is authenticated with the DCE Security Service, the client is given an encrypted *ticket*. The client presents this ticket to a server as proof that it is who it claims to be. The protection level defines how often the server checks the client's authentication.

Servers running in the Encina environment can use all three security features. When configuring a server, administrators determine whether to make use of the features. User authentication is the most basic DCE security mechanism. Protection levels provide further security control, and authorization provides even more detailed security control.

Servers check the security for an RPC by following a sequence from the most basic to the most advanced security feature (authentication to authorization). For users and applications to obtain access to Encina server resources as authenticated, protected, and authorized clients, they must

- Be authenticated to DCE.
- Communicate at (at least) the minimum level of protection specified by the server.
- Be authorized to access the resources.

A client that is unauthenticated or underprotected is treated as an unauthenticated client; unauthenticated clients are granted or denied access to a resource based on the permissions that an administrator grants to unauthenticated principals. A client that is authenticated and protected but does not have adequate permissions for a resource is considered authenticated but unauthorized; unauthorized clients are denied access to resources.

Authentication, protection levels, and authorization are described in more detail in the next three sections.

Authentication

The Authentication Service enables principals to prove their identities to DCE and to other principals. Each principal has an associated *account*, which holds information about the principal (for instance, the principal's password). The account also contains information about the *groups* to which a principal belongs. Each principal can be assigned to one or more groups. Groups simplify authorization, as described in "Authorization" on page 10.

Before requesting access to resources, a principal proves its identity by authenticating to DCE. An interactive principal (such as a user) typically authenticates by logging into the system (for example, by using the **dce_login**

command) and specifying a password. A noninteractive principal (such as a server) authenticates when it is started by obtaining a *key* (password) from its DCE *keytab file*, which is located on the machine on which it runs.

When an authenticated principal starts a process, that process inherits the principal's authentication information. However, a principal's authentication information eventually expires, causing processes started by that principal to lose authentication. Interactive principals refresh their authentication by specifying their passwords. Noninteractive principals periodically reauthenticate programmatically by using their keytab files, ensuring that their processes are authenticated even over long periods of time.

Protection levels

Protection levels determine the level of security on RPCs between clients and servers. The application designer must decide what level of security is needed. For example, must authentication occur only when the client binds to the server, or must each RPC be authenticated? Clients use authenticated RPCs for secure communications with servers. Authenticated RPCs ensure that the sender is authenticated and define the protection level used for the RPC.

DCE provides various levels of protection, ranging from not authenticating, through authenticating at the beginning of an RPC session, to encrypting all data. These levels are summarized in Table 2. The names shown in parenthesis in the table (`rpc_c_protect_level_default`, `rpc_c_protect_level_none`, and so forth) are the DCE-defined constants for the protection levels.

Table 2. RPC protection levels

Protection level	Meaning
Default (0) (`rpc_c_protect_level_ default`)	Use the DCE default protection level.
None (1) (`rpc_c_protect_level_ none`)	Perform no authentication.
Connect (2) (`rpc_c_protect_level_ connect`)	Verify authentication only when the client establishes a connection to the server.
Call (3) (`rpc_c_protect_level_ call`)	Verify authentication at the beginning of each RPC.
Packet (4) (`rpc_c_protect_level_ pkt`)	Verify authentication at the beginning of each RPC and verify that packet headers have not been modified.
Packet Integrity (5) (`rpc_c_protect_level_ pkt_integrity`)	Verify authentication at the beginning of each RPC and verify that none of the data transferred between client and server has been modified.
Packet Privacy (6) (`rpc_c_protect_level_ pkt_privacy`)	Verify authentication at the beginning of each RPC and verify that none of the data transferred between client and server has been modified. In addition, encrypt each RPC.

In general, the higher the protection level for an RPC, the higher the performance cost. The application designer and system administrator determine the optimal trade-off between data integrity and performance cost when defining an RPC's level of protection.

Authorization

Authorization enables administrators to control access to resources and services. Administrators use DCE *access control lists* (*ACLs*) to specify which users can access or use administrative services, application interfaces, functions within those interfaces, and data.

An ACL grants permission to perform operations on the object (for example, the server or file) with which it is associated. It can grant permissions to individual principals or to groups of principals. Groups simplify authorization: users with similar responsibilities can be assigned to the same group and that group can be given permissions, which are granted to each member of the group.

Specific permissions are granted to authenticated principals. Principals who are unauthenticated can be granted the same or fewer permissions, or they can be denied access altogether.

Transactions

A transaction is a tool for distributed systems programming that simplifies failure scenarios. A *transaction* is a set of operations that transforms data from one consistent state to another. This set of operations is an indivisible unit of work, and in some contexts, a transaction is referred to as a *logical unit of work* (LUW).

Transactions provide the *ACID properties*:

- *Atomicity*. A transaction's changes are atomic: either all operations that are part of the transaction happen or none happen.
- *Consistency*. A transaction moves data between consistent states.
- *Isolation*. Even though transactions can execute concurrently, no transaction sees another's work in progress. The transactions appear to run serially.
- *Durability*. Once a transaction completes successfully, its changes survive subsequent failures.

As an example, consider a transaction that transfers money from one account to another. Such a transfer involves money being deducted from one account and deposited in the other. Withdrawing the money from one account and depositing it in the other account are two parts of an *atomic* transaction: if both cannot be completed, neither must happen. If multiple requests are processed against an account at the same time, they must be *isolated* so that only a single transaction can affect the account at one time. If the bank's central computer goes down just after the transfer, the correct balance must still be shown when the system becomes available again: the change must be *durable*. Note that *consistency* is a function of the application; if money is to be transferred from one account to another, the application must subtract the same amount of money from one account that it adds to the other account.

Transactions can be completed in one of two ways: they can commit or abort. A successful transaction is said to *commit*. An unsuccessful transaction is said to *abort*. Any data modifications made by an aborted transaction must be completely undone (*rolled back*). In the above example, if money is withdrawn from one account but a failure prevents the money from being deposited in the other account, any changes made to the first account must be completely undone. The next time any source queries the account balance, the correct balance must be shown.

Distributed transactions and the two-phase commit process

A *distributed transaction* is one that runs in multiple processes, usually on several machines. Each process works for the transaction. This is illustrated in Figure 6, where each oval indicates work being done on a different machine and each arrow indicates an RPC.

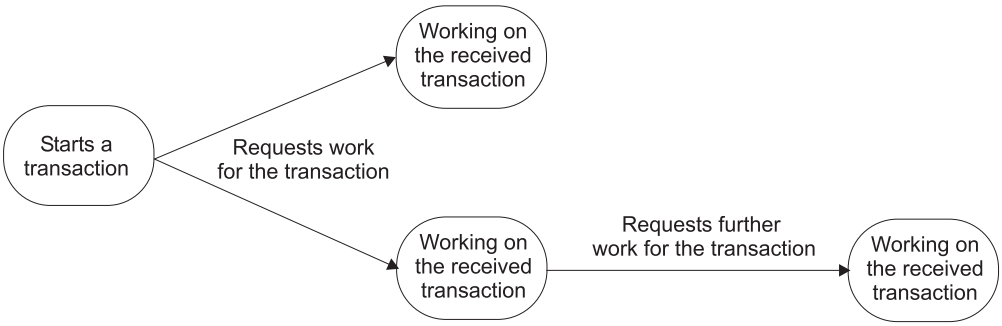

Figure 6. Example of a distributed transaction

Distributed transactions, like local transactions, must adhere to the ACID properties. However, maintaining these properties is greatly complicated for distributed transactions because a failure can occur in any process, yet even in the event of such a failure, each process must undo any work already done on behalf of the transaction.

A distributed transaction processing system maintains the ACID properties in distributed transactions by using two features:

- *Recoverable processes*. Recoverable processes are those that log their actions and thus can restore earlier states if a failure occurs.
- *A commit protocol*. A commit protocol allows multiple processes to coordinate the committing or aborting of a transaction. The most common commit protocol, and the one used by Encina, is the two-phase commit protocol.

Recoverable processes can store two types of information: transaction state information and descriptions of changes to data. This information allows a process to participate in a two-phase commit and ensures isolation and durability. Transaction state information must be stored by all recoverable processes. However, only processes that manage application data (such as resource managers) must store descriptions of changes to data. Not all processes involved in a distributed transaction need to be recoverable. In general, clients are not recoverable because they do not interact directly with a resource manager. Encina calls processes that are not recoverable *ephemeral*.

The *two-phase commit protocol*, as the name implies, involves two phases: a prepare phase and a resolution phase. In each transaction, one process acts as the coordinator. The *coordinator* oversees the activities of the other participants in the transaction to ensure a consistent outcome.

In the *prepare phase*, the coordinator sends a message to each process in the transaction, asking each process to prepare to commit. When a process prepares, it guarantees that it can commit the transaction and makes a permanent record of its work. After guaranteeing that it can commit, it can no longer unilaterally decide to abort. If a process cannot prepare (that is, if it cannot guarantee that it can commit the transaction), it must abort.

In the *resolution phase*, the coordinator tallies the responses. If all participants are prepared to commit, the transaction commits; otherwise, the transaction aborts. In either case, the coordinator informs all participants of the result. In the case of a commit, the participants acknowledge that they have committed.

Transaction processing monitors

Transaction processing is supported by programs called *transaction processing monitors* (TP monitors). TP monitors perform the following three types of functions:

- *System run-time functions*: TP monitors provide an execution environment that ensures the integrity, availability, and security of data; fast response time; and high transaction throughput.
- *System administration functions*: TP monitors provide administrative support that lets users configure, monitor, and manage their transaction systems.
- *Application development functions*: TP monitors provide functions for use in custom business applications, including functions to access data, to perform intercomputer communications, and to design and manage the user interface.

Encina provides a TP monitor, the Encina Monitor.

Introduction to Encina

Encina is a family of software products for building and running large-scale, distributed client/server systems. Encina uses and enhances the facilities provided by DCE. Figure 7 shows the high-level architecture of Encina.

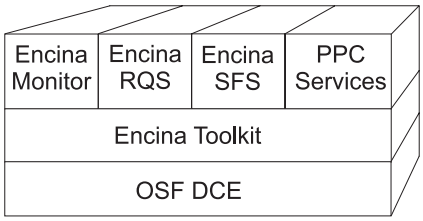

Figure 7. Architecture of Encina

This section provides a brief overview of the Encina products. Subsequent chapters describe in more detail various parts of Encina as they are used to develop a transactional client/server application.

The Encina Monitor

The Encina Monitor, or just the Monitor, is a TP monitor that provides the means to develop, run, and administer transaction processing applications. The Encina Monitor, in conjunction with resource managers, provides an environment to maintain large quantities of data in a consistent state, controlling which users and clients access specific data through defined servers in specific ways. The Monitor provides an open, modular system that is scalable and that interoperates with existing computing resources such as IBM mainframes.

The application built in subsequent chapters of this manual uses the Monitor. The Monitor and its programming interface are described in more detail in Chapter 3, "Writing a Monitor client/server application," on page 23.

The Recoverable Queueing Service (RQS)

The Recoverable Queueing Service (RQS) allows applications to queue transactional work for later processing. Applications can then commit their transactions with the assurance that the queued work will be completed transactionally at a later time.

The application built in this manual uses RQS to queue shipping requests for later processing. RQS is described in more detail in Chapter 5, "Using RQS," on page 43.

The Structured File Server (SFS)

The Encina Structured File Server (SFS) is a record-oriented file system that provides transactional integrity, log-based recovery, and broad scalability. SFS uses *structured files*, which are composed of records. The records themselves are made up of fields. For example, each record can contain information about an employee, with fields for the name, employee number, and salary. SFS is described in the *Encina SFS Programming Guide*.

Peer-to-Peer Communications (PPC) Services

PPC Services enable Encina transaction processing systems to interoperate with systems, typically mainframes, that have System Network Architecture (SNA) LU (Logical Unit) 6.2 communications interfaces. PPC Services provide bidirectional transactional communications, which enable applications to share data between mainframes and Encina. For example, Encina applications can both make requests of services provided by mainframe-based applications and service requests from mainframe systems, manipulating data on both systems with transactional consistency in either case.

The application built in this manual uses PPC to interact with a mainframe. PPC is described in more detail in Chapter 7, "Using Encina Peer-to-Peer Communications," on page 61.

The Encina Toolkit

The Encina Toolkit is a collection of modules, libraries, and programs that provide the functions required for large-scale distributed client/server system development. The modules of the Toolkit include log and recovery services, transaction services, and Transactional Remote Procedure Call (TRPC, an extension to the DCE RPC technology). These modules transparently ensure distributed transactional integrity. The Toolkit also provides Transactional-C (Tran-C), a transactional extension to the C programming language. In this manual, the only parts of the Toolkit that are used directly are Tran-C, which we use for creating transactions, and TRPC, which are used for ensuring transactional integrity of RPCs.

Lower-level modules of the Encina Toolkit include the following:

- The Distributed Transaction Service (TRAN), which coordinates transactions.
- The Lock Service (LOCK), which prevents conflicting access to data.
- The Log Service (LOG) and Recovery Service (REC), which guarantee that changes made to data on behalf of a transaction are either performed in their entirety or appear never to have occurred.
- The Volume Service (VOL), which enables applications to address storage in terms of logical units called *volumes*. Volumes can consist of single or multiple physical disk partitions, can include entire disks, and can span multiple physical

disks. The Volume Service maintains the storage used by the Log Service, which in turn stores the data required by the Recovery Service to restart a recoverable application.

- The Transaction Manager-XA Service (TM-XA), which implements the transaction manager side of the X/Open XA interface (see Chapter 6, "Interacting with a relational database," on page 53). TM-XA coordinates distributed transactions with relational database managers.
- The Transarc/Encina DCE Utilities Library (TRDCE), which provides utilities for constructing client and server programs.

As the application is developed in subsequent chapters, we do not use the lower-level modules directly (although application programs can access them if they need to). For example, we use Tran-C, not TRAN, for creating transactions. Tran-C itself calls TRAN; we do not have to call TRAN directly.

Scope and layout of the remainder of this manual

Throughout the remainder of this manual, we develop a sample application. This application illustrates many of the techniques used in distributed client/server applications. Although it is simplified, the application uses a number of DCE and Encina components. As much as possible, we follow good design and coding principles. In those cases where we deviate from such principles (generally in an attempt to keep the application simple and our discussion focused on Encina programming issues), we explicitly say so.

We begin this application in the next chapter, where we develop a simple, nontransactional client/server application. In subsequent chapters, we make the application transactional, run it under the Monitor, and have it interact with a database, RQS, and PPC. We discuss using the X/Open TX interface to manage transactions. Finally, we use the application to demonstrate some basic concepts of error isolation in the Encina environment.

Chapter 2. Writing the interface for a sample client/server application

In this chapter and throughout the remainder of this manual, we develop a simple Encina application. We start in this chapter by designing and writing the interface. In the next two chapters, we write the client/server aspects of the application and make the application transactional. In subsequent chapters, we add features using other parts of Encina.

Overview of the sample application

This manual uses a simple order entry system as an example of a basic application. Within this sample application, the user has one option: to order an item. The application checks for the ordered item in its database, decrements the inventory, and sends requests to billing and shipping. Orders of over $1000 are considered high priority and are shipped before orders under this amount. A real application would, of course, have more features and options, and a more robust user interface.

Figure 8 shows the basic design of this application.

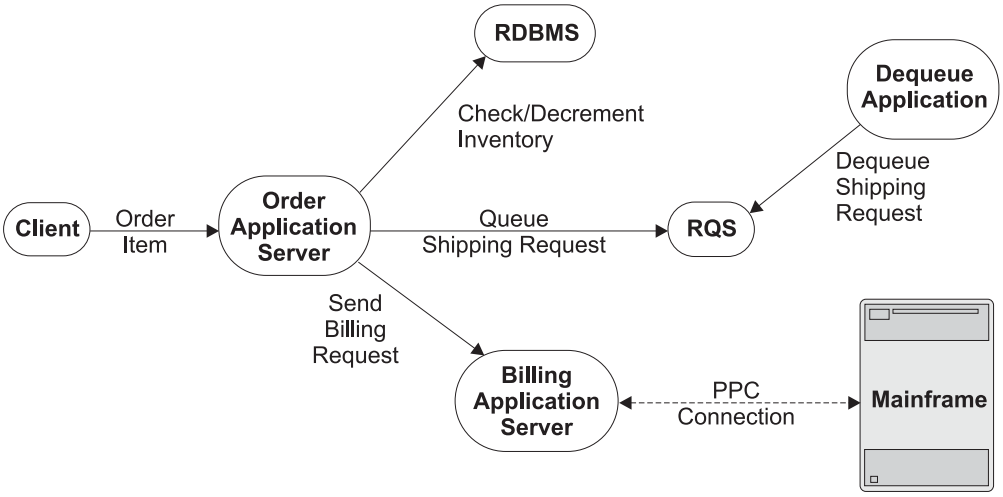

Figure 8. Sample order-entry application

In this application, the client, which interacts with the user, makes a remote procedure call to an application server (which might be running on the same machine or on a different machine). This application server, in turn, does three things: it checks for the item in a relational database, decrementing that database entry by the amount ordered; it queues a shipping request, using the Recoverable Queueing Service (RQS); and it sends a billing request to a mainframe. It executes the billing request by making a remote procedure call to another application server, which in turn uses Encina Peer-to-Peer Communications (PPC) support to start a conversation with the mainframe. If any of these operations fails (for example, if there is insufficient inventory to fill the order), the server returns a status code to the client, indicating that an error has occurred. If all operations succeed, a small, stand-alone application dequeues the shipping request.

This application is designed to demonstrate Encina functionality. It is not intended to show the best possible or the most robust design. As much as possible, we try to follow good design and programming practices. However, the application is far simpler than a real application would be. For example, the server interface offers only one function. Furthermore, the application does not try to recover from errors; it simply aborts when an error occurs.

Some application design choices were made to demonstrate certain Encina features, even though an actual application would not need to be designed this way. For example, one application server makes an RPC to another, which in turn uses PPC to interact with a mainframe. The first application server could have contacted the mainframe directly. It does not because we want to demonstrate the way in which one application server can be a client to another application server.

This chapter develops the interface between the client and the server. The shaded area in Figure 9 shows the part of the application that is described in this chapter. Other chapters develop other parts of the application.

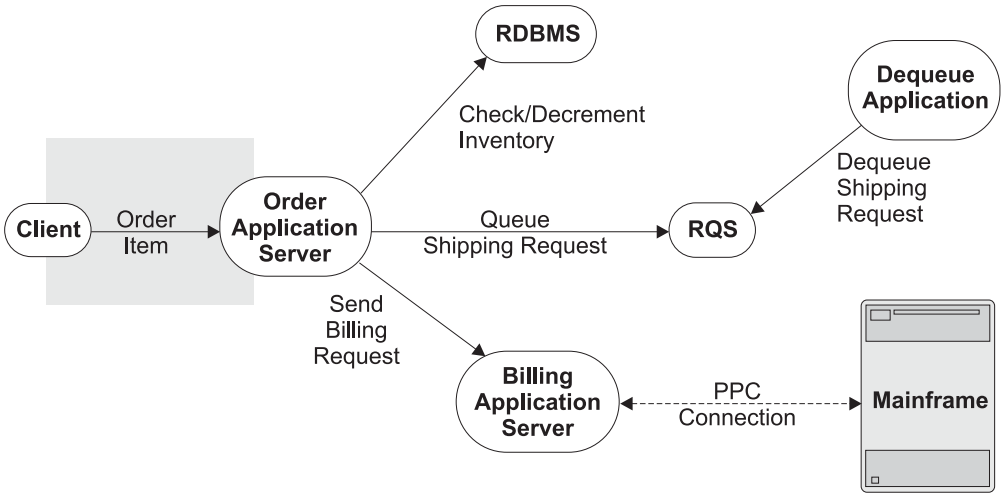

Figure 9. The interface of the sample order-entry application

Defining the interface

The first step in writing an application is defining the interface the server exports (makes available to the client). To do so, we must decide which functions the server exports, what their arguments are, and so forth.

One decision that we must make is exactly how to modularize the interface: does the interface consist of a number of small, general purpose functions that perform specific task or a smaller number of larger functions that perform multiple tasks? For example, ordering an item in the order-entry system described in the previous section involves the following steps:

- Checking to see whether the item is available
- Decrementing the inventory
- Sending a billing request
- Queueing a shipping request

As part of designing the interface, we must decide whether the server exports four functions (one for each task), one function (for "ordering an item"), or some number in between.

In general, it is practical to have the interface reflect the logic of what the client wants to do and to allow the server to perform as much of the work as possible. This approach has several benefits:

- It simplifies client programming. The client must know only that it has to order an item; it does not need to know the steps involved in this process.
- It makes it easier to change implementation details. If we later decide that "ordering an item" also includes sending electronic mail to the manufacturing department if the inventory drops below a certain amount, only the server, not the client, needs to be modified. Similarly, we could replace one relational database with another or with Encina SFS without modifying the client. Because there are usually fewer copies of the server than the client, and because the servers are often in locations that are easier to update (for example, on central machines rather than on PCs running on users' desks), the update procedure is also simplified.
- It reduces the number of remote procedure calls (RPCs) made by the client to the server. In this example, the client must make only one RPC, not four separate RPCs (one for each of the four parts of an order listed above). This can result in better performance.
- It provides for better application security because only the server needs to know how to access the database.

Our example uses this "light client/heavy server" approach. The client makes a single RPC. The server then performs the steps involved in ordering an item, issuing RPCs of its own.

The example interface

For simplicity, our example interface consists of only one function: **OrderItem**. As input, this function takes three arguments: the stock number of the item, the number of items ordered, and the identifier of the customer ordering the item. It returns a status code.

If this function were a local procedure, its prototype would look like that shown in Figure 10.

```
unsigned long OrderItem (unsigned long stockNum,
                         unsigned long numOrdered,
                         unsigned long customerId);
```

Figure 10. Prototype for the OrderItem function

To be invoked as a remote procedure call, this function must be defined in an *interface definition file* using the DCE Interface Definition Language (IDL) or the Encina transactional extension to IDL, TIDL. For our example, because we are using the Monitor for binding, we are using a TIDL file.

Creating the Transactional Interface Definition Language file

In a TIDL file, we must specify the following about each function:

- The arguments to the function and their types
- Whether the function is to be invoked transactionally or nontransactionally

- Whether each argument is input, output, or both

The **OrderItem** function is defined in the TIDL file as shown in Figure 11.

```
[nontransactional] error_status_t OrderItem(
                                [in] unsigned long stockNum,
                                [in] unsigned long numOrdered,
                                [in] unsigned long customerId);
```

Figure 11. TIDL definition for the OrderItem function

The type **error_status_t** is defined by DCE for error status. It is equivalent to the unsigned long type, but it also specifies to DCE that this is a status parameter or return value. This fact is used by DCE if we specify that DCE use a status code to return errors it detects to our client rather than generating exceptions (see "Creating the Transactional Attribute Configuration File" on page 19).

The **OrderItem** function is defined as *nontransactional* because it is not invoked transactionally by the client. However, the function itself starts a transaction, as described in Chapter 4, "Making the sample application transactional," on page 37.

The full TIDL file must include a *universal unique identifier* (*UUID)* generated by the DCE **uuidgen** command. The UUID is a number that uniquely identifies an interface across all network configurations. You can use the following command to generate the UUID as well as the skeleton for the TIDL file:

```
% uuidgen -i -o OrderInterface.tidl
```

Because we are using Monitor transparent binding (which we add to our application in Chapter 3, "Writing a Monitor client/server application," on page 23), our client automatically binds to a server exporting an interface with this UUID when it makes an RPC. Were we to use explicit binding, we would specify this UUID when obtaining a binding handle to the server.

We must edit the skeleton produced by the **uuidgen** command to add the interface name (`OrderInterface`) and the prototype for the function (**OrderItem**) that makes up the interface. The complete TIDL file for this application is shown in Figure 12. The version (1.0) is the version number of the interface; our sample application does not use the version number.

```
[
uuid(002978fe-bb72-1ea6-b3fb-9e620404aa77),
version(1.0)
]
interface OrderInterface
{
import "tpm/mon_handle.idl";
[nontransactional] error_status_t OrderItem(
                                    [in] unsigned long stockNum,
                                    [in] unsigned long numOrdered,
                                    [in] unsigned long customerId);
}
```

Figure 12. TIDL file for the example application

The **import** statement in the TIDL file imports a file needed for a client that is using the Monitor binding facilities (see Chapter 3, "Writing a Monitor client/server application," on page 23).

Creating the Transactional Attribute Configuration File

A client or a server can use an attribute configuration file (ACF) or a transactional attribute configuration file (TACF) to modify the way the TIDL and IDL compilers create stubs. Our application uses a TACF, so the remainder of this section describes TACFs. However, much of what is discussed also applies to ACFs.

The TACF has two primary uses:

- To control the way binding occurs
- To control the way errors and exceptions are reported

Our client is using Monitor universal binding, which is explained in detail in "Binding in the Monitor environment" on page 25. Using this method, the client specifies an extra parameter — a binding string — which specifies how the Monitor is to determine the server to which the client binds. We specify that the client is going to use universal binding in the TACF.

The TACF also allows communications errors and exceptions in the underlying DCE layers to be returned to the client. Errors that are detected by the server are returned to the client as status codes. However, errors detected by DCE (for example, network communications errors) normally generate exceptions, which typically cause the client to exit. Applications can be designed to handle exceptions, but this is a more complicated programming task than simply checking a status code. We can use a TACF to specify that such exceptions are instead to be returned as status codes, which can be handled by the client in the same way that it handles other status codes.

Figure 13 shows a TACF. The handle definition in the TACF (**[explicit_handle (encina_handle_t void)]**) specifies that we are using Monitor universal binding. See "Using Monitor universal binding" on page 25 for more information.

The TACF also specifies that communications status and fault status are to be returned by the **OrderItem** function. Thus, for example, if a network error occurs, an indication of this is returned by the **OrderItem** function and the client can handle the error as it sees fit; an exception is not raised and the application is not terminated.

```
/* TACF for OrderInterface */
[explicit_handle (encina_handle_t void)]
interface OrderInterface
{
      [comm_status, fault_status] OrderItem();
}
```

Figure 13. Using a Transactional Attribute Configuration File to control errors and exceptions

Processing the TIDL and TACF files

The TIDL file is compiled by using the TIDL compiler, **tidl**. It produces a number of files, including an IDL file, which must in turn be compiled by the IDL compiler, **idl**. The IDL compiler produces the necessary stub files (the code that turns the local procedure call into a remote procedure call) and the

OrderInterface.h file. The TIDL compiler automatically compiles the TACF for the interface if a TACF exists (see "Creating the Transactional Attribute Configuration File" on page 19).

Figure 14 shows the files used and produced in the compilation process. The dotted lines in the figure indicate files that are optional. Figure 15 shows how several of these output files are processed further by the IDL compiler.

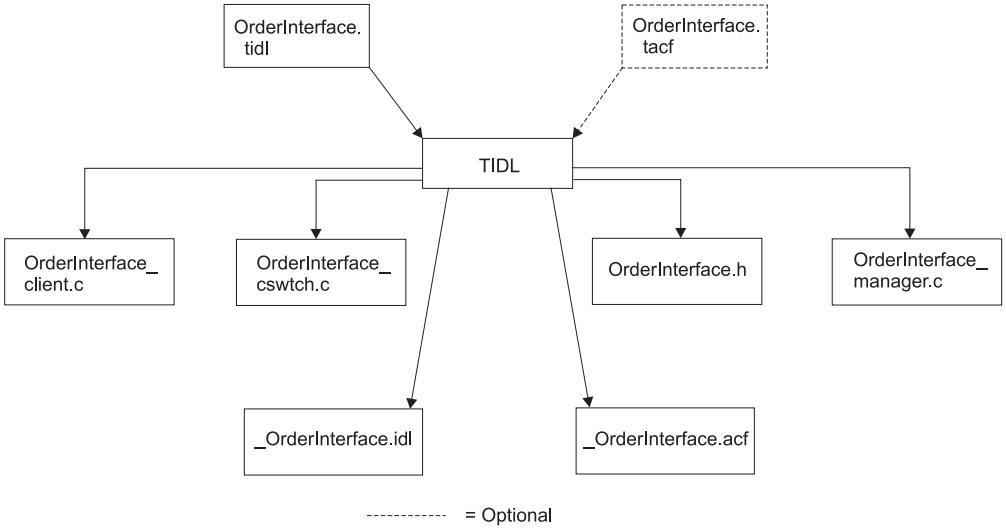

Figure 14. Files used and produced by the TIDL compiler

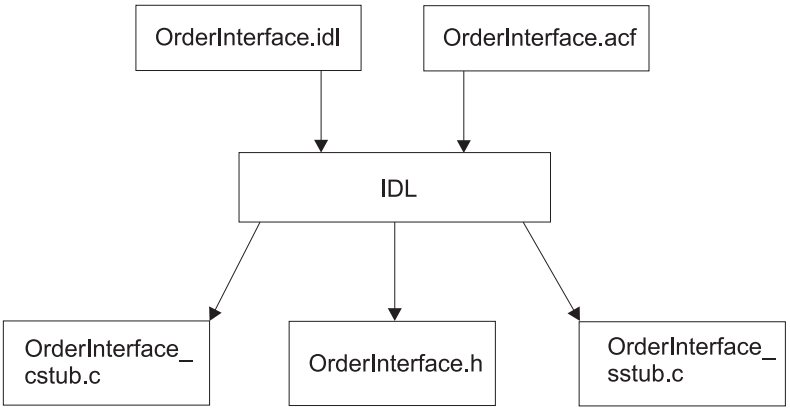

Figure 15. Files used and produced by the IDL compiler

The **OrderInterface.h** header file is included in both the client and the server. The C files produced are compiled and linked with the client and server.

Implementing the server interface

The interface as we have defined it consists of one function, **OrderItem**, which takes three arguments and returns a status code. The function in turn calls three functions:

- **PlaceOrder**, which checks for the item in a database and decrements that database by the number ordered
- **QueueItemForShipping**, which places information about the order on a queue for subsequent shipping

- **BillForItem**, which bills the customer's account maintained in a mainframe database

If any of these functions fail, the **OrderItem** function returns. Figure 16 shows this function at this point in its development.

```
#include "OrderInterface.h"
error_status_t OrderItem(idl_ulong_int stockNum,
                         idl_ulong_int numOrdered,
                         idl_ulong_int customerId)
{
 idl_long_int costPerItem, totalCost;
 short priority;
 /* Start a transaction here, as described in the chapter on transactions. */
 ...

 PlaceOrder(stockNum, numOrdered, &costPerItem);

 totalCost = numOrdered * costPerItem;
 if (totalCost > 1000)
        priority = HIGH_PRIORITY;
 else
        priority = NORMAL_PRIORITY;

 QueueItemForShipping(stockNum, numOrdered,
                      customerId, priority);

 BillForItem(customerId, totalCost);

 /* If the transaction aborts, return failure indication.*/
 ...


 return SUCCESS;
}
```

Figure 16. The OrderItem function

The application as shown in Figure 16 does not perform any error checking or error recovery. If a function call fails, the **OrderItem** function does not try to undo the results of previous successful function calls. For example, if billing for the item fails, the application does not try to undo the order or remove it from the shipping queue. In Chapter 4, "Making the sample application transactional," on page 37, we use a transaction to implement error checking and recovery and fill in the missing parts of this function.

A note on data types

In the TIDL file, we used the data type **unsigned long**. This type in DCE IDL files and Encina TIDL files, unlike the C data type of the same name, is defined to be exactly 32 bits long. All of the types that are used in TIDL and IDL files are defined this way. The C types, on the other hand, are defined so that they can be different sizes on different systems; on many platforms, **unsigned long** integers are 32 bits long, but on others, they are 64 bits long.

The parameters of the **OrderItem** function must be the same as those defined in the TIDL file; thus, we cannot define them as **unsigned long**. IDL and TIDL provide data types for use in C programs. The **OrderItem** function uses the **idl_ulong_int** data type for unsigned 32-bit integers. DCE functions also customarily use the **error_status_t** data type for status codes. This type is identical to the **idl_ulong_t** type, and our application uses it for status codes.

For a complete list of available DCE data types, see the *OSF DCE Application Development Guide*.

Notes on building and running the application

As we create and enhance our application, each chapter ends with a section of notes on building and running the application. These notes are not intended to provide all of the information needed, which varies from platform to platform; they are intended to provide some guidance for the general steps involved in building and running an Encina application. For more details, see Appendix A, "Building Encina applications," on page 83 and the manuals for the specific Encina components.

At this point, our application is divided into the following four files:

- **OrderInterface.tidl** — the TIDL file for the order interface
- **OrderInterface.tacf** — the client-side TACF file for the order interface
- **OrderServer.c** — the application code for the order server, which currently contains only the code for the **OrderItem** function
- **order.h** — an application-specific header file

The server (and the client, when we write it) must include the **OrderInterface.h** file generated by the IDL compiler and the **order.h** file. The **order.h** file is written as part of the application; it includes definitions of macros, constants, and data types used by both the client and the server.

Chapter 3. Writing a Monitor client/server application

In the previous chapter, we defined the interface for a sample client/server application. In this chapter, we begin to write that application.

Chapter 1, "Basic concepts of distributed computing," on page 1 introduced the Encina Monitor. The current chapter starts by providing more background information about the Monitor, particularly about the Monitor programming model. For more information on the Monitor, see the *Encina Monitor Programming Guide*.

An overview of the Encina Monitor

The Encina Monitor provides a number of features that we use in our application. Many of these features simplify the program or automate some of the basic tasks that must be performed by a client/server application. The Monitor also provides a number of run-time and administrative benefits, including load balancing and centralized administration. Although these features are not explicitly discussed in this chapter, they provide further benefits to our application.

The Encina Monitor operating environment

The Monitor provides an environment that simplifies creating and running client/server applications. This run-time environment is the Monitor *cell*, or single administrative unit, which consists of *nodes* (machines) on which applications and Monitor software run. The machines on which a Monitor cell runs are a subset of the machines on which a DCE cell runs; all nodes on which the Monitor runs must be part of the same DCE cell because the Monitor uses DCE services such as the Cell Directory Service and the Security Service. More than one Monitor cell can exist in a single DCE cell.

The Monitor is *not* a single process. It consists of a number of processes. Some of these processes (such as the *cell manager*, which manages the entire cell, and the *node manager*, one of which manages each node on which a server process runs) are part of the Monitor environment; you do not have to write them. Others, such as the client and the application server, are part of your application. Some of these processes are shown in Figure 17 on page 24. The node manager runs on the same node as the application servers it manages. The cell manager can run on any node in the cell. In Figure 17 on page 24, those processes that are part of the application are in the lower half of the figure. The processes in the upper half of the figure are part of the Monitor environment.

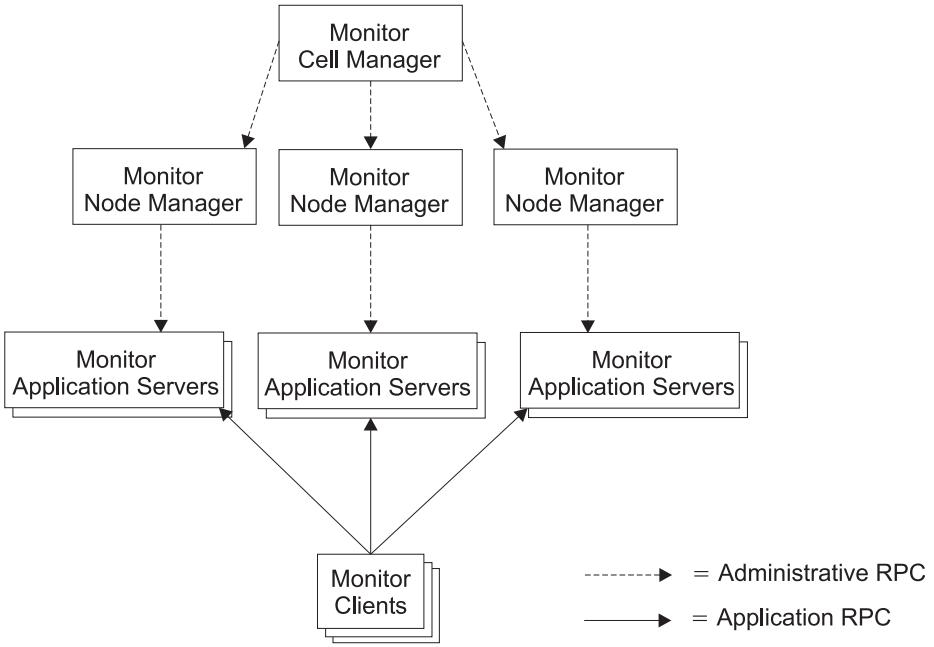

Figure 17. Monitor architecture

A Monitor application follows the general three-tiered design that we have used in the previous two chapters: a client makes an RPC to an application server, which in turn communicates with resource managers and other servers. A Monitor application server consists of one or more processes called *processing agents* (PAs), each of which executes the application server code. All PAs that make up a server run on the same machine and are administered as a single entity.

In this chapter, we write a Monitor application server, which implements the order interface, and a Monitor client. We also write a second Monitor application server for processing billing information.

Monitor features used by application programs

The Monitor provides a number of features that simplify the writing of application programs:

- Simplified binding. The client, when it makes an RPC, can specify whether binding occurs to any server that offers the interface, to a specific server, or to a server based on the data supplied in a named routing table. The Monitor handles all the details.
- Simplified initialization. The Monitor performs most of the initialization needed by both client and server; the application does not need to initialize each of the underlying components individually.
- Ease of scheduling and load balancing. A Monitor application server can consist of multiple processing agents, each of which executes the server code. The Monitor also allows administrators to assign priorities to servers. For example, an administrator can assign priorities such that servers on more powerful machines service more RPCs.
- Automated recoverability. The server needs to call only a single function during initialization. The Monitor handles all other aspects of making the server recoverable, including initializing and using the Encina Log and Recovery Services.

- Simplified access to DCE services. The Monitor simplifies the use of many of the other underlying DCE features in addition to RPCs and CDS. For example, many aspects of security are handled automatically by the Monitor.

We use a number of these features as we enhance our application. These and many other features provided by the Monitor are described in the *Encina Monitor Programming Guide*.

Binding in the Monitor environment

A client making an RPC to a server must bind to that server. A Monitor client can specify the server to which it binds in several ways:

- It can bind to any server that supports the desired interface.
- It can bind to a specific server that supports the interface.
- It can bind to a server based on the data in one of the remote procedure's parameters. This method is referred to as *data-dependent routing*, in that the RPC is routed to a server based on the data used.

The client can specify the type of binding to use as a parameter to the RPC. This parameter, which must be the first parameter passed to the remote procedure call, is called a *binding string*. The exact format of the string depends upon the type of binding being used. This type of binding, in which a binding string is explicitly passed as a parameter, is called Monitor *universal binding*. It is the type of binding we specified in the TACF we created in "Creating the Transactional Attribute Configuration File" on page 19.

A client can also use Monitor simple binding. In Monitor *simple binding*, the client does not specify a binding string when making an RPC. Instead, a default binding string is assigned administratively to an interface. If a default is not specified, the client binds to any server that supports the interface.

Our example uses Monitor universal binding to bind to any server that supports the interface. Assuming our server is replicated on multiple machines, this makes the client/server application resistant to server failures. This also enhances load balancing because the administrator can specify the percentage of RPCs that go to each server supporting the desired interface.

The example also demonstrates how easy it is to change the binding method by simply changing the binding string. "Using data-dependent routing" on page 31 expands the example, to show how data-dependant routing could be added to the application.

Using Monitor universal binding

An application that is using Monitor universal binding typically specifies this in the TACF. The TACF for the sample application was shown in Figure Figure 13 on page 19. The following line from that TACF specifies the binding method:

```
[explicit_handle (encina_handle_t void)]
```

The `encina_handle_t` modifier specifies that the client passes a binding string in each RPC that it makes. The `void` modifier specifies that manager functions at the server do not receive this binding string as a parameter. If the TACF had not included the `void` attribute, the binding string specified by the client would have been passed to the manager function as the first parameter of the RPC.

Writing the server

A server must implement its interface and perform initialization. In the previous chapter, we implemented the interface. In this chapter, we describe the steps for server initialization.

The Monitor automatically initializes most of the underlying Encina Toolkit components such as Tran-C and the Transaction Service. The Monitor also handles all the details of registering the server with CDS. Our application needs only to direct the Monitor to perform certain initialization steps that it does not perform automatically and to register the interface explicitly.

The general steps for initializing a server in the Monitor environment follow:

1. Call the **mon_InitServerInterface** function to register the interface (see "Registering the interface").
2. Inform the Monitor what other initialization needs to be performed in addition to its normal default initialization. In our case, we initiate interaction with a resource manager (see "Initializing the resource manager").
3. Call the **mon_InitServer** function to initialize both the server and Encina (see "Initializing Encina" on page 27).
4. Perform any actions that must be performed after Encina has been initialized but before the server begins accepting RPCs. Our server performs no such actions at this point; later, after we modify it to use RQS, it will do so.
5. Call the **mon_BeginService** function to signal to the Monitor that the server is ready to listen for RPCs (see "Listening for RPCs" on page 27).

Registering the interface

The Monitor handles registration in CDS. We need only to specify to the Monitor the interface to be registered. This is done with the Monitor **mon_InitServerInterface** function.

The **mon_InitServerInterface** function takes two arguments: an interface handle and a manager entry point vector. These arguments can be obtained from the TIDL-generated **OrderInterface.h** file. However, the Monitor provides a macro, **MON_SERVER_INTERFACE**, that generates both arguments, given the interface name, the major version number, and the minor version number.

Figure 18 shows the call to the **mon_InitServerInterface** function that registers the **OrderInterface** interface. The version number, as specified in the TIDL file, is 1.0, so the second and third arguments to the **MON_SERVER_INTERFACE** macro are 1 and 0, respectively. Because **MON_SERVER_INTERFACE** is a macro, there can be no spaces between its arguments.

```
status = mon_InitServerInterface
           (MON_SERVER_INTERFACE(OrderInterface,1,0));
```

Figure 18. Registering the interface in the Monitor environment

Initializing the resource manager

The resource managers that we will work with follow the X/Open Specification for Distributed Transaction Processing and so are referred to as *XA-compliant* resource managers. Interaction with resource managers is described in detail in Chapter 6, "Interacting with a relational database," on page 53. For now, we simply add a call

to the **mon_RegisterRmi** function to our server. We pass this function a switch, which is initialized by the resource manager client library, and the name of the resource manager instance. This function registers the resource manager, informing Encina that it is an XA-compliant resource manager and can participate in transactions. The function returns a resource manager ID for the resource manager.

Figure 19 shows initialization of the resource manager.

```
int rmiId;
/* Set up interaction with a resource manager */
status = mon_RegisterRmi(&db_xa_switch, RM_NAME, &rmiId);
CHECK_STATUS(status);
```

Figure 19. Initializing the resource manager

Any server that deals with a resource manager must be recoverable. (Recoverable processes were described in "Distributed transactions and the two-phase commit process" on page 11.) This means that the server keeps track of transaction state information so that the state can be restored in the event of a failure. In the Monitor environment, the Monitor automatically makes a server recoverable if the server calls the **mon_RegisterRmi** function.

Initializing Encina

To initialize the server, we need only to call the **mon_InitServer** function. This function initializes the server and all underlying Encina Toolkit components. It also performs any of the initialization that we specified (such as making the server recoverable).

After this function has been called successfully, we can perform any additional initialization that relies on Encina having been initialized. Currently, our application does not need to perform any such initialization. However, in Chapter 5, "Using RQS," on page 43, we initiate communications with an RQS server, so we need to add a call to a function that does this initialization.

Listening for RPCs

To begin listening for RPCs, the application calls the **mon_BeginService** function. This function informs the Monitor that our server is ready to accept RPCs. If the **mon_InitServer** function is not called, the **mon_BeginService** function also initializes Encina. Our application uses the **mon_InitServer** function because later we will add an additional initialization step between the calls to **mon_InitServer** and **mon_BeginService**. The **mon_BeginService** function does not return until the server is shut down. Any cleanup work can be performed after the function returns.

The server application

Figure 20 on page 28 shows the complete server at this point. We have not modified the **OrderItem** function.

```
int main (void)
{
 unsigned32 status;
 int rmiId;
 extern struct xa_switch_t db_xa_switch;
 /* Register the interface */
 status = mon_InitServerInterface(
 MON_SERVER_INTERFACE(OrderInterface,1,0));
 CHECK_STATUS(status);
 /* Set up interaction with a resource manager */
 status = mon_RegisterRmi(&db_xa_switch, RM_NAME, &rmiId);
 CHECK_STATUS(status);

 /* Initialize the server and Encina */
 status = mon_InitServer();
 CHECK_STATUS(status);


 /* Later, we will add RQS initialization here */

 ...

 /* Begin listening for RPCs */
 ...
 status =  mon_BeginService();
 CHECK_STATUS(status);
 exit(0);
}
```

Figure 20. The Monitor application server

Writing the client

In our application, the client interacts with the user and makes a remote procedure call on the user's behalf. Because this manual is not about writing user interfaces, our user interface is simple; the user invokes the client with three command-line arguments: a customer ID, the stock number of the item to order, and the number of items to order.

In addition to implementing the user interface, the client must

- Initialize any Encina components it is using.
- Make the remote procedure call.
- Notify the user of the call's success or failure.

The **mon_InitClient** function initializes the Monitor client execution environment and all the Encina components needed by the client. Our application is using TRPC, which must be initialized. If we were not using the Monitor, we would have to call the appropriate TRPC initialization functions directly. However, the **mon_InitClient** function performs this initialization for us.

The **mon_InitClient** function takes two arguments: the name of the application and the name of the Monitor cell in which the application is running. In our application, the latter is obtained from an environment variable.

The client makes the RPC by calling the **OrderItem** function. Invoking this function looks very much like invoking a local function. The only difference is the first parameter, which is the binding string. In this example, the client uses a

binding string of "interface:". This specifies that it binds to any server that offers the desired interface. After the call to **OrderItem** returns, the client checks the return status and informs the user of the success or failure of the order.

The client code is shown in Figure 21.

```
int main(int argc, char ** argv)
{
 idl_ulong_int custId, stockNum, numOrdered;
 unsigned32 status;
 char *cellName;
 if (argc != 4){
      fprintf(stderr,
           "Usage OrderItem <custId> <stockNum> <numOrdered>\n");
      exit (1);
 }

 custId = strtoul(argv[1],NULL,10);
 stockNum = strtoul(argv[2],NULL,10);
 numOrdered = strtoul(argv[3],NULL,10);

 /* Determine the cell name */
 cellName = getenv("ENCINA_TPM_CELL");
 if (cellName == NULL){
      fprintf(stderr,
         "You must set the ENCINA_TPM_CELL environment variable.\n");
      exit(1);
 }

 /* Initialize the client */
 status = mon_InitClient("OrderItem", cellName);
 CHECK_STATUS(status);

 /* Invoke the RPC */
 status = OrderItem("interface:", stockNum, numOrdered, custId);
 if (status != SUCCESS){
      fprintf(stderr, "Order failed.\n");
 }
 else {
      fprintf(stderr, "Order processed.\n");
 }
 exit(0);
}
```

Figure 21. The client portion of the application

If the client wanted to bind to a specific server instead of any server offering the interface, it could specify that in the binding string by using the word `server:` in the binding string, followed by the name of the server. In that case, the call to the **OrderItem** function is specified as follows:

```
status = OrderItem("server:orderServer", stockNum, numOrdered, custId);
```

In this example, the name of the server, as specified when it was defined, is `orderServer`.

Using other Monitor features

This section describes several other Monitor features that can be used by simple applications. For more information on these and other features, refer to the *Encina Monitor Programming Guide* and *Encina Administration Guide Volume 1: Basic Administration*.

Load balancing and scheduling

The Monitor provides the following methods for simplifying load balancing and scheduling:

- Each application server can be assigned a priority.
- Each application server can consist of multiple processing agents.
- Each processing agent can be multithreaded.

The first two options require no additional programming. They are set when the application server is configured and started. Multithreaded servers require some additional programming.

Specifying a server priority

An administrator can specify a priority when starting a server. This priority is used by the Monitor to balance requests among application servers that export the same interface. Client requests are distributed over a group of application servers based on their priority. The priority is used only by clients that use transparent binding.

Setting the number of processing agents

A Monitor application server consists of one or more processes called *processing agents* (PAs), each of which executes the application server code. When configuring the application server, an administrator can specify that it use more than one PA. A single application server with several processing agents needs to be configured and started only once; the Monitor then manages the processing agents as a single group. The programmer does not need to be concerned with whether the server contains one PA or several; the use of multiple PAs is transparent to the program.

Using multithreaded PAs

A *thread* is a single sequential flow of control within a process. The processes we have used so far have consisted of single threads. However, DCE and Encina provide tools for creating multiple threads within a single process. These threads can be thought of as lightweight processes sharing the same address space. Although the operating system continues to view a multithreaded process as a single process, this process can perform several operations concurrently.

The PAs in our Monitor application server currently service one client at a time. To use Monitor terminology, they are *exclusive* PAs. Monitor PAs can also serve multiple clients at the same time. Such PAs are called *concurrent shared* PAs. For such PAs, the Monitor automatically starts a new thread to handle each incoming RPC.

An application server, as part of its initialization, can specify whether its processing agents permit shared access. The default permits only exclusive access. The **mon_SetSchedulingPolicy** function specifies whether a server permits shared access of its PAs.

If we specify the use of concurrent shared PAs, the Monitor automatically makes the application server multithreaded. However, multithreaded servers can be a problem with databases that are not thread-safe (that is, that are not designed to correctly isolate the actions of one thread in a process from other threads in that process). Moreover, all threads in a process share any global variables used by the process; thus, a multithreaded process that uses global variables might need to take steps to synchronize access to these variables. Due to these additional concerns, we are not adding multithreading to our server.

Security

The Monitor automatically uses a number of the DCE security features. We do not need to modify the application to use security in the Monitor environment. The administrator specifies what level of security to use when starting the application server. There are several security features that we can use:

- *RPC authentication*. The administrator who starts the server specifies an RPC authentication level for the interface. The Monitor rejects any RPCs from clients that do not specify at least that level. A Client can use the **mon_SetRpcProtectionLevel** function to set the protection level for the next RPC it makes.
- *Access control lists* (*ACLs*). ACLs can be placed on an entire interface or on individual functions in an interface. The Monitor rejects RPCs from clients that are not granted access by the appropriate ACL.

Note that in in both of these cases, the server program does not need to do anything. The server never sees the rejected RPCs.

In addition, clients can set a *login context*, specifying on whose behalf an RPC is being made. Clients can set their login context using the **mon_BindSetLoginContext** function.

For more information on security, see Chapter 1, "Basic concepts of distributed computing," on page 1.

Using data-dependent routing

In the example program, there can be multiple copies of the same server running. All copies are identical; all offer the same interface. If the client binds by interface (that is, if it specifies ″interface:″ in its binding string), the RPC it makes goes to any of the servers. However, there are applications in which servers offer the same interface, but for which we want the RPC to go to a particular server. For example, if our application were busy enough we might want to split the database into multiple databases—one for stock numbers 1 through 1000, the next for stock numbers 1001 through 2000 and a third for stock numbers 2001 through 3000. Different copies of the application server interact with each. The servers are still identical, but each server receives only the RPCs that hve stock numbers within a specified range.

One way to do this is would be to modify the client. It could examine the stock number, then specify a server in a binding string. However, we can instead simply change the binding method we use and allow the Monitor to route the RPC to the correct server. Routing of RPCs based on the value of one or more of the parameters is called *data-dependent routing*.

To use data-dependent routing, we must do three things:

- Create a *routing table*, which specifies which server an RPC is to be routed to based on the data in the RPC.
- Modify the TACF to specify which parameter or parameters serve as the *routing key* (that is, the value upon which the routing decision is to be made).
- Modify the client to specify that it is binding based on a routing table and not based on interface.

A routing table simply specifies values or ranges of values and a binding string based on those values. Figure 22 on page 32 shows an example of a binding table. In this example, RPCs that have a key value of 1 to 1000 go to orderServer1. Those

in which a key value is 1001 to 2000 go to orderServer2, and those with a key value of 2001 or higher go to orderServer3. The square brackets ([and]) indicate that the range is inclusive. That is, 1 and 1000 are included in the first range.

```
orderBindingTable =
{
[1,1000]:     "server:orderServer1"
[1001,2000]:  "server:orderServer2"
[2001,max]:   "server:orderServer3"
}
```

Figure 22. An example binding table

We must modify the TACF to specify the routing key. To specify a routing key, we must include the data types and parameter names for the parameters that make up the routing key. For our example, the modified TACF specifies that the key for the **OrderItem** function is the *stockNum* parameter. This is shown in Figure 23. The name of the parameters for the routing key (in this case, *stockNum*) must match the names specified in the TIDL file.

```
/* TACF for OrderInterface */
[explicit_handle (encina_handle_t void)]
interface OrderInterface
{
      [comm_status, fault_status] OrderItem();
      [routing_key (unsigned long, stockNum)] OrderItem();
}
```

Figure 23. TACF for use with data-dependent routing

Finally, we change the call to the **OrderItem** function to specify the binding table, as shown in the following example:

```
status = OrderItem("table:orderBindingTable", stockNum, numOrdered, custId);
```

Routing tables are created administratively by using either Enconsole or **enccp**. See the online help for Enconsole or *Encina Administration Guide Volume 3: Advanced Administration*. For more details on data dependent routing and the use of routing tables, see the *Encina Monitor Programming Guide*.

Using delegation

Authorization decisions are based on the principal of the initiator of the request. However, when a Monitor application server makes an RPC, it does so on behalf of the server's principal, not the client's principal. *Delegation* enables a server to make an RPC on behalf of the client that initiates the call, preserving the identity of the initiator of the operation. A *delegation chain* is the collection that consists of the initiator and one or more intermediaries. Monitor clients can turn on delegation for an RPC by calling the **mon_SetDelegation** function. A server can determine the current delegation status using the **mon_GetCallerDelegationType** function.

For more information, see the *Encina Monitor Programming Guide*.

Making the server a client of another server

A Monitor application server, like any other server, can also act as a client of other servers. Our application server acts as a client of an RQS server. It also acts as a client of another Monitor application server. The use of RQS with our application is described in Chapter 5, "Using RQS," on page 43. In this section, we address what we need to do to enable our application server to make RPCs to another application server. In particular, our *OrderInterface* server makes TRPCs to the server that exports the billing interface, which in turn uses PPC to communicate with a mainframe.

To add a new application server to the application, we must perform two primary tasks:

- Define and implement the interface for the new server.
- Implement the new billing server.

We do not have to perform any client-specific initialization, such as calling the **mon_InitClient** function. This initialization is performed automatically as part of server initialization.

To distinguish between our two application servers, we call our initial application server, which exports the order interface, the order application server (or simply the *order server*). We call the new application server the billing application server (or simply the *billing server*).

Defining the interface

To define the billing interface, we must create a TIDL file and, because we are using Monitor universal binding again, a TACF. Except for the interface name, the TACF is the same as that for the order interface.

The new TIDL file defines one function, **BillForItem**. This function takes two arguments: a customer ID and a dollar amount. It does not return status. If the server that implements this function encounters an error condition (such as insufficient funds to cover the cost of the items ordered), it simply aborts the transaction (see Chapter 4, "Making the sample application transactional," on page 37).

The new TIDL file is shown in Figure 24 on page 34. Note that the function is defined as transactional. This means that a TRPC is made to the new server, which enables the new server to abort the transaction if it encounters an error and guarantees that the billing operation adheres to the ACID properties.

```
[
uuid(002513cc-da57-1ec0-8a0f-9e620a3aaa77),
version(1.0)
]
interface BillingInterface
{
import "tpm/mon_handle.idl";
[transactional] void BillForItem(
                                "interface:",
                                [in] unsigned long customerID,
                                [in] unsigned long amount);
}
```

Figure 24. TIDL file for the billing interface

Implementing the new application server

Implementing the new application server involves two steps:

1. Performing the required initialization steps
2. Implementing the **BillForItem** function

At this point, the server initialization consists only of registering the interface and initializing Encina. Initializing Encina is the same as for the order server. Registering the interface is shown in Figure 25.

```
/* Register the interface */
status = mon_InitServerInterface(
                        MON_SERVER_INTERFACE(BillingInterface,1,0));
CHECK_STATUS(status);
```

Figure 25. Registering the billing server interface

Figure 26 shows the skeleton of the **BillForItem** function. This function makes a PPC connection to a mainframe to check the customer's available funds and debit the customer's account. As with previous functions, this function does not return status; if it detects a failure, it aborts the transaction. We complete the function in Chapter 7, "Using Encina Peer-to-Peer Communications," on page 61.

```
void BillForItem(ild_ulong_int customerId,
 idl_ulong_int amount)
{
 /* Use PPC to check for the amount on a mainframe */
 /* To be filled in in the PPC chapter  */
}
```

Figure 26. The BillForItem function

Notes on building and running the application

At this point, our application consists of the following eight files:

- **OrderInterface.tidl** — the TIDL file for the order interface
- **OrderInterface.tacf** — the TACF file for the order interface
- **OrderServer.c** — the code for the Order Server
- **BillingInterface.tidl** — the TIDL file for the billing interface
- **BillingInterface.tacf** — the TACF file for the billing interface

- **BillingServer.c** — the code for the billing server
- **OrderItem.c** — the code for the client
- **order.h** — an application-specific header file

Both the client and the order server must also include the **OrderInterface.h** file generated by the IDL compiler and the **order.h** file, which includes definitions used in both the client and the server. In addition, both the client and the server must link with the **encina** and **dce** libraries.

The client must include the **tpm/mon_client.h** file. The order serverand the billing server must include the **tpm/mon_server.h** file. The client must link to the Monitor client library, **EncMonCli**. Both the order server and the billing server must link to the Monitor server library, **EncMonServ**. Because this library is a superset of the **EncMonCli** library, the order server does not have to include the Monitor client library.

To start a Monitor application, use the Encina Enconsole utility. Enconsole provides an interactive environment for defining and starting Monitor cells and servers. You must first define the application server to the Monitor and then start it. You can set tracing options when defining the server. For more information on starting servers, see *Encina Administration Guide Volume 1: Basic Administration*.

Before the client is started, the ENCINA_TPM_CELL environment variable must be set to the name of the Monitor cell in which the application server is running. To run the client, simply specify the name of the client, followed by the three arguments for the application. For example, the following command orders 15 instances of the item with stock number 5 for the customer with customer ID 2:

```
% OrderItem 2 5 15
```

Chapter 4. Making the sample application transactional

This chapter builds on the sample application that was started in the previous chapter by making the application transactional. It discusses the steps that must be followed to add transactional capabilities to the server. It describes the Encina Tran-C language and the various ways that aborted transactions can be handled.

Making the application transactional

The server divides the concept of ordering an item into three steps: the actual ordering of the item, placing the order in the shipping queue, and billing the customer. However, at this point in the development of our sample application, if one function fails, the results of previous functions are not undone. For example, if the billing operation fails, the inventory database is still updated and the item is still shipped. We must modify the server so that these three actions are tied together in such a way that either all happen or none happens. In other words, we must group these three operations into a single transaction.

As described in Chapter 1, "Basic concepts of distributed computing," on page 1, a transaction is a group of actions that adheres to the ACID properties. That is, a transaction is atomic, consistent, isolated, and durable. In the context of our application, "atomic" means that either all three operations happen or none does. "Consistent" means that if we deduct three items from the database, we ship and bill for three items. "Isolated" means that, in a multiclient environment, work our transaction does for one client is invisible to transactions from other clients until our transaction ends. "Durable" means that after the transaction commits, all actions happen, even in the event of system failures; for example, even if a disk fails, our changes are recorded in the database.

We must do three things to make the server transactional:

- We must make the server recoverable.
- We must modify the initialization steps so that our server initializes the parts of Encina needed for transactions. Depending upon the component we use for this, we may also have to modify the way the server terminates.
- We must delimit (indicate the beginning and end of) the transaction, specifying which operations are to be part of the transaction.

We already peformed the first step. The call to the **mon_RegisterRmi** function, described in Chapter 3, "Writing a Monitor client/server application," on page 23, automatically made the server recoverable. If we were not running in the Monitor environment, we would have to initialize and use a recovery service (such as the Encina Recovery Service). This in turn would require us to use a log service.

The final two steps are related: the initialization steps we need depend upon which Encina component we use to delimit transactions. The Encina Toolkit provides several transactional interfaces. In this manual, we use two of them, Tran-C and TX (the X/Open transactional interface). In this chapter, we use Tran-C. In Chapter 8, "Using TX," on page 73, we will modify the application to use TX instead.

The Monitor automatically initializes Tran-C, so we have no other initialization to perform. However, any application that uses Tran-C transaction constructs (as our server does) must register the name of the module (typically the name of the

source file) with the Tran-C run time by using the **inModule** statement. If we were not using the Monitor, we would have to explicitly initialize Tran-C, as described in the *Encina Transactional Programming Guide*.

After we perform these steps, Encina handles all the details of transactions and two-phase commit processing (see Chapter 1, "Basic concepts of distributed computing," on page 1) for us. We do not have to write the code that coordinates processing if the transaction commits or that rolls back changes if it aborts. Encina does this for us. For example, if we abort the transaction after having updated the database and queueing a shipping request, Encina rolls back changes to the database and removes the shipping request from the queue.

Specifying which operations are part of a transaction

The final step in adding transactions to our server is to specify which operations are part of the transaction. To do this, we must delimit the transaction, specifying where the transaction begins and where it ends. We use Tran-C to accomplish this.

Tran-C provides a number of features used by transactional applications. It provides several constructs for delimiting transactions. It also automatically associates a transaction with a thread of execution and handles the flow of control, automatically transferring execution to the appropriate location if a transaction aborts. It provides a number of additional features, all of which are described in the *Encina Transactional Programming Guide*. We use some of these other features in later chapters.

The Tran-C construct that delimits the transaction in the server is the **transaction** construct. The **transaction** construct delimits a transaction and provides a mechanism for specifying the transfer of control when the transaction commits or aborts. The **transaction** construct is shown in Figure 27.

```
transaction{
      /* Operations that are part of the transaction go here */

}onCommit{
      /* Operations to perform after the transaction commits go here */

}onAbort{
      /* Operations to perform after the transaction aborts go here */

}
```

Figure 27. The Tran-C transaction construct

In the construct, the keyword **transaction** indicates the start of the transaction. When the flow of control reaches the closing brace for this construct, Encina tries to commit the transaction. If the transaction commits, the flow of control proceeds into the **onCommit** clause. If at any point the transaction is aborted (by the current process or by any other process in the transaction), control automatically transfers to the **onAbort** clause. If the transaction aborts, we can use several Tran-C functions to get more information about why the abort occurred.

Figure 28 on page 39 shows the **OrderItem** function after we have modified it to use the **transaction** construct to group its three functions into a transaction.

```
error_status_t OrderItem(idl_ulong_int stockNum,
                         idl_ulong_int numOrdered,
                         idl_ulong_int customerId)
{

 idl_long_int returnStatus;
 idl_long_int costPerItem, totalCost;
 short priority;

 transaction{
      PlaceOrder(stockNum, numOrdered, &costPerItem);
      totalCost = numOrdered * costPerItem;
      if (totalCost > 1000)
           priority = HIGH_PRIORITY;
      else
           priority = NORMAL_PRIORITY;

      PlaceItemOnQueue(stockNum, numOrdered,
                       customerId, priority);

      BillForItem(customerId, totalCost);
 }onCommit{
      fprintf(stderr, "We committed.\n");
      return SUCCESS;
 }onAbort{
      fprintf(stderr, "We aborted. %s\n", abortReason());
      return ORDER_FAILED;
 }
}
```

Figure 28. Adding transactions to the server

Note that the functions that make up the transaction do not return status codes. This does not mean that the server is no longer performing error checking. Instead, the called functions abort the transaction if they discover an error. Control then switches to the **onAbort** clause. The server prints an error message, including the text that the aborting operation specified when it aborted, which is returned by the **abortReason** function. This is described in more detail in "Aborting transactions."

Aborting transactions

Tran-C provides functions for explicitly aborting a transaction. If it detects a failure, the server uses an abort function to abort a transaction. For example, if a billing error occurs, the server aborts the transaction; all actions in the transaction, such as changes to the database, are then rolled back to the state before the transaction.

In this manual, we use the following two Tran-C functions for aborting transactions:

- **abort**
- **abortWithCode**

Both functions abort the transaction. The difference between them concerns how you specify an abort reason.

The **abort** function provides a simple abort mechanism. It takes one argument, a string. This string is simply passed to applications that are notified of the abort. These applications use the **abortReason** function to retrieve the string.

The **abortWithCode** function provides a more flexible abort mechanism. It takes one argument, an abort code. Applications that are notified of the abort can take action based on this code, including converting the code to a string.

These two mechanisms are described in somewhat more detail in the following sections. Note that at this point our server does not explicitly abort transactions. However, it must be able to handle aborts generated by other parts of the application or by the underlying RPC mechanism. In the next chapter, when we add a second server, that server *will* explicitly call an abort function.

Aborting with strings

The **abort** function aborts a transaction. The application aborting the transaction can pass a string specifying the reason for the abort. All applications that are notified of the abort can retrieve this string.

This method is used by our server as it now stands. If any part of the application aborts the transaction, control is transferred to the **onAbort** clause. Our application uses the **abortReason** function to determine why the transaction aborted. This function returns whatever string the aborting component specified as an abort reason.

Figures Figure 29 and Figure 30 show an example of a call to the **abort** function as it might be made by some part of our application. It also shows how our server uses the **abortReason** function to display the reason for the abort before returning from the RPC. Note that in our example application, the call to the **abortReason** function occurs in the order server (that is, in the server that we have been developing up to this point). The call to the **abort** function occurs in the billing server, which we develop in the next chapter. In general, the abort can be initiated by any process in the distributed transaction, including the process that started the transaction.

```
/* Billing Server */
abort("Insufficient funds");
```

Figure 29. Aborting a transaction

```
/* Order Server */

transaction{
 .
 .
 .
}onAbort{
     fprintf(stderr, "Transaction aborted. %s.\n", abortReason());
     return ORDER_FAILED;
}
```

Figure 30. Handling an aborted transaction

If the transaction aborts, the order server writes the following string to the standard error stream:

```
Transaction aborted. Insufficient funds.
```

Aborting with an abort code

Encina applications can also use an abort code rather than an explicit abort string. The application that receives this abort code can then take action based on this code, including converting it to a string. Abort codes offer several advantages over strings:

- It is easier to compare codes (which are integers) than it is to compare strings. Thus, an application can more easily take action based on a code.
- Codes can be translated into strings by using external catalogs of messages. It is thus easier to internationalize programs using codes.

The abort code interface is somewhat more complex than the string interface. Thus, for applications whose only response to an abort notification is to print a string and exit, aborting with strings can be a better approach, especially if such applications are used in only one language. For more information on aborting using abort codes, see Appendix B, “Using abort codes,” on page 87.

Notes on building and running the application

Both the client and the server must continue to include the header files specified in Chapter 3, “Writing a Monitor client/server application,” on page 23. In addition, the server must include the Tran-C header file, **tc/tc_server.h**.

Chapter 5. Using RQS

This chapter discusses using the Encina Recoverable Queueing Service (RQS) with our example application. RQS allows applications to queue transactional work to be completed at a later time. Applications can then commit any existing transactions with the assurance that the queued work will not be lost. RQS guarantees that after an element has been added to a queue and the transaction has committed the element remains in the queue until dequeued by another transaction.

An overview of RQS

Transactions are isolated from one another. In general, this means that other transactions are prevented from accessing data that a transaction is using until the transaction is complete. For example, a transaction can lock records in a database, preventing other users from accessing those records until the transaction commits or aborts. As a result, a design goal of transactional applications is to complete transactions quickly, unlocking locked data and allowing access to it by other transactions as quickly as possible. This can be a problem, however, when one part of the transaction takes much longer than other parts.

In our sample application, for example, the steps in ordering an item are accessing the database, shipping the item, and billing for the item. Accessing the database and billing for the item are relatively quick operations (in this case, billing means debiting against a credit database). Shipping the item, however, can take longer. We need an approach that guarantees shipping will happen but allows the transaction to commit without waiting. For example, it is preferable not to wait for a person in the shipping warehouse to acknowledge the shipping request.

We can solve this problem by using the Encina Recoverable Queueing Service (RQS). Instead of waiting for a reply from shipping, our server simply places a request on an RQS queue. It then commits with the assurance that the shipping request will not be lost; the request remains on the queue until it is dequeued by an application in the shipping department.

Encina also provides a way for applications to queue TRPCs, which can be forwarded to Monitor application servers. The *Queued Request Facility* (*QRF*) enables applications to queue data along with the name of a Monitor application server and a function name. QRF monitors specified queues and forwards requests to the specified application server. Our application does not use QRF, so we do not discuss it further in this manual. For more information, see the *Encina Monitor Programming Guide*.

Queues and elements

Queues are linear data structures that can be used to pass information from one application to another. Applications *enqueue* (add) elements to the tail of a queue and *dequeue* (remove) elements from the head of a queue in a first-in first-out (FIFO) manner. This is shown in Figure 31 on page 44.

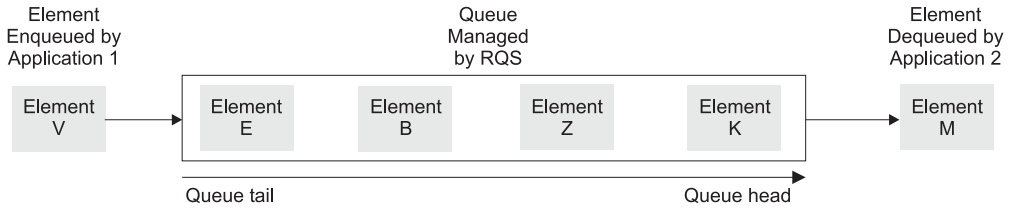

Figure 31. FIFO behavior of queues

Enqueueing and dequeueing are performed from within the scope of a user transaction. RQS guarantees that after an element has been added to a queue and the transaction has committed, that element remains in the queue until dequeued by another transaction. Successfully enqueued elements are not lost due to system failures, media failures, or failed dequeue attempts. If the dequeueing transaction is aborted, the element is returned to the queue.

Each queue is maintained by one and only one RQS server. All interactions with that queue are handled by the server. An application queues an element, for example, by making an RPC to the RQS server, which then places the element on the queue.

Client applications use queues to store data in the form of elements. An *element* contains record-oriented data specific to an application. The fields of an element store related pieces of the data. For example, a shipping element might have fields for storing the customer ID, the item number, and the number ordered.

Each element must have a type, which is specified when the element is queued. An *element type* is a named specification that defines the data type and size for each field of an element. Element types are independent of queues; elements of different element types can be queued and dequeued from the same queue. Types are typically defined administratively, although they can also be defined programmatically. RQS provides a number of field types that can be used when defining an element type.

Prioritization and queue sets

An application might want to specify that certain elements be dequeued before other elements. That is, it might want to specify a priority for an element. However, when an application dequeues an element from a queue, it dequeues the next available element, not the "most important" element. To prioritize elements in RQS, an application uses queue sets.

A *queue set* is a collection of queues. Inside the queue set, queues are ranked based on a *priority class*. Elements can be queued to the specific queue in the queue set that has the appropriate priority class. The dequeueing application then dequeues an element from the set as a whole; RQS returns an element to the application from a queue based on the priorities. Priorities can be weighted (using a concept called service levels, as described in the *Encina RQS Programming Guide*) so that the dequeueing application does not always dequeue the highest-priority element first. However, in our example, we will use a strict prioritization scheme.

Note that if a dequeueing application does not want to use the queue set, it can still dequeue an element from an individual queue in the set. Multiple applications can be used to dequeue elements from the queues in a set. For example, one

application can dequeue elements from the set as a whole (as our sample dequeueing application does) while other applications can dequeue elements from specific queues.

Adding RQS to our application

Our application will use RQS to queue shipping requests. In this way, the order transaction can commit without waiting for the shipping department to actually process the request but with assurance that the request will be processed. Specifically, we will do two things:

1. Modify the order application server so that it queues the shipping request to an RQS queue.
2. Write a simple application to dequeue the requests from the shipping queue and process them.

The parts of the application that we develop in this chapter are shaded in Figure 32.

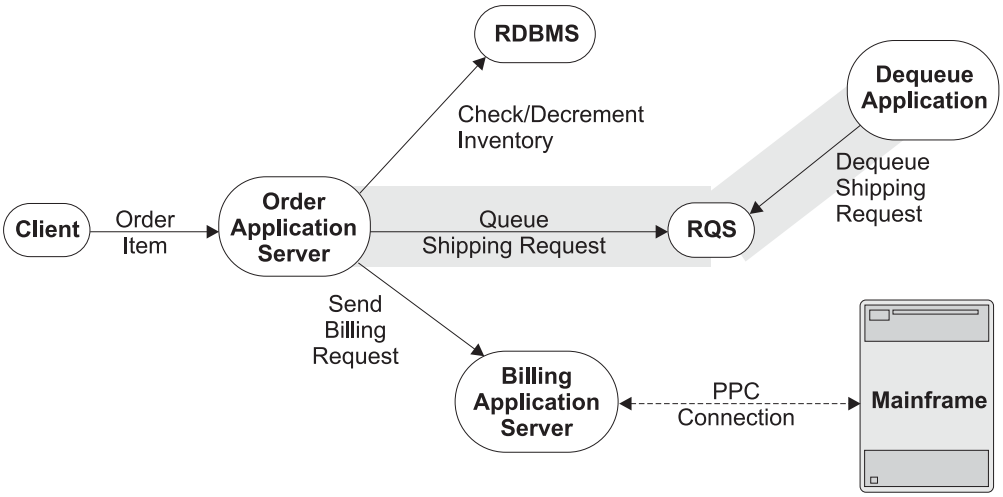

Figure 32. Sample Order-Entry Application: Adding RQS

The application server queues a shipping request to one of two queues—a normal-priority queue (**normalShippingQueue**) or a high-priority queue (**priorityShippingQueue**)—based on the value of the order. These two queues form a queue set called **shippingQueueSet**. The dequeueing application dequeues from the queue set, which has been set up such that dequeues occur following a strict prioritization scheme. Requests on the high-priority queue are returned to the dequeueing application before requests on the normal-priority queue. This is summarized in Figure 33 on page 46.

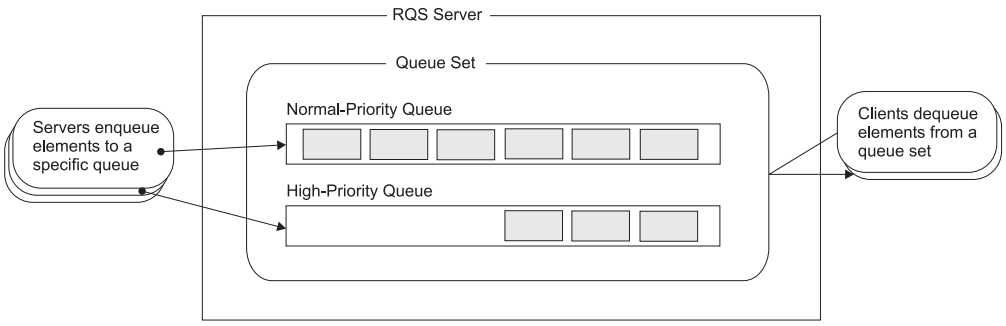

Figure 33. Sample application's use of an RQS queue set

Although RQS provides administrative functions for creating queues, queue sets, and element types, we will not use these in our program. Instead, we will assume that the queues and element types have already been created and that the two queues have already been placed in a queue set, either using another program or using the **rqsadmin** command. "Building and running the sample application" on page 51 shows several example **rqsadmin** commands for doing so. For more information, see the *Encina RQS Programming Guide* and the *Encina Administration Guide Volume 2: Server Administration*.

Queueing a shipping request

In previous chapters, our application server called a function called **PlaceItemOnQueue** to queue a shipping request. In this section, we write this function. We also write two other functions: one to initiate communications with the RQS server and one to close communications. These functions will be called during application server initialization and termination, respectively. These three functions will be grouped into one source file, allowing all three to access a single global static variable used to hold the handle to the RQS server (see "Getting a handle to an RQS server" on page 47).

Defining the element type

All elements used in RQS are of types defined by a user. The element type specifies the layout of fields in the element. When an application enqueues an element, it specifies the type of that element. When it dequeues an element, the type is returned to the dequeueing application, providing it with the information it needs to interpret the field layout.

We must define the element type or types that our program is going to use. These types must then be created at the RQS server, either programmatically or by an administrator. Because creation of element types is generally an administrative task, we are going to assume that the element types, like the queues themselves, have been created by an administrator prior to program execution. "Building and running the sample application" on page 51 shows the **rqsadmin** commands that create the necessary queues and element types.

In our application, all elements queued by the order server have three fields:

- The customer ID (*customerId*)
- The stock number of the item ordered (*stockNum*)
- The number of items ordered (*numOrdered*)

Our application needs only one element type, **shippingType**, which contains these three fields. The type of each of the fields is `rqs_unsignedInt32`, the RQS field type corresponding to the DCE idl_ulong_t data type. The layout of the **shippingType** element type is shown in Figure 34.

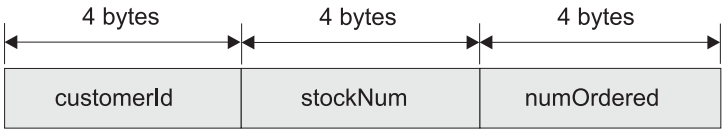

Figure 34. The shippingType element type

Getting a handle to an RQS server

Before we can add elements to a queue, we must first obtain a handle to the RQS server that controls the queue. This is done by using the **rqs_GetServerHandle** function.

Our application can be designed such that each time an element is enqueued, the enqueueing function would first get a handle to the server, and then queue the element. However, because each queue is under the control of one and only one RQS server, our application would obtain a handle to the same server each time. Thus, to reduce the overhead of having to obtain a handle (which involves looking up the server in the Cell Directory Service) each time an item is shipped, the server obtains this handle just once.

During initialization, the order server calls a function we call **ConnectToShippingServer**. This function obtains the handle to the RQS server and saves it in a global static variable, which is accessible to the other functions in the source file, including the function that enqueues shipping requests. Figure 35 shows this function.

```
/* Global Variable */
static rqs_serverHandle_t rqsHandle;

error_status_t ConnectToShippingServer(void)
{
 rqs_status_t status;

 status = rqs_GetServerHandle(RQS_SERVER_NAME, &rqsHandle);
 if (status != RQS_SUCCESS)
      return RQS_FAILURE;
 else
      return SUCCESS;
}
```

Figure 35. Getting a handle to the RQS server

The **rqs_GetServerHandle** function takes one input argument: the name of the RQS server. To simplify things, the order server uses a constant string (`RQS_SERVER_NAME`) to specify the RQS server name; we define the `RQS_SERVER_NAME` in a header file used by the application. The **rqs_GetServerHandle** function returns a handle to the RQS server. This handle is then used in RPCs to that server. In particular, the order server uses the handle when it enqueues a shipping request. Because the order server cannot function if it cannot queue shipping requests to RQS, it exits if it cannot obtain a handle to the RQS server.

The **ConnectToShippingServer** function is called during initialization, after Encina has been initialized but before we begin listening for RPCs. This is shown in Figure 36. When the server terminates (when the **mon_BeginService** function returns), the server calls the **DisconnectFromShippngServer** function, which simply calls the **rqs_FreeServerHandle** function to free the server handle.

```
int main()
{
...
/* Initialize Encina */
mon_InitServer();

/* Initialize interaction with RQS */
status = ConnectToShippingServer();
if (status != SUCCESS){
      fprintf(stderr, "Unable to connect to RQS server.\n");
      exit(1);
}

/* Begin listening for RPCs */
mon_BeginService();
DisconnectFromShippingServer();

}
```

Figure 36. RQS initialization

Adding the shipping request to the queue

Our server adds an element to one of the shipping queues by calling the function **PlaceItemOnQueue**. The server, prior to calling this function, determines whether the request is high or normal priority, based on the value of the order (greater or less than $1000). The server then calls this function with four parameters: the three items that make up the element to be queued and a parameter that specifies which queue (high priority or normal priority) is to be used. This function in turn places the item on an RQS queue using the **rqs_Enqueue** function.

The **PlaceItemOnQueue** function is passed three separate parameters corresponding to the three fields in the defined element type. We must combine these three parameters into a single element before we enqueue an element. That is, we must arrange them contiguously in memory, with no space between the fields. The fields must be in the order specified by the element type, and each field must take up its full width (which, for each of the three fields in our application, is 4 bytes).

There are various ways to combine the separate values into a single element. One common way is to define a structure that contains the three elements and use this structure as the element. This is a straightforward way of creating the element; however, although it works on most systems, it is not guaranteed to be portable by the ANSI C standard (which allows a compiler to insert padding bytes between fields in a structure). Because all three of our fields are of the same type, we could have simply created an array of three elements and copied the three elements to the array. However, we want to demonstrate a more general approach, which can be used in all cases. To provide a general solution while maintaining strict ANSI compliance in our example, we copy the three parameters into a buffer using the C **memcpy** function. This is shown in Figure 37 on page 49.

```
idl_ulong_t itemLength = sizeof(customerId)
                      + sizeof(stockNum)
                      + sizeof(numOrdered);


/* Allocate the item to queue and pack fields into it */
itemToQueue = malloc(itemLength);

/* Start copying at start of itemToQueue buffer */
itemCursor = itemToQueue;

memcpy(itemCursor, customerId, sizeof(customerId));
itemCursor += sizeof(customerId);
memcpy(itemCursor, stockNum, sizeof(stockNum));
itemCursor += sizeof(stockNum);
memcpy(itemCursor, numOrdered, sizeof(numOrdered));
```

Figure 37. Initializing the element

Each call to the **memcpy** function copies 4 bytes. The first call copies the *customerId* parameter to the first 4 bytes of the element. The second call copies the *stockNum* parameter to the next 4 bytes of the element; the third call copies the *numOrdered* parameter to the final 4 bytes of the element.

After the element has been set up, we enqueue it with the **rqs_Enqueue** function. The application specifies the following information when calling this function:

- The handle returned by the **rqs_GetServerHandle** function
- The name of the queue: **normalShippingQueue** or **priorityShippingQueue**
- The element type: **shippingType**
- The element we are enqueueing and its length

The **rqs_Enqueue** function returns an element identifier (ID) for the enqueued element. Because our server does not access elements again after queueing them, it does not use this returned ID. Figure 38 shows the call to the **rqs_Enqueue** function.

```
if (priority == HIGH_PRIORITY)
      queueName = "priorityShippingQueue";
else
      queueName = "normalShippingQueue";


status = rqs_Enqueue (rqsHandle, queueName,
                      elementType, itemLength,
                      itemToQueue,
                      NULL, /* Ignore work accumulation */
                      &elementId);
if (status != RQS_SUCCESS){
     abort("Enqueue attempt failed.");
}
```

Figure 38. Enqueueing the shipping request

This function must be called from within the scope of a transaction (for example, from within a Tran-C **transaction** construct). In our application, the transaction is started by the server before it calls the **PlaceItemOnQueue** function.

Dequeueing a shipping request

The application that dequeues shipping requests is a stand-alone RQS client. The same application interacts with a user and dequeues elements from the RQS queue set, with no intervening application server. In this section, we write only that portion of the application that interacts with RQS. A simple but complete RQS client program is available with the documentation.

Our dequeueing application must do the following three things:

1. Obtain a handle to the RQS server.
2. Dequeue elements from a queue set.
3. Free the handle to the server.

As was the case with the order server, the dequeueing application obtains a handle to the server once, storing it in a global static variable. It then loops, dequeueing requests until the user decides to terminate. It then frees the handle to the server.

Obtaining and freeing the handle are performed using the same functions that were used by the order server. The rest of this section describes dequeueing an element from a queue set, which we do by using the **rqs_QSDequeue** function.

When we enqueued an element, we had to select a queue in the queue set: the high-priority queue or the normal-priority queue. However, when we dequeue from a queue set, we do not have to specify this. We simply specify the name of the queue set. Because our queues are using strict prioritization, RQS first checks the high-priority queue. If there is an element on that queue, RQS returns it to the application. If not, RQS checks the normal-priority queue.

We must specify what RQS does if there is no element on any queue in the queue set. RQS can either wait until an element is available or return with a status code indicating that no element is available. Our application waits for an element if none are available. To do this, it specifies `TRUE` as the value of the *blockOnEmpty* argument of the **rqs_QSDequeue** function.

We must specify whether RQS should delete the element from the RQS server after we have successfully dequeued it. To optimize moving elements between queues, RQS allows applications to specify that an element not be deleted from the server when it is dequeued. This allows the element to be requeued to a different queue without having to actually resend the element to the RQS server. An element that has been dequeued and is not now on any queue but that has not been deleted from the server is called an *orphan*. Because our application is not requeueing elements, it specifies `rqs_DeleteElement` as the value of the *deleteOption* argument of the **rqs_QSDequeue** function. An element is thus deleted from the server after our application has successfully dequeued it.

The **rqs_QSDequeue** function returns a structure of type **rqs_elementDescriptor_t**. This structure contains the element and fields that specify the element type, the queue from which the element was dequeued, and so forth. However, because our application is using only one element type and is not moving elements between queues, it ignores all fields except the field that contains the element.

In general, applications do not know the size of the element that will be returned to them when they dequeue an element. An application might be processing several different element types, or the element types might contain varying-length fields. Thus, applications cannot allocate the **rqs_elementDescriptor_t** structure.

Instead, an application declares a pointer to an element of this type and passes the address of this pointer to the function that performs the dequeueing (in our case, the **rqs_QSDequeue** function). RQS then allocates the structure for the application. When the application is done with the structure, it must free the structure using the **rqs_Free** function, which frees memory allocated for the application by RQS. Our application must also follow this procedure, even though, in our simple example, we do know the exact size of the element to be dequeued.

When the element is returned, we must extract the three fields from it. We do this by using the C **memcpy** function in a fashion similar to the way we packed the fields into the element before enqueueing it. We place the fields in the three parameters that were passed to this function. The function return these three parameters to the calling application.

The complete **DequeueFromShippingQueueSet** function is shown in Figure 39. This function must be called from within a transaction that is started before the function is called.

```
error_status_t DequeueFromShippingQueueSet(idl_ulong_int *stockNum,
                        idl_ulong_int *numOrdered,
                        idl_ulong_int *customerId, short priority)

{
 rqs_elementDescriptor_t *itemDequeued;
 char *itemCursor;
 rqs_status_t status;

 /* Dequeue the element */
 status = rqs_QSDequeue(rqsHandle, "shippingQueueSet",
                            rqs_deleteElement,
                            TRUE, /* Wait for element to dequeue */
                            &itemDequeued);

 if (status != RQS_SUCCESS)
      return DEQUEUE_FAILED;

 /* Break the item into its three components. */
 itemCursor = itemDequeued->value;
 memcpy(customerId, itemCursor, sizeof(*customerId));
 itemCursor += sizeof(*customerId);
 memcpy(stockNum, itemCursor, sizeof(*stockNum));
 itemCursor += sizeof(*stockNum);
 memcpy(numOrdered, itemCursor, sizeof(*numOrdered));

 rqs_Free(itemDequeued);

 return SUCCESS;
}
```

Figure 39. Dequeueing the shipping request

Building and running the sample application

Our application now uses the following three new source files:

- **ShipItem.c** — the code for the part of the Order Server that interacts with RQS
- **DequeueRequest.c** — the simple client that dequeues shipping requests
- **orderRqs** — an application-specific header file containing RQS information used by both the Order Server and the dequeueing client

The order application server and the new dequeueing client must include the **rqs.h** header file. Both must link with the **EncRqs** library.

Before the application can be run, we must create the two queues, create the queue set, and add the queues to the queue set. This can be done with **rqsadmin** commands shown in Figure 40. These commands perform the following actions:

- Create two queues, a normal-priority shipping queue and a high-priority shipping queue.
- Create a queue set.
- Insert the two shipping queues into the queue set.
- Specify that we wish to use a strict (unweighted) priority scheme in the queue set.
- Create the element type we need for our application.

Note that when we insert the queues into the queue set, we assign a priority of 5 to the normal-priority queue and a priority of 1 to the high-priority queue. The absolute numbers do not matter as long as the normal-priority queue has a lower priority (a higher number) than the high-priority queue.

For information on the **rqsadmin** command, see *Encina Administration Guide Volume 2: Server Administration*.

```
% rqsadmin create queue \
 -server /.:/order_cell/server/rqsShippingServer \
 normalShippingQueue disableaccumulation
% rqsadmin create queue \
 -server /.:/order_cell/server/rqshippingServer \
 priorityShippingQueue disableaccumulation
% rqsadmin create qset \
 -server /.:/order_cell/server/rqshippingServer \
 shippingQueueSet
% rqsadmin insert queue \
 -server /.:/order_cell/server/rqshippingServer \
 shippingQueueSet normalShippingQueue 5
% rqsadmin insert queue \
 -server /.:/order_cell/server/rqshippingServer \
 shippingQueueSet priorityShippingQueue 1
% rqsadmin set servicelevels \
 -server /.:/order_cell/server/rqshippingServer \
 shippingQueueSet strict
% rqsadmin create type \
 -server /.:/order_cell/server/rqshippingServer \
 shippingType 3 customerId unsignedInt32 \
 stockNum unsignedInt32 numOrdered unsignedInt32
```

Figure 40. Using rqsadmin to create queues, queue sets, and element types

Chapter 6. Interacting with a relational database

Most applications need to deal with data. Often, this data is stored in a database. This chapter discusses the ways in which our application can interact with a resource manager and access data stored in a relational database.

Resource managers and distributed transaction processing

A *resource manager* (RM) controls access to a resource such as a database. A resource manager can be a relational database management system (RDBMS) or one of the Encina resource managers, such as SFS or RQS. In this chapter, we modify our application to allow it to interact with an RDBMS and access data stored in a relational database.

Resource managers and the XA specification

Our application must be able to access a relational database transactionally. Thus, the RDBMS must support transactional semantics. For example, if a transaction is aborted, any changes made to the database must be undone. The X/Open document *Distributed Transaction Processing: The XA Specification* describes what a resource manager must do to support transactional access. Resource managers that follow this specification are said to be *XA-compliant* and can participate in distributed transactions with Encina.

The XA Specification does not mandate any particular interface for starting and ending transactions or for interacting with a resource manager. Although X/Open does define several interfaces for accessing relational databases and for managing transactions, an application is free to use other interfaces, as long as those interfaces support the XA standard (which all of the Encina transactional interfaces support). The X/Open-defined interface to relational databases is SQL, which we use in this chapter and which is described in the next section. The defined transactional interface is TX, which we use in a subsequent chapter; in this chapter, we continue to use Tran-C.

SQL and embedded SQL

The Structured Query Language or *SQL* (pronounced “sequel”) is a fourth-generation language used by many RDBMSs. There is an ANSI/ISO standard that defines the language, as well as an X/Open specification based on the standard. Most RDBMSs also provide their own extensions to SQL. We will, as much as possible, use ANSI SQL. Our example can, with little change, be used with a number of different RDBMSs.

SQL can be used in a number of ways. The various RDBMS vendors provide interactive environments for issuing SQL statements, form-building products that use SQL, and so forth. However, in writing a program that accesses a database, we need to use SQL from within the program. To do this, we make use of *embedded SQL*, in which we embed SQL statements in a C program. ANSI/ISO and X/Open provide a standard way of embedding SQL statements in a program. This standard is followed by all the major SQL vendors and is used in our example.

A program that contains embedded SQL statements cannot simply be compiled because a compiler does not know how to process the embedded SQL statements. The program must first be processed by a SQL precompiler. A SQL *precompiler* is a

program that allows you to embed SQL statements in a C (or other high-level language) program. Each of the major database vendors provides a precompiler that can process a source program and translate the embedded SQL calls to RDBMS library calls. A precompiler produces a file that can then be compiled by a standard C compiler. This is illustrated in Figure 41. In this figure, the **PlaceOrder.precomp** file, which contains embedded SQL statements, is converted by the precompiler into a standard C source file (**PlaceOrder.c**). It is then compiled and linked to produce the executable file.

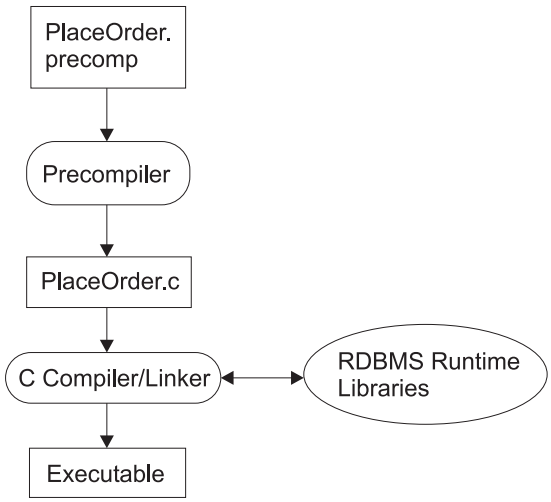

Figure 41. SQL precompiler input and output files

Modifying the application to interact with a resource manager

We have to do two things to modify our application to use a resource manager:

1. Register the resource manager.
2. Use SQL to read and write records in the database.

In Chapter 3, "Writing a Monitor client/server application," on page 23, we showed the function that is used to register the resource manager. At that point, though, we deferred discussing the function. In this chapter, we examine its use in more detail and explain the arguments that we must pass to it. Figure 42 on page 55 shows the part of the application we develop in this chapter.

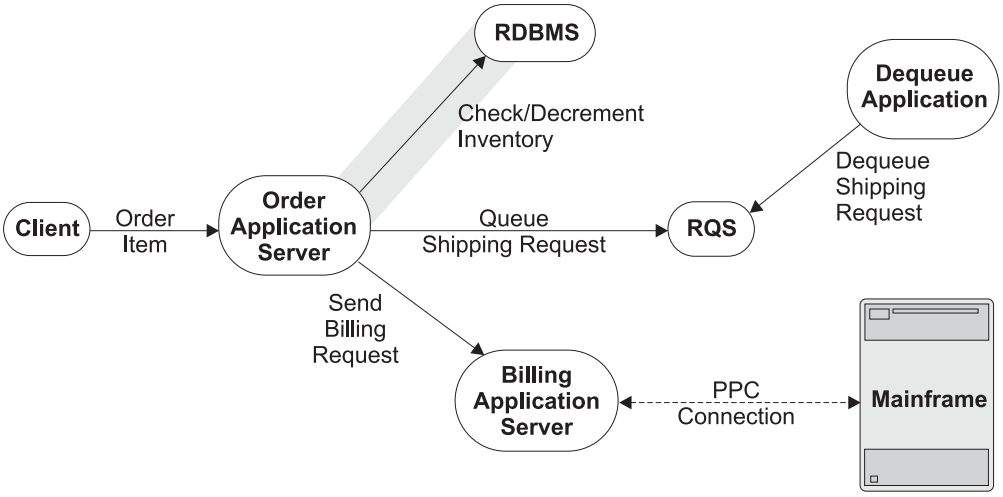

Figure 42. Sample application: interacting with a resource manager

To use the resource manager, we divide our application server code into two source modules:

- The **OrderServer.c** source file, which remains essentially as it was in the last chapter. The resource manager is registered in this file.
- The **PlaceOrder.precomp** source file, which contains all the code that interacts with the database. All embedded SQL is in this module; thus, this module is the only one that must be processed by the SQL precompiler before it is compiled. (Note that the file suffix depends upon the precompiler. Different precompilers for different RDBMSs use different suffixes.)

Registering the resource manager

The resource manager must be registered with the Monitor environment. This is done by using the **mon_RegisterRmi** function during Monitor initialization, prior to the call to the **mon_InitServer** function. We must supply the following information:

- The resource manager's XA switch. This switch is provided by the RDBMS vendor.
- The name of this instance of the resource manager. This name must match a name already configured with the Monitor. The Monitor uses the information already configured about this resource manager to register it.

The **mon_RegisterRmi** function returns an ID, which can be used in those SQL calls that take an interface ID. With some RDBMSs, this ID enables you to work with multiple instances of that RDBMS. In our application, we do not need this ID.

Typically, the RDBMS library exports the XA switch. We need to declare only an external variable of the appropriate name, as specified in the RDBMS documentation. For example, the Oracle library exports its XA switch under the name *xaosw*. The linker initializes this variable; we can use it in the call to the **mon_RegisterRmi** function. For example, to register Oracle as the resource manager, the call shown in Figure 43 on page 56 is used.

```
int rmiId;
extern struct xa_switch_t xaosw;
mon_RegisterRmi(&xaosw, "myOracle", &rmiId);
```

Figure 43. Registering a resource manager

Accessing the database

After the resource manager has been registered, we can begin to access the relational database. Embedding SQL code directly in an application is the simplest and most common way to access relational databases programmatically. It is the approach we will follow in our example in this chapter.

Although the embedded SQL interface is straight forward, it has the drawback that you can use only one resource manager: if you want to use multiple resource managers (or multiple instances of the same resource manager), there is no way to indicate which SQL code is for which manager or which embedded SQL statements are to be translated by which preprocessor. Calling the RDBMSs' library routines directly rather than using the preprocessor to translate embedded SQL into library calls is one way to use multiple resource managers. However, the functions used depend on which resource manager is used. Alternatively, statements that access different relational databases can be placed in different source files and precompiled separately. Each precompiler produces the calls needed for the appropriate database.

The database

Our example database consists of one table (called *inventory*). Each record in this table contains three fields: the stock number, the quantity, and the price. A more complex application would have more fields and tables. However, to be simple, we limit our example database to the three fields needed by our application.

Table 3 shows the definition of the three fields. We use a naming convention that uses underbars in the names of the fields (for example, *stock_num*) to distinguish fields from variables (which use uppercase letters to distinguish words, as in *stockNum*). Note that the field type name varies depending upon RDBMS vendor. For example, Oracle calls the INTEGER type NUMBER and Sybase calls it INT.

Table 3. Fields in the example SQL database

Field Name	Field Type
stock_num	INTEGER
num_available	INTEGER
item_cost	INTEGER

The inventory table is indexed on the *stock_num* field, enabling us to select records based on the stock number. We assume that the database and table exist prior to execution. A sample SQL statement to create the table is shown in "Building and running the sample application" on page 60.

Using embedded SQL

To use embedded SQL, our application needs to do the following things:

- Declare the variables needed in embedded SQL statements.

- Query the database using the given stock number to make sure that there are sufficient items in stock to satisfy the order. We then determine the price per item and return this information to the caller (which then passes this information to the billing server).
- Update the database (that is, decrement the database by the number ordered) if there are sufficient items in stock to place the order.

Embedded SQL allows us to freely use SQL statements in a C program. To embed SQL statements in a program, we begin each SQL statement with the **EXEC SQL** keywords.

Declaring the variables needed for SQL

Embedded SQL provides a method for declaring variables that are to be used in SQL statements. Such variables are called host variables. *Host variables* are variables declared in the host language (in our case, in C) and shared with the RDBMS. A program uses input host variables to pass information to the RDBMS. The RDBMS returns information to the program in output host variables. Host variables are declared in a *declare section*, which is delimited using the **DECLARE SECTION** keywords.

Host variables can be used in a C program in the same way as any other variables. They can also be used in any embedded SQL statements by preceding the name with a **:** (colon).

In our application, we use three host variables in embedded SQL statements. These must be declared in a declare section, as shown in Figure 44.

```
EXEC SQL BEGIN DECLARE SECTION;
      unsigned long stockNumber;
      unsigned long pricePerItem;
      unsigned long quantityAvailable;
EXEC SQL END DECLARE SECTION;
```

Figure 44. Declaring host variables in our application

Once these variables are declared, the *stockNumber* variable is initialized at the beginning of the function by assigning the *stockNum* parameter to it. The *quantityAvailable* and *pricePerItem* variables are initialized when they are read from the database; this is shown in Figure 45. The *pricePerItem* variable is assigned to the function parameter that returns this value to the calling program.

```
void PlaceOrder (idl_ulong_int stockNum,
                 idl_ulong_int numOrdered,
                 idl_ulong_int *costPerItem);
{
....
stockNumber = stockNum;
...

*costPerItem = pricePerItem;
}
```

Figure 45. Using host variables in our application

Querying the database

Our application uses the SQL **SELECT** statement to determine the quantity of the specified item currently in stock and the price of the item. It reads the values of

the *num_available* and *item_cost* fields into the *quantityAvailable* and *pricePerItem* host variables. This is shown in Figure 46.

```
EXEC SQL SELECT num_available, item_cost
      INTO :quantityAvailable, :pricePerItem
      FROM inventory
      WHERE stock_num = :stockNumber;
```

Figure 46. Embedded SQL for querying the database

Updating the database

Our application uses the SQL **UPDATE** statement to update the database to reflect the new quantity available after the item has been ordered. Prior to performing the update, the application first makes sure that there are enough items available to fill the order. If there are, the application calculates the number that will be available after the order has been fulfilled. It then performs the actual updating, setting the number of items available to the new value. This is shown in Figure 47.

```
if (quantityAvailable < numOrdered)
      abort("Insufficient quantity.");
quantityAvailable -= numOrdered;

EXEC SQL UPDATE inventory
      SET num_available = :quantityAvailable
      WHERE stock_num = :stockNumber;
```

Figure 47. Embedded SQL for updating the database

Error handling

Our application must also have a mechanism for deciding whether embedded SQL statements execute correctly. When SQL statements are issued interactively, the interactive system generally provides an indication of whether the statement executed successfully. However, such a method does not work from within a program, so another method is needed for embedded SQL. Embedded SQL provides a standard mechanism for checking the success of SQL statements, and many RDBMS products provide additional mechanisms. In our application, we use the standard error–handling mechanism, the SQL communications area.

The *SQL communications area* (*SQLCA*) defines the **sqlca** data structure. This data structure has fields for error, warning, and status information. These fields are updated by the RDBMS after each SQL statement is executed. An application can then check these fields to determine whether the SQL statement was successful. The **sqlca** field of most interest to us in our application is the *sqlcode* field, which contains the return code of the most recently executed SQL statement.

The **sqlca** structure is included in a program by using an embedded SQL include statement. After each SQL call, our application checks the *sqlcode* field to determine if the call was successful. If the call fails, we abort the transaction. For the **UPDATE** statement, this is shown in Figure 48 on page 59.

```
EXEC SQL INCLUDE sqlca;
...
EXEC SQL UPDATE inventory
      SET num_available = :quantityAvailable
      WHERE stock_num = :stockNumber;
if (sqlca.sqlcode != SQL_SUCCESS){
      abort("Database update failed.");
}
```

Figure 48. Error handling in the example application

The complete PlaceOrder function

The complete **PlaceOrder** function is shown in Figure 49.

```
void PlaceOrder (idl_ulong_int stockNum,
                 idl_ulong_int numOrdered,
                 idl_ulong_int *costPerItem);
{
 EXEC SQL INCLUDE sqlca;
 EXEC SQL BEGIN DECLARE SECTION;
 unsigned long stockNumber;
 unsigned long pricePerItem;
 unsigned long quantityAvailable;
 EXEC SQL END DECLARE SECTION;

 stockNumber = stockNum;

 /* Determine the number in stock and the cost. */
 EXEC SQL SELECT num_available, item_cost
      INTO :quantityAvailable, :pricePerItem
      FROM inventory
      WHERE stock_num = :stockNumber;
 if (sqlca.sqlcode != SQL_SUCCESS){
      abort("Database lookup failed.");
 }
 *costPerItem = pricePerItem;
 if (quantityAvailable < numOrdered){
      abort("Insufficient stock.");
 }

 /*
 * Update the database. This update will be
 * backed out if it later turns out we do not
 * have sufficient funds in the billing database.
 */

 quantityAvailable -= numOrdered;

 EXEC SQL UPDATE inventory
      SET num_available = :quantityAvailable
      WHERE stock_num = :stockNumber;
 if (sqlca.sqlcode != SQL_SUCCESS){
      abort("Database update failed.");
 }

}
```

Figure 49. The PlaceOrder Function

Building and running the sample application

Our application now consists of one new source file—**PlaceOrder.precomp**. To build the application, we must first run the precompiler on this precompiler file. The command used depends upon the actual RDBMS that is used. Consult the documentation for the RDBMS you are using for specific information. The RDBMS documentation also specifies any database libraries with which the application must be linked.

Before the application is run, we must create the database table that is used by our application. The SQL command shown in Figure 50 creates the necessary table. As described previously, the data type name (*number* in the example shown) varies from RDBMS to RDBMS. The application may also need to include an RDBMS-specific header file.

```
create table inventory
   (stock_num number not null,
   num_available number not null,
   item_cost number not null)
```

Figure 50. Creating the RDBMS table needed for the sample application

Before the application can access the table, we must insert a few sample entries (rows) into the table. Figure 51 show an example of this. The values are inserted into the fields in the order specified when the table is created. For example, the first `insert` statement creates an item with Stock Number 1 and specifies that there are 100 of these available at a cost of 10 (dollars) each.

```
insert into inventory values (1, 100, 10)
insert into inventory values (2, 75, 5)
insert into inventory values (3, 200, 100)
```

Figure 51. Inserting sample rows into the inventory table

Chapter 7. Using Encina Peer-to-Peer Communications

The Encina Peer-to-Peer Communications (PPC) Services enable Encina applications to interact with applications running in Systems Network Architecture (SNA) networks, typically on mainframes. In this chapter, we expand our application to enable it to communicate with a mainframe using PPC Services.

Overview of PPC

Many Encina applications need to interact with applications on systems such as mainframes that are not running Encina. Encina Peer-to-Peer Communications (PPC) Services allow Encina applications to interact with applications running in SNA networks.

PPC Services consist of two components: the PPC Executive and the PPC gateway. The *PPC Executive* provides the application interface for Encina applications. The *PPC gateway* provides the bridge between an Encina/DCE system and an SNA network. The PPC gateway runs on a machine called the *PPC gateway* that is part of a DCE cell and an SNA network; it is transparent to application programs. This model is shown in Figure 52.

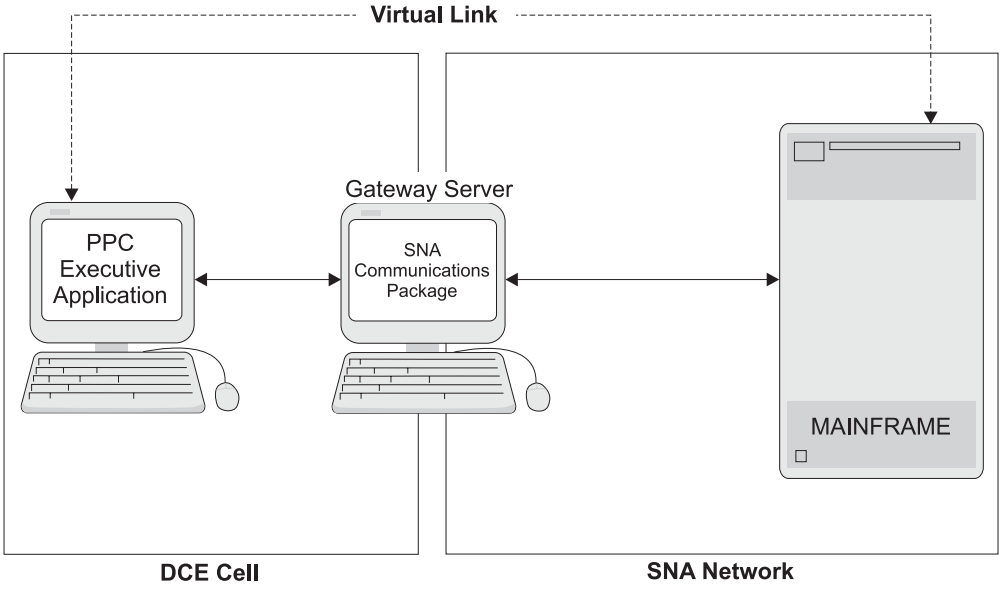

Figure 52. PPC communications model

PPC Executive applications are fully integrated into the Encina/DCE environment. That is, while using Encina and DCE to communicate with applications in the Encina/DCE environment, an application can communicate with a mainframe usng SNA.

PPC uses different terminology and a different programming interface from the rest of Encina. The remainder of this section describes SNA/PPC terminology and the PPC programming interface.

Logical units and transaction programs

A *logical unit* (*LU*) is an abstract representation that allows an application program to access an SNA network to communicate with another end user. The SNA networking specifications define a number of types of LUs. LUs of type 6.2 provide peer-to-peer communications between end users; the SNA protocol for peer-to-peer communications is called the *LU 6.2 protocol*. Multiple application programs can use the same LU.

The two-way communication between two application programs is called a *conversation*. The two applications are *partners* in the conversation and exchange information. The application programs are also referred to as *transaction programs*, and the logical name that an application uses to identify peers is called the *transaction program name* (*TPN*). Generally, the term TPN refers specifically to the function that is invoked by an incoming communication. A TPN is the PPC equivalent of a function in an exported interface under DCE/Encina. Thus, a PPC application can contain one or more TPNs.

To establish a conversation, one program *allocates* the conversation; that is, it specifies the LU and TPN with which it wants to communicate. The program that allocates the conversation is called the *allocator*. The program that is the recipient of an allocator's conversation request is called the *acceptor*. To end a conversation, one peer *deallocates* the conversation; its peer receives notification of the deallocation. Between allocation and deallocation, the application programs can exchange data and do work on each other's behalf. Figure 53 provides an example of a typical conversation.

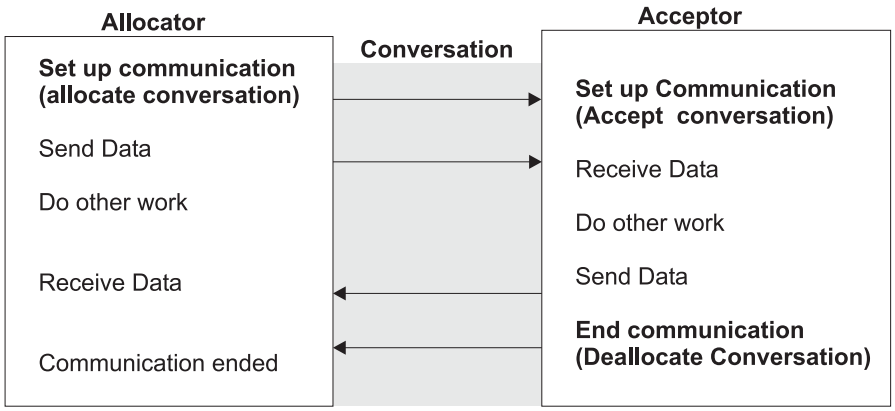

Figure 53. PPC conversations

To allocate a conversation to a peer, a program requires certain initialization information, such as the peer's TPN, the name of the partner LU, and so forth. This information is called *side information*. It is generally supplied and maintained by the system administrator in a *side information file*, which a program reads before allocating a conversation.

Peer-to-peer communications and client/server communications

The peer-to-peer communications model differs from the client/server model used by the rest of Encina. In the Encina client/server model, a client initiates a remote procedure call (RPC) to a server and waits for a response. The server receives and processes the RPC, and then returns to the client. The client and server are *not* peers. The server acts only on RPCs received from the client. A single path of execution weaves from the client, through the server function, and back to the client.

In contrast, in the peer-to-peer model of LU 6.2, an application allocates a conversation to another application, which starts processing the conversation concurrently. The partners establish a conversational context, sharing control of the conversation and exchanging data. The partners are true peers. Either side can send or receive data, ask the other side to do work, and so forth. There is still an originator of a conversation, akin to the client that originates an RPC; but once the conversation is established, there is no distinction between the roles of the two partners.

Synchronizationl level and logical units of work

Every PPC conversation has a synchronization level. The *synchronization level* (*synclevel*) specifies the degree to which the conversation is transactional. LU 6.2 supports three synchronization levels. The highest level, *synclevel syncpoint (SL2)*, provides transactional conversations, which use the two-phase commit protocol. Lower synchronization levels provide for simple confirmation that messages have been received (*SL1*) or for no confirmation at all (*SL0*). In our application, we will use synclevel syncpoint conversations.

The PPC term for a transaction is *logical unit of work* (*LUW*). Synclevel syncpoint conversations work on behalf of an LUW. As with other transactions, LUWs in PPC can be completed by being committed or aborted. To *commit* an LUW, either peer calls for a *syncpoint*. To *abort* an LUW, either peer can *backout* the LUW.

An application starts a transaction before allocating a synclevel syncpoint conversation. LUWs are then *chained*: when one is committed or aborted, another starts automatically. The conversation is thus a series of one or more LUWs.

Programming interfaces

PPC applications are written using the Common Programming Interface–Communications (CPI-C), as specified by IBM and X/Open. CPI-C provides a programming interface for PPC communications. The CPI-C interface provides a number of services, including the following:

- Allocating, accepting, and deallocating conversations
- Sending and receiving data
- Synchronizing processing between programs
- Notifying peers of errors

The IBM Common Programming Interface–Resource Recovery (CPI-RR) provides functions for committing and aborting transactions. These functions are also supported by PPC Services and can be used by applications involved in synclevel syncpoint (SL2) conversations. (Our application does not use the CPI-RR functions but instead uses Encina for starting and ending transactions.)

Designing the PPC application

When designing PPC applications, many design choices are often made based on an existing application. For example, an Encina application may be written based on an existing mainframe application with which it will communicate. Design choices such as which peer allocates the conversation and whether the conversation consists of one or a series of LUWs are based on what the existing application is already doing. These decisions, in turn, necessitate other decisions, such as what interface to use; for example, Tran-C cannot chain transactions, so our application cannot use Tran-C if the mainframe application expects the conversation to be a series of LUWs and not a single LUW.

In other cases, design is dictated by the overall architecture of the program. For our application, PPC is used to query a mainframe database. Thus, our application allocates the conversation. Figure 54 shows the overall design of the application; the shaded areas indicate the portion that uses PPC, which we will write in this chapter.

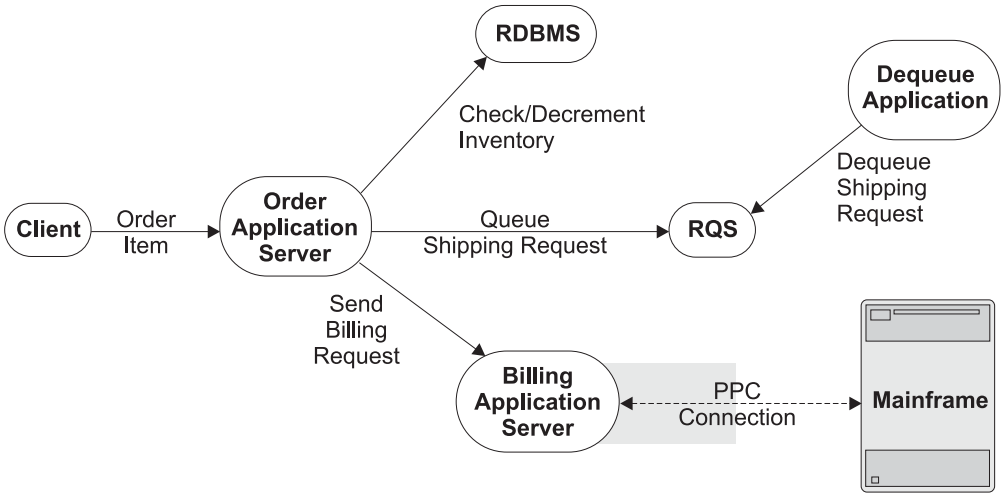

Figure 54. PPC portion of the sample application

Because our application is accessing and modifying permanent data, we want our conversation to be transactional; in the language of PPC, it must be a synclevel syncpoint (SL2) conversation. Furthermore, in our design, the conversation is short-lived. It consists of a single LUW. The billing server allocates a conversation to the mainframe, requesting an account balance. The mainframe application checks its account database, debits the account if sufficient funds exist, and returns an indication of success or failure to the Encina application. The billing server aborts the transaction if the billing operation fails. A failure in the underlying PPC mechanism can also cause the transaction to be aborted. Because PPC sets abort reasons, we can continue to display abort reasons in the order application server even though the abort reasons are not set by the application itself.

Remember that our PPC application is a Monitor application server, which we referred to as the billing server in previous chapters. The one function exported by this server, **BillForItem**, is invoked from within a transaction started by the order server. The order server initiates commit processing, not the billing server. The billing server needs only to abort in the case of error, not commit in the case of success.

The application did not need to be designed this way. The PPC order application server could have used PPC directly without an intermediate application server. We used the intermediate application server in our example because we wanted to demonstrate the use of a second application server, not because we were required to do so. Also, our application is a very simple PPC application. It does not deal with the special conditions necessary when LUWs are chained, nor does it have to handle the special processing necessary when the mainframe side initiates the transaction. We could, for example, have used long-lived conversations, saving the performance cost of allocating and deallocating a conversation for each order.

Writing the PPC application

In this section, we complete our billing application server, which uses a PPC conversation to interact with a billing program on a mainframe. Our application must perform the following four steps:

1. Initialize PPC
2. Allocate a conversation
3. Exchange data
4. Deallocate the conversation

We write a separate function to perform each of these steps. The initialization function, **InitializePpc**, is called as part of server initialization because we want to initialize only once, not at each RPC (see Figure 55). The other three functions are called by the **BillForItem** function, the remote procedure exported by our billing server. Figure 56 shows the **BillForItem** function. Note that the *convesationId* variable that is passed to the latter three functions is a pointer. It is returned by the **AllocateConversation** function and passed to the **DebitCustomerAccount** and **DeallocateConversation** functions.

```
int main()
{
...
status = mon_InitServer();
status = InitializePpc();
if (status != SUCCESS){
      fprintf(stderr, "Unable to initialize PPC.\n");
      exit(1);
}
status = mon_BeginService();
}
```

Figure 55. PPC initialization

```
void BillForItem(idl_ulong_int customerId,
                     idl_ulong_int amount)
{
 unsigned long status;
 CONVERSATION_ID conversationId;
 status = AllocateConversation(conversationId);
 if (status != SUCCESS)
     abort("Conversation Allocation Failed.");
 status = DebitCustomerAccount(conversationId, customerId, amount);
 if (status != SUCCESS)
      abort("Data Exchange Failed.");
 status = DeallocateConversation(conversationId);
 if (status != SUCCESS)
      abort("Conversation Deallocation Failed.");
}
```

Figure 56. Billing for the item

Each function returns a status code. In the case of the **InitializePpc** function, if the function fails, we terminate the server. If any of the other three functions fail, we abort the transaction using the Tran-C **abort** function. The transaction can also be aborted by the remote peer if it detects an error (for example, if the account does not contain sufficient funds).

Initializing PPC

Before an application can allocate or accept conversations, it must initialize PPC Services. If the application is using synclevel syncpoint (SL2) conversations, it must also initialize Encina. Because our application is a Monitor application server, Encina initialization is performed by the Monitor, so we have to initialize only the PPC Services.

The **cpic_Init** function initializes PPC Services. This function takes one argument, the application's complete LU name. This complete LU name (for example, `MAINFRAM.REMOTLU1`) is furnished by the system administrator and must match the one used when PPC was configured. To simplify our application, the application header file uses the `LU_NAME` constant to define the LU name.

A PPC Executive application that is going to allocate a conversation must first determine the identity of its conversation partner. The partner's identity is stored in a side information file; this information is read into the program using the **cpic_ReadSideInfo** function. Again, to simplify our application, the application header file uses the `SIDE_INFO_FILE_NAME` constant to define the name of the side information file. Figure 57 shows a sample entry in a side information file.

```
sideInfo {
      "CHCKACCT", /* Symbolic Destination Name */
      "", /* Use default mode */
      "BillingApp", /* Partner name (see below) */
      "QUERY", /* TPN */
      ENCRYPT_NONE, /* No encryption */
      SECURITY_NONE, /* No login security */
      "", "" /* User ID and password */
}

partner {
      "BillingApp", /* Partner name */
      CONNECTION_GWY_TCP, /* Use PPC Gateway */
      "REMOTLU1" /* LU */
}
```

Figure 57. Sample side information file entry

When our application initializes a conversation, it specifies the symbolic destination name. This provides a key into the side information file, which in turn provides more information about the conversation. For example, it specifies that the partner name is **BillingApp** (which in turn keys into the partner section of the table, which provides the LU to use—in this case **REMOTLU1**), that the conversation connects to the **QUERY** TPN, and so forth. (For more information on all the entries in the side information file, see the *Encina Administration Guide Volume 2: Server Administration*.)

Figure 58 on page 67 shows the **InitializePpc** function.

```
error_status_t InitializePpc(void)
{
 char *luName = LU_NAME;
 char *sideInfoFile = SIDE_INFO_FILE_NAME;
 CM_RETCODE returnCode;

 returnCode = cpic_Init(luName);
 if (returnCode)
      return INIT_FAILED;
 returnCode = cpic_ReadSideInfo(sideInfoFile);
 if (returnCode)
      return INIT_FAILED;

 return SUCCESS;
}
```

Figure 58. PPC initialization

Allocating conversations

Allocating a conversation includes the following steps:

1. Initializing the conversation
2. Setting the synclevel
3. Actually allocating the conversation

To initialize a conversation, our application calls the CPI-C **Intitialize_Conversation** function. This function takes one input argument, the symbolic destination name of the program to which the conversation is to be allocated. This name provides a key into the side information file read in as part of PPC initialization, which provides other information about the conversation, such as the partner name and the TPN. In our program, we use the symbolic destination name `CHCKACCT`.

After initializing the conversation but before actually allocating it, our application must specify that we are using a transactional (synclevel syncpoint) conversation. To do so, the application calls the CPI-C **Set_Sync_Level** function. For a synclevel syncpoint conversation, a transaction must be started before allocating the conversation. In our application, the transaction is started by the order server before it invokes the **BillForItem** remote procedure that is implemented by the billing server.

The conversation is allocated using the CPI-C **Allocate** function. A return status code of `CM_OK` indicates that the conversation has been successfully allocated. The application is in *send* state and can begin sending data.

The **AllocateConversation** function is shown in Figure 59 on page 68.

```
error_status_t AllocateConversation(CONVERSATION_ID conversationId)
{
 CM_RETCODE returnCode;
 char *symDestName = "CHCKACCT";
 char *syncLevel = CM_SYNC_POINT;

 /* Initialize the conversation and set the synclevel */
 Initialize_Conversation(conversationId, symDestName,
                                                &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;
 Set_Sync_Level(conversationId, syncLevel, &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;
 Allocate(conversationId, &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;

 return SUCCESS;
}
```

Figure 59. Allocating a conversation

Exchanging data

Applications exchange data by using the CPI-C functions that send and receive data. Our application only needs to send data. In particular, it must send the customer ID and the amount of the purchase to the mainframe application. The mainframe application uses this information to query a database and aborts the transaction if it encounters a failure (for example, if there is not enough money in the account to cover the amount of the purchase). To send data, our application uses the CPI-C **Send_Data** function. Note that we need to specify whether the data is buffered or sent immediately. Our application uses the **Set_Send_Type** function to specify that data is sent immediately.

Because the data supplied to the **Send_Data** function must be a single buffer, our application must first pack the two fields that it sends into one buffer. To do this, it uses the standard C **sprintf** function. We could also have used the C **memcpy** function, as we did in the RQS portion of the application in Chapter 5, "Using RQS," on page 43. Note that we have chosen a very simple data format for our example. Actual mainframe applications typically follow more formal rules and use more complex formats.

If the **Send_Data** function returns the status code `CM_OK`, the data was successfully sent to the mainframe. However, this does not mean that the mainframe successfully debited the customer account; our application is only assured of this when the transaction commits. If the mainframe application cannot debit the account, due to insufficient funds or to some other problem, it aborts the transaction. Our application is then notified of this in the **onAbort** clause in the order server (where the transaction was started), not via a status code in the billing server.

The **DebitCustomerAccount** function is shown in Figure 60 on page 69.

```
error_status_t DebitCustomerAccount(CONVERSATION_ID convId,
                                    idl_ulong_int customerId,
                                    idl_ulong_int amount)
{
 char debitInfo[MAX_SEND_LENGTH];
 int debitInfoLen = 0;
 REQUEST_TO_SEND_RECEIVED requestToSendReceived;
 CM_RETCODE status;

 /* The data is sent immediately rather than buffered */
 SEND_TYPE sendType = CM_SEND_AND_FLUSH;

 /* Pack the buffer to send to the peer */
 sprintf(debitInfo, "%lu %lu", customerId, amount);
 debitInfoLen = strlen(debitInfo);

 Set_Send_Type(convId, &sendType, &status);
 if (returnCode != CM_OK)
      return CPIC_ERROR;

 Send_Data(convId, debitInfo, &debitInfoLen,
                &requestToSendReceived, &status);
 if (returnCode != CM_OK)
      return SEND_ERROR;

 return SUCCESS;
}
```

Figure 60. Sending and receiving data

Deallocating conversations

To deallocate a conversation, one peer initiates deallocation. The other peer receives notification of the deallocation when it calls the **Receive** function.

To deallocate a conversation, an application performs the following steps:

1. Sets how it wants to deallocate the conversation (for example, normally or abnormally) using the **Set_Deallocate_Type** function or using the default deallocation type of CM_DEALLOCATE_SYNC_LEVEL.
2. Deallocates the conversation using the **Deallocate** function.
3. For synclevel syncpoint conversations, completes the transaction.

For our application, the billing server initiates deallocation. It performs the first two steps. The final step—completing the transaction—is performed by the order server after the RPC to the billing server returns.

If the conversation is a synclevel syncpoint conversation, the application must first deallocate the synclevel conversation and then commit the transaction. These steps may appear to be in the wrong order. However, for synclevel syncpoint conversations, the **Deallocate** function does not actually deallocate the conversation. It simply marks the conversation to be deallocated when the transaction is complete. In our case, the transaction is not committed by the billing server but by the order server after the RPC to the billing server returns. The conversation is deallocated at that point.

Figure 61 on page 70 shows the **DeallocateConversation** function. Note that the application calls the **Set_Deallocate_Type** function to set the deallocation type to

CM_DEALLOCATE_SYNC_LEVEL. We could have eliminated this call in our application because CM_DEALLOCATE_SYNC_LEVEL is the default.

```
error_status_t DeallocateConversation(CONVERSATION_ID conversationId)
{
 CM_RETCODE returnCode;

 Set_Deallocate_Type(conversationId, CM_DEALLOCATE_SYNC_LEVEL
                &returnCode);
 if (returnCode != CM_OK)
      return DEALLOCATION_FAILURE;

 Deallocate(conversationId, &returnCode);
 if (returnCode != CM_OK)
      return DEALLOCATION_FAILURE;

 return SUCCESS;

}
```

Figure 61. Deallocating a conversation

If an abort occurs, Encina deallocates the conversation abnormally. This is the method used by our example application. However, in general, either peer in a conversation can deallocate abnormally by setting the deallocation type to CM_DEALLOCATE_ABEND, then calling the **Deallocate** function; this method unconditionally deallocates the conversation, regardless of synchronization level or conversation state.

The PPC application: the mainframe side

Developing the mainframe side of the PPC application is beyond the scope of this document. The mainframe application can be written a number of ways, using various languages or packages (such as CICS). Figure 62 on page 71 shows the basic outline of the mainframe application.

Note that the mainframe application initiates commit processing by calling the **Commit** function. The transaction is not committed, however, until the Encina PPC application also completes the transaction, which occurs when the RPC to the billing server returns to the order server and the Tran-C transaction block is completed.

Also note that because PPC automatically chains transaction, the mainframe has a local transaction of its own to end after the conversation has been deallocated. We did not need to perform this step on the Encina side because Tran-C automatically handles this for us. For more information, see the *Encina PPC Services Programming Guide*.

```
HandleCustomerDebit(tpn, ...)
{
 ...

 /* Accept the conversation. */
 ACCEPT_CONVERSATION(tpn, convId, &status);

 CheckSuccess(status);

 /* Receive data from allocator */
 Receive(convId, receiveBuffer, ...);
 if ( ...) {
      /* If successful, actually debit the account. */
      DEBIT_CUSTOMER_ACCOUNT(customerId, amount);
 }


 /* If not successful, set the deallocate type to abend
 (abnormal deallocation), deallocate, then call
 Backout. Otherwise, deallocate, then Commit.
 */
 if (debitStatus != SUCCESS) {
      deallocType = CM_DEALLOCATE_ABEND;
      Set_Deallocate_Type(convId, &deallocType, &status);
 }

 Deallocate(convId, &status);

 /* Conversation now marked for deallocation after
 * commit or backout. At this point, commit or back out,
 * based on success of debit operation.
 */
 if (debitStatus == SUCCESS)
      Commit(&status);
 else
      Backout(&status);

 /* End any local (non-distributed) transaction */

 ...

}
```

Figure 62. Outline of the mainframe side of the PPC application

Notes on building and running the application

Our application now consists of one additional source file—**ppcConstants.h**. This header file contains PPC constants such as the name of the side information file. It is included by the Billing Server.

In addition to the include files listed in the previous chapters, the billing application server must now include the **ppc/cpic.h** file. The billing application server must link to the **EncPpcExec** library in addition to the libraries listed in the previous chapters.

The billing server is started like any other Monitor application server (see Chapter 3, “Writing a Monitor client/server application,” on page 23). For further details on administering and running PPC applications, see the *Encina Administration Guide Volume 2: Server Administration*.

Chapter 8. Using TX

In previous chapters, we used Tran-C to start and end transactions. In this chapter, we discuss some of the situations in which you may want to use TX instead of Tran-C. In this chapter, we also describe how to use TX with our example application.

Introduction to X/Open TX

X/Open defines an application program interface for starting and ending transactions, directing the completion of transactions, and obtaining status information about transactions. This interface is called the **TX interface**. Encina supports the TX interface and provides several extensions to it.

TX can be used to manage transactions in clients and servers. In this chapter, we use it in a Monitor application server.

TX transactions

Unlike Tran-C, TX uses explicit functions to begin and end transactions.

- The **tx_begin** function starts a transaction.
- The **tx_commit** function tries to commit a transaction.
- The **tx_rollback** function aborts a transaction.

The initiator of a transaction is the only process that can initiate commit processing; no other process can call the **tx_commit** function. Another process can indicate that it would like to abort the transaction (for example, by calling the **tx_rollback** function or the Tran-C **abort** function). However, this only marks the transaction for subsequent abort. The transaction is not actually resolved until the initiator calls either the **tx_commit** function or the **tx_rollback** function. Remember that, under the rules for the two-phase commit process, all processes must agree to commit a transaction; if any process in the distributed transaction wants to abort the transaction, it is aborted. Thus, if another process has marked the transaction to be aborted and the initiator calls the **tx_commit** function, the transaction is aborted; the **tx_commit** function returns a status code informing the initiator that the transaction aborted.

In TX, the initiator of the transaction is not informed immediately if another process requests that the transaction be aborted. There is no TX equivalent of the Tran-C **onAbort** clause; that is, there is no automatic transfer of control when another process initiates an abort. The initiator finds out about the requested abort when it tries to commit the transaction. However, in some cases, this can mean that the process first does a great deal of work that the transaction manager then has to undo. To avoid this, the initiator can call the **tx_info** function, which returns information about a transaction. If this function indicates that an abort has been requested, the process can call the **tx_rollback** function immediately.

TX and Tran-C

A single process must use either Tran-C or TX. A transaction begun with a TX function must end with a TX function and must not contain any Tran-C constructs (such as **transaction** or **abort**). Likewise, a process using Tran-C must not call any TX functions. Tran-C and TX can, however, be used in a single transaction when

more than one process is involved. That is, a Tran-C client can make a TRPC to a TX server, or a TX client can make a TRPC to a Tran-C server.

Encina extensions to TX enable an application to use either of the abort mechanisms described in Chapter 4, "Making the sample application transactional," on page 37. Prior to calling the **tx_rollback** function, the application can set an abort string by calling the **tx_set_rollback_string** function or set an abort code by calling the **tx_set_rollback_code** function. It can retrieve abort strings, including strings passed to the Tran-C **abort** function, by using the **tx_get_rollback_string** function. Similarly, it can retrieve abort codes, including those passed to the Tran-C **abortWithCode** function, by using the **tx_get_rollback_code** function.

Unlike Tran-C transactions, TX transactions can be chained. When transactions are **chained**, a new transaction is automatically started when a transaction commits or aborts. This may be necessary, for example, if the process is using PPC to communicate with a mainframe application that is expecting chained transactions.

Tran-C also provides some advanced features not provided by TX. For example, Tran-C provides simplified threading capabilities (see the *Encina Transactional Programming Guide*).

When to use TX

As noted earlier, TX is an alternative to Tran-C. The following cases can dictate the use of TX:

- The application is being ported from another system and already uses TX. In this case, it may be simpler to continue to use TX under Encina than to modify the application to use Tran-C.
- The application is using C++ rather than C. Conflicts between the C++ exception mechanism and the Tran-C exception mechanism can make it difficult to use Tran-C constructs in C++ applications.
- The transaction must span a lexical scope. For example, the transaction may have to be started in one function and committed in another. This can happen in event-driven programming.
- The application is written in some high-level language other than C (such as COBOL).
- The application is written using a tool that automatically uses TX. This is true of some PC-based tools.
- The application is using PPC and must chain transactions to communicate properly with the mainframe application. This is described briefly in Chapter 7, "Using Encina Peer-to-Peer Communications," on page 61.

Using TX in the order application server

In this section, we modify one of our application servers (the order server) to use TX instead of Tran-C to manage transactions. To use TX, we must do the following three things in the order server:

- Modify the application to initialize the TX interface.
- Remove the Tran-C constructs and instead use TX to delimit the transaction.
- Modify the termination steps to close the TX interface as part of application shutdown.

We do not need to change the client or the billing server. Moreover, the billing server continues to use Tran-C and, in particular, still calls the Tran-C **abort** function if an error occurs.

Initializing the TX interface

Normally, the TX interface is initialized simply by calling the **tx_open** function, which also opens communications with RDBMSs that the application is using. However, in our case, we are using TX in a Monitor application server. By default, the Monitor automatically initializes Tran-C. We must direct the Monitor not to initialize Tran-C. To do so, as part of initialization, our application calls the **mon_ServerUsesTx** function before calling the **mon_InitServer** function. The Monitor then calls the **tx_open** function to complete initialization; our server does not need to explicitly call the **tx_open** function. If we were starting transactions in the client, we would have had to call the **tx_open** function.

The new initialization steps are shown in Figure 63.

```
mon_ServerUsesTx();
status = mon_InitServer();
```

Figure 63. TX initialization in a Monitor application server

Starting and ending a transaction using TX

Using TX, transactions are started by calling the **tx_begin** function and ended by calling either the **tx_commit** function or the **tx_rollback** function. Between these statements, the application does the work associated with the transaction, which for our application is the same work it performed in previous chapters.

All TX functions return a status code, which we must check. This is especially important in the case of the **tx_commit** function. As noted earlier, TX does not have the equivalent of the Tran-C **onAbort** clause. Instead, our application is notified that another process wants to abort the transaction by a return code from a TX function. For example, if the billing server aborts the transaction, our server is notified of the abort by a non-zero return code from the **tx_commit** function. (If the function is successful, it returns a status code of zero.)

The code is shown in Figure 64 on page 76.

```
int txRetCode;
...
txRetCode = tx_begin();
if (txRetCode) {
      /* Print error message and return. */
      fprintf (stderr,"Transaction failed.\n");

}
PlaceOrder(stockNum, numOrdered, &costPerItem);
totalCost = numOrdered * costPerItem;
PlaceItemOnQueue(stockNum, numOrdered, customerId);
BillForItem(customerId, totalCost);

txRetCode = tx_commit();
if (txRetCode) { /* Commit failed.*/
      fprintf (stderr,"Transaction failed: %s\n",
      tx_get_rollback_string();
      return ORDER_FAILED;
}
else
      return SUCCESS;
```

Figure 64. Using TX to start and end a transaction

Note in the example code that if the transaction aborts, our application is notified only when it calls the **tx_commit** function. At this point, all the work done as part of the transaction is rolled back. This is not a problem in our application because the transaction is short and not much work is done during any of the steps. However, in applications that do more work (and thus would have much more work to roll back), waiting until the end of the transaction to roll back the work can be a problem. We can solve this problem by either of the following methods:

- Modifying the three functions that are part of the transaction to return status codes, rather than abort the transaction themselves, when they encounter an error. The server can then check these status codes and call the **tx_rollback** function to abort the transaction if necessary.
- Calling the **tx_info** function after each function call. The **tx_info** function returns information about the current transaction. If another process has marked the transaction to be aborted, the **tx_info** function returns this information. The server can then immediately call the **tx_rollback** function to abort the transaction if necessary.

An example of the second approach is shown in Figure 65 on page 77. We pass a structure of type **TXINFO** to the **tx_info** function. The *transaction_state* field of this structure contains the state of the transaction. If the transaction is still active (and thus has not been marked for abort), the value of this field is `TX_ACTIVE`.

```
int txRetCode;
TXINFO txInfo;

txRetCode = tx_begin();
if (txRetCode) {
      /* Print error message and return */
      fprintf (stderr,"Transaction failed.\n");
}
 PlaceOrder(stockNum, numOrdered, &costPerItem);
 txRetCode = tx_info(&txInfo);
 if (txInfo.transaction_state != TX_ACTIVE) {
      tx_rollback();
      fprintf (stderr,"Transaction failed: %s\n",
      tx_get_rollback_string());
      return ORDER_FAILED;
 }

 /* Place similar calls to tx_info after other functions */
 ...
txRetCode = tx_commit();
if (txRetCode) {
      fprintf (stderr,"Transaction failed: %s\n",
      tx_get_rollback_string());
      return ORDER_FAILED
}
```

Figure 65. An alternate method of detecting aborts using TX

Closing the TX interface

To close the TX interface, our server calls the **tx_close** function. The server calls this function as part of server termination, after the **mon_BeginService** function returns. This is shown in Figure 66.

```
mon_BeginService();
tx_close();
```

Figure 66. Closing the TX interface

Notes on building the application

To use TX, the application must include the **tx/tx.h** header file. No additional libraries are needed: TX functions are included in the Encina Toolkit libraries that we have been linking with since Chapter 3, “Writing a Monitor client/server application,” on page 23.

Chapter 9. Using nested transactions

In this chapter, we introduce the concept of nested transactions as a way to achieve error isolation. After introducing the concepts, we add a nested transaction to our example.

Introduction to nested transactions

Using transactions as we have to this point does not always allow applications the granularity of error isolation that may be desired. If the transaction aborts, all changes are rolled back. As our example application is now written, this is the desired behavior. However, for more complicated transactions, we may want a finer granularity in error isolation. For example, we may not want to undo all parts of a transaction due to an error in one operation.

As an example, consider the billing part of our application. Currently, the billing algorithm is the following:

1. The order server makes an RPC to the billing server.
2. The billing server, using PPC, queries the billing database on the mainframe and decrements the account balance.
3. If the customer has insufficient funds in the billing database, the transaction is aborted. Any other PPC failures also result in the transaction being aborted.

The abort in Step 3 results in all parts of the transaction being aborted. Not only is the change to the billing database backed out; changes to the inventory database are backed out, and the shipping request is dequeued. This algorithm is used for orders from all customers.

However, suppose we want to extend credit to preferred customers. These customers are listed in preferred customer database, which also records the current credit and maximum credit for preferred customers. The preferred customer database is local; thus our application does not have to access the mainframe for preferred customers. To use this database, we change the billing algorithm as follows:

1. The order server first checks the preferred customer database (which could be another RDBMS or an SFS file).
2. If the customer has an entry in that database, we increment the "current credit" amount by the amount of the order.
3. If the current order places that customer over the credit limit, we abort the transaction to back out any changes we made to the database.
4. Only if the customer is not a preferred customer or does not have sufficient credit do we make an RPC to the billing server.

In our current transactional model, the abort in Step 3 in the new billing algorithm backs out not only any changes to the preferred customer database but all work done by the transaction. We can of course change the algorithm so that the application does not abort in the case of insufficient funds but instead queries the database and then decrements it only if sufficient funds exist. However, as we will discuss in "Changing the design of the application server" on page 81, there are reasons for not doing so.

We need a way to isolate any errors that occur in the interaction with the local database, preventing such errors from aborting the entire transaction. The solution is to check and decrement the local database from within a nested transaction. A *nested transaction* is a new transaction begun from within the scope of another transaction.

Nested transactions offer several features, including:

- Nested transactions enable an application to isolate errors in certain operations.
- Nested transactions allow an application to treat several related operations as a single atomic operation.
- Nested transactions can operate concurrently.

Nested transactions, like any other transactions, do incur a performance cost. Therefore, they should be used only when necessary.

Nested and top-level transactions

As described in the previous section, a nested transaction is begun within the scope of another transaction. The transaction that starts the nested transaction is called the *parent* of the nested transaction. There are two types of nested transactions:

- A *nested top-level transaction* commits or aborts independently of the enclosing transaction. That is, after it is created, it is completely independent of the transaction that created it. The Tran-C **topLevel** construct for creating nested top-level transactions. The syntax of this construct is identical to that of the **transaction** construct, but the **topLevel** keyword is used instead of the **transaction** keyword.
- A *nested subtransaction* commits with respect to the parent transaction. That is, even though the subtransaction commits, the permanence of its effects depends on the parent transaction committing. If the parent transaction aborts, the results of the nested transaction are backed out. However, if the nested transaction aborts, the parent transaction is not aborted. The easiest way to create a nested subtransaction transaction in Tran-C is to simply use a **transaction** block within the scope of an existing transaction. Tran-C automatically makes the new transaction a subtransaction of the existing transaction.

In this chapter, when we discuss nested transactions, we are generally referring to nested subtransactions unless we specify otherwise.

A series of nested subtransactions is viewed as a hierarchy of transactions. When transactions are nested to an arbitrary depth, the transaction that is the parent of the entire tree (family) of transactions is referred to as the *top-level transaction*. If the top-level transaction aborts, all nested transactions are aborted as well.

By default, nested subtransactions of the same parent transaction are executed sequentially within the scope of the parent. The Tran-C **concurrent** and **cofor** statements can be used to create subtransactions that execute concurrently with each other on behalf of their parent transaction. For more information, see the *Encina Transactional Programming Guide*.

Using nested transactions in the example application

In this section, we will add a nested transaction to the example that we developed in preceding chapters.

Changing the design of the application server

We are changing our application to perform one additional action: prior to checking a mainframe database for billing information, the application first checks a local preferred customer database. Only if this lookup fails do we check the mainframe database. The function that checks the local database aborts the transaction if the customer is not a preferred customer (not in the database) or if it encounters an error. To prevent this abort from aborting the whole transaction, we isolate it using a nested transaction.

We could implement this extra lookup without using nested transactions. For example, we could have the new function return a status code and have the application query the mainframe only if the function returns an error. However, there are several reasons for using a nested transaction:

- Using a nested transaction isolates any errors that occur in the underlying RPC mechanism. Such errors could abort the transaction. A nested transaction isolates these aborts in the same way that it isolates aborts explicitly generated by the application.
- Using a nested transaction allows the function to abort (and thus back out) its own actions. If it did not use a nested transaction, the function would first have to query the database, then update it if the query indicated sufficient credit. If each update required an RPC, this design would result in two RPC for each successful call. By contrast, if the function instead aborts the transaction, it needs only one RPC. Although this results in an abort (and hence more work for the application) in the failure case, it is more efficient in the normal case.
- None of the other functions called as part of the transaction returns a status code. All abort if they encounter an error. Thus, using a nested transaction gives the new function an interface similar to other functions in the transaction.

Note that in this particular case, we could not reverse the order of the lookups (that is, first check the mainframe database and then check the local database). The LU 6.2 specification (and hence PPC) does not support nested transactions. Thus, if the mainframe transaction were to abort, the top-level transaction (not simply the nested transaction) would be aborted.

Creating the nested transaction

To create a nested transaction in our application, we simply use a second **transaction** construct within the scope of the first. Tran-C creates a nested subtransaction when it encounters the new **transaction** construct. When execution reaches the closing brace of the nested subtransaction, Tran-C initiates commit processing and commits the subtransaction with respect to the parent transaction. That is, changes made in this subtransaction are backed out if the parent transaction aborts. Figure 67 on page 82 shows the top-level transaction and the nested subtransaction.

```
error_status_t OrderItem(idl_ulong_int stockNum,
             idl_ulong_int numOrdered,
             idl_ulong_int customerId)
{

 idl_long_int returnStatus;
 idl_long_int costPerItem, totalCost;
 short priority;
 volatile short preferredCustomer = TRUE;

 transaction{
      PlaceOrder(stockNum, numOrdered, &costPerItem);
      totalCost = numOrdered * costPerItem;
      if (totalCost > 1000)
           priority = HIGH_PRIORITY;
      else
           priority = NORMAL_PRIORITY;

      PlaceItemOnQueue(stockNum, numOrdered,
                                       customerId, priority);
      transaction{/* Begin a nested transaction. */
           BillPreferredCustomer(customerId, totalCost);
      } onAbort {
           /* If nested transaction aborts. */
           preferredCustomer = FALSE;
      }

      /* If nested transaction aborted, check mainframe database */
      if (!preferredCustomer)
           BillForItem(customerId, totalCost);
 }onCommit{
           fprintf(stderr, "We committed.\n");
           return SUCCESS;
 }onAbort{
           fprintf(stderr, "We aborted. %s\n", abortReason());
           return ORDER_FAILED;
 }
}
```

Figure 67. Using a nested transaction in the sample application

The **BillPreferredCustomer** function is part of the nested transaction. If this function is successful, the nested transaction commits. However, the results can still be backed out if the parent transaction is aborted. If the **BillPreferredCustomer** function aborts, only the nested transaction is aborted, not the parent transaction. Control transfers to the **onAbort** clause of the nested transaction, where the *preferredCustomer* flag is set to FALSE.

The **BillForItem** function, which queries the mainframe database, must be part of the top-level transaction and must be invoked only if the nested transaction aborts. Therefore, the BillForItem function is invoked if the value of the *preferredCustomer* flag is FALSE.

Appendix A. Building Encina applications

This appendix provides information needed in compiling and building Encina applications. It lists the C header files, COBOL COPY-files, and libraries for all parts of Encina. It also lists platform-specific libraries.

For information about C++ libraries, see *Encina Object-Oriented Programming Guide*.

Encina include files and libraries for C programs

Table 4 specifies the header files and libraries that must be used when compiling and linking Encina applications written in C. For information on C++ and Java libraries, see *Encina Object-Oriented Programming Guide*

The order of the list of libraries for each component is significant. In addition, user libraries should be specified before the libraries listed here.

Note that on UNIX platforms, the library names shown are the names that are used with the *-l* compiler option (for example, *-l***EncRqs**); the actual name of each library file is prefaced by **lib** and has a suffix of **.a** (for example, **libEncRqs.a**). On Windows platforms, the actual name of the library file that is linked with the application is prefixed by **lib** and has a prefix of **.lib** (for example, **libEncRqs.lib**).

Table 4. Encina include files and libraries for C programs

Encina component	Header files	Libraries
Toolkit Executive		
Transaction Service	**tran/tran.h**	**Encina**
Thread-to-Tid Mapping Service	**threadTid/threadTid.h**	**Encina**
TRPC	**trpc/trpc.h**	**Encina**
Encina Abort Facility	**encina/afac.h**	**Encina**
DCE Utilities Library	**trdce/trdce.h**	**Encina**
TX Interface	**tx/tx.h**	**Encina**
TX Interface with Encina Extensions	**tx/tx_extensions.h**	**Encina**
Transactional-C	**tc/tc.h**, **tc/rpc/tc_trpc.h**	**Encina**
Toolkit Server Core		
Transactional-C Server Extensions	**tc/tc_server.h**, **tc/rpc/tc_trpc.h**	**EncServer**, **Encina**
Lock Service	**lock/lock.h**	**EncServer**, **Encina**
Log Service	**log/log.h**	**EncServer**, **Encina**
Recovery Service	**rec/rec.h**	**EncServer**, **Encina**
TM-XA Service	**tmxa/tmxa.h**	**EncServer**, **Encina**
TranLog - Client	**tranLog/tranLog.h**	**EncClient**, **Encina**
TranLog - Server	**tranLog/tranLog.h**	**EncServer**, **Encina**
Volume Service	**vol/vol.h**	**EncServer**, **Encina**

Table 4. Encina include files and libraries for C programs (continued)

Encina component	Header files	Libraries
Restart Service	**restart/restart.h**	**EncServer**, **Encina**
Structured File Server (SFS)		
Base Functionality	**sfs/sfs.h**	**EncSfs**, **Encina**
ISAM Interface	**tisam/tisam.h**	**EncSfs**, **Encina**
Recoverable Queueing Service (RQS)		
Base Functionality	**rqs/rqs.h**	**EncRqs**, **Encina**
Peer-to-Peer Communications (PPC) Services		
X/Open CPI-C	**ppc/cpic.h**	**EncPpcExec**, **Encina**
IBM SAA CPI-C and CPI-RR	**ppc/cmc.h**, **ppc/srrc.h**	**EncPpcExec**, **Encina**
Encina Monitor		
Client	**tpm/mon_client.h**	**EncMonCli**, **Encina**
Jam Client	**tpm/mon_client.h**, **tpm/mon_jam.h**	**EncMonCliJam**, **EncMonCli**, **Encina**
Administrative Client	**tpm/ema.h**	**EncMonCli**, **Encina**
Application Server	**tpm/mon_server.h**	**EncMonServ**, **EncServer**, **EncClient**, **Encina**

Encina COPY-files and libraries for COBOL applications

Table 5 shows the COPY-files and libraries that should be used when compiling and linking Encina applications written in COBOL. The COBOL products are not available in the Encina 2.5 release on Windows.

Table 5. Encina COPY-Files and libraries for COBOL programs

Encina component	COPY-files (used as needed)	Libraries
Monitor Client		**EncMonCliCobol**, **EncMonCli**, **Encina**
Monitor Application Server	**ADL-AUTHN.cbl**, **ADL-AUTHZ.cbl**, **ENCINA-OPAQUE.cbl**, **MON-ACCESS.cbl**, **MON-LOCK.cbl**, **MON-PROTS.cpy**, **MON-RESERV.cbl**, **MON-SCHED.cbl**, **MON-STATUS.cbl**, **MON-SUSP.cbl** or **MON-SUSPEND.cbl**, **MON-TIMER.cbl**, **MON-TRAN.cbl**, **OPAQUE.cbl**, **SFS-CONFIG-PROTS.cpy**, **TX-PROTS.cpy**	**EncMonServCobol**, **EncMonServ**, **EncServer**, **Encina**
TX Interface with Encina Extensions	**TXINFDEF.cbl**, **TXSTATUS.cbl**	**Encina**
SFS External File Handler (EXTFH)	None	**EncSfsExtfh**, **EncSfs**, **Encina**

Platform-specific libraries

Additional libraries must be linked with the application code. The additional libraries depend on the operating system and machine type. On most platforms, the DCE library (**dce**) must be explicitly specified. For various platforms, the libraries needed are shown in Table 6. For those platforms on which command-line compilers are typically used, the actual command-line arguments for the compilers are shown in the table.

Table 6. Platform-specific libraries

Platform	Additional libraries to include
Solaris 2.x	*-ldce -lm -lnsl -lthread -lsocket*
HP-UX	*-ldce -lndbm -lM -ldld -lc_r*
AIX	*-ldce*
Windows (Microsoft Visual C++ and IBM DCE)	**msvcrt**, **pthreads**, **libdce**

Note: Information on system libraries may change with time. See the release notes for the platform you are using for the most current information on system libraries for that platform. Also refer to the make file installed with the sample Encina applications.

Other platform-specific compiler and linker options

Table 7 lists other platform-specific compiler and linker options for UNIX platforms. Table 8 lists compiler and linker flags required on Windows platforms.

This information can change with time. For the most up to date list, see the Makefile for the sample Encina applications installed as part of Encina.

Table 7. Platform-specific compiler and linker options for UNIX

Platform	Additional compiler options
Solaris 2.x	*-D_REENTRANT*
HP-UX	*-D_REENTRANT -Aa -D_HPUX_SOURCE -Dhpux -Dhp9000s800*
AIX	*-Dunix -D_BSD -D_ALL_SOURCE*

Note: On AIX 4.3 or later, when building executables that use DCE libararies and other third-party libraries that use **.so** libraries, specify the *-brtl* and *-bnortllib* options, in addition to those shown in Table 7.

Table 8. Platform-specific compiler and linker options for Windows

Flags	Description
-MD	Uses the dynamically-linked C run-time library.
-DWIN32	Defines the symbol WIN32. **Note:** If you define the symbol elsewhere before you include Encina header files, you can omit this flag.

Table 8. Platform-specific compiler and linker options for Windows (continued)

Flags	Description
-DIBM-DCE	Enables Encina header files to be used with this version of DCE.
-Gz	Sets the default stack discipline to stdcall.

Note: Most Encina header files declare the calling convention for all functions. However, on Windows platforms you must specify either the *-Gz* or the *-Gd* flag for some DCE and Encina functions, including IDL- and TIDL-generated code. These requirements will be removed in a future Encina release.

Appendix B. Using abort codes

This appendix describes the Encina Abort Code Facility.

Overview of aborting with abort codes

In the example application developed in the body of this manual, a character string is specified when a function aborts. This character string can be retrieved and printed by the function that receives notification of the abort. Encina applications can also use abort codes rather than explicit character strings. Abort codes offers the following advantages over strings:

- As integers, they are easier to compare than strings. Thus, an application can respond to a code more easily than to a string.
- They can be translated into strings using external catalogs of messages. Thus, programs that use codes can be internationalized more easily than programs that use strings.

However, the abort code interface is somewhat more complex than the character-string interface. Thus, aborting with strings can be a better approach for those applications that are used in only one language and that respond to abort notifications by simply printing a string and exiting.

The basic steps in using the abort code mechanism follow:

- Each process that can abort transactions defines a set of abort codes and an associated abort formatting function. The formatting function translates the abort codes into character strings.
- Each process that can abort transactions specifies which abort codes it is using.
- Each process that can be notified of an abort registers the abort formatting function for each set of abort codes used in transactions in which it participates. When the process calls the Tran-C **abortReason** function or the TX **tx_get_rollback_string** function, Encina automatically calls the formatting function associated with the abort code to translate the code into a string.

For example, to use abort codes with the initial transactional server (the order server) that we developed in Chapter 4, "Making the sample application transactional," on page 37, we define a set of abort codes, which we call order abort codes to associate them with the order server. We then write a simple abort formatting function to convert these codes into strings. At run time, the order server specifies that it is using the order abort codes for aborting transactions. It also registers the abort formatting function because, in addition to aborting transactions, it also is notified of aborts and prints out abort messages. Note that the client does not need to register the abort formatting function because the server manages transactions. The client does not receive abort notifications and thus does not handle abort codes.

In the case of the order server, the same process both initiates aborts and receives notification of them. The billing server, which is invoked from the order server, can also abort transactions, so it too must define and use abort codes. We must also write an abort formatting function associated with these billing abort codes. However, the billing server does *not* need to register this abort formatting function because it does not print abort messages; it only initiates aborts. The order server

must register the abort formatting function for the billing server abort codes; this formatting function must be linked with the order server code (that is, it is not invoked as an RPC). The order server therefore registers two abort formatting functions: one for its own abort codes and one for the abort codes used by the billing server. This is summarized in Figure 68.

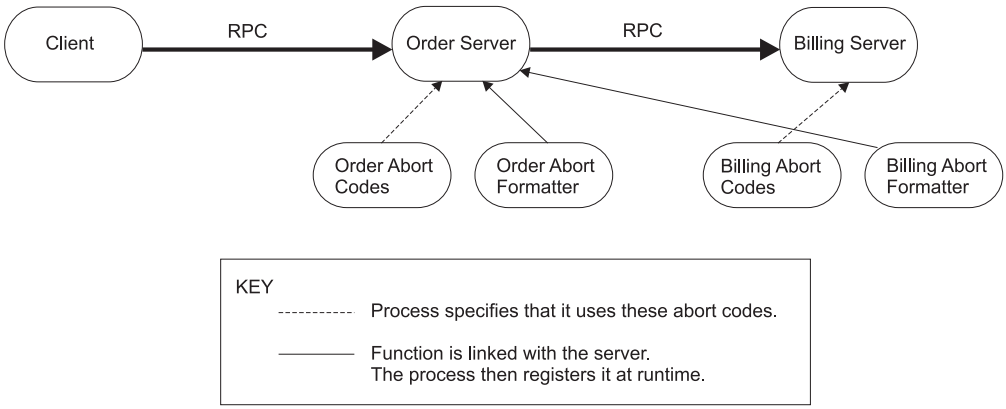

Figure 68. Using abort codes in our application

The abort code facility can be used with Tran-C, TX, or the underlying Toolkit components. In this appendix, we show only the Tran-C interface.

Defining abort codes

Defining abort codes involves the following two steps:

1. Defining the abort codes themselves
2. Writing the abort formatting function

Defining abort codes

For each condition under which an application can abort a transaction, we must define an abort code. An *abort code* is an unsigned-integer constant that indicates the reason why a transaction aborted. These codes are typically placed in a header file. In our example, we define abort codes for a few common reasons, as shown in Figure 69.

```
/* Abort codes and abort format used to abort transactions */
typedef enum {
   BAD_STOCK_NUM = 1,
   INSUFF_STOCK,
   ILLEGAL_QTY,
   RESOURCE_MGR_OP_FAILED,
   ENQUEUE_FAILED,
} order_abort_t;

static char ORDER_ABORT_FORMAT[] =
 "0014ad20-e154-1d68-85b0-9e62092caa77";
```

Figure 69. Defining abort codes for our example

Each set of abort codes must have a unique format identifier so that Encina can associate a formatting function with the abort reason generated by an aborted

transaction. The format identifier is a DCE UUID (universal unique identifier) that uniquely identifies the format for abort reasons. This format UUID is created with the DCE uuidgen utility.

Writing the abort formatting function

For each set of abort codes defined, we must provide an abort formatting function. The purpose of the formatting function is to take the information in an abort reason and use it to generate output appropriate to the application. This formatting function can then be registered by any process that wants to obtain the abort reasons. When invoked, the formatting function is automatically passed two arguments: a pointer to an abort reason for the aborted transaction and a pointer to a buffer that can be used to hold the string that corresponds to the abort reason.

In our example, the function **OrderAbortFormatter** is defined as the formatting function for the abort codes defined by the order server (see Figure 70). The function checks the abort code set for the abort reason, and based on the value of the abort code, returns a string that describes the reason for the abort in the *bufferP* parameter. Note that this example generates a printable string in English. In production, an application might instead look up abort strings in a message catalog, making it easier to internationalize the application.

```
static void OrderAbortFormatter(encina_abortReason_t *abortReasonP,
 char *bufferP)
{
 char *abortString;

 switch(abortReasonP->code) {
 case BAD_STOCK_NUM:
           abortString = "Stock number out of range.";
           break;
 case INSUFF_STOCK:
       abortString = "Stock not available in that quantity.";
           break;
 case ILLEGAL_QTY:
           abortString = "Illegal value for a stock quantity.";
           break;
 case RESOURCE_MGR_OP_FAILED:
           abortString =
                 "Operation on underlying resource manager failed.";
           break;
 case ENQUEUE_FAILED:
           abortString = "Enqueue attempt failed.";
           break;
 default:
           abortString = "Unknown abort code.";
 }
 strcpy(bufferP, abortString);
}
```

Figure 70. Example function for formatting an abort reason

Aborting a transaction with an abort code

Each module that can abort a transaction using abort codes must specify which abort codes it is using with the **useAbortFormat** function, which can be placed after the call to the **inModule** function, as shown in Figure 71 on page 90. In this example, the order server specifies that it is using the order abort codes. The billing server makes a similar call to specify that it is using the billing abort codes. Encina uses this information to determine which abort formatting function to call

when an application calls the **abortReason** function (see "Using abort data"). Note that although each module can register multiple abort format functions, it can use only one set of abort codes and thus calls **useAbortFormat** only once.

```
inModule("OrderServer");
useAbortFormat(ORDER_ABORT_FORMAT);
```

Figure 71. Specifying the abort format to use

After specifying which abort codes it is using, the module uses the **abortWithCode** function to abort transactions. The **abortWithCode** function requires one argument, an integer abort code that describes the reason for the abort. For example, in Chapter 5, "Using RQS," on page 43, when an attempt to queue a shipping request to RQS failed, the application aborted the transaction using the string &odq;Enqueue attempt failed.&cdq; We can change the application to instead specify an abort code, as shown in Figure 72. The abort formatting function for the order abort codes (shown previously in Figure 70 on page 89) can convert this abort code into a string.

```
status =
rqs_Enqueue (...);
if (status != RQS_SUCCESS)
 abortWithCode(ENQUEUE_FAILED);
```

Figure 72. Aborting a transaction using an abort code

Using abort data

In previous chapters, we used the Tran-C **abortReason** function to return a string that contains the reason a transaction was aborted. We can continue to use this function when aborting with abort codes. However, because the process that aborts transactions specifies an abort code instead of a string, we must tell Encina which formatting function to use to convert the code into a string. A module must register an abort formatting function for each set of abort codes for which it needs to print corresponding abort strings. For example, our order server must register two abort formatting functions: one for its own abort codes and one for the billing server's abort codes.

To register an abort formatting function, we use the **encina_RegisterAbortFormatter** function. This function takes two arguments: a pointer to a format UUID and the name of a formatting function. The format UUID passed to the function must be of type uuid_t. Our application uses strings to represent UUIDs (see the example header file in Figure 69 on page 88). A string representing a UUID can be converted to a UUID using the uuid_from_string DCE function. To simplify our application, we use the **REGISTER_ABORT_FORMATTER** macro, which converts the string to a UUID and then invokes the **encina_RegisterAbortFormatter** function. This macro is defined in the **order.h** header file.

Figure 73 on page 91 shows how the order server registers the abort formatting functions it uses. Figure 74 on page 91 shows the **REGISTER_ABORT_FORMATTER** macro.

```
mon_InitServer();
REGISTER_ABORT_FORMATTER(ORDER_ABORT_FORMAT,OrderAbortFormatter);
REGISTER_ABORT_FORMATTER(BILLING_ABORT_FORMAT,BillingAbortFormatter);
 mon_BeginService();
```

Figure 73. Registering abort formatting functions

```
/*
 * REGISTER_ABORT_FORMATTER -- register a function to convert abort
 * codes to strings. */

#define REGISTER_ABORT_FORMATTER(formatUuidString, formatFunction)\
BEGIN_MACRO \
 unsigned32 _status; \
 uuid_t _abortFormatUuid; \
 uuid_from_string((unsigned_char_t *) (formatUuidString), \
 &_abortFormatUuid, &_status); \
 CHECK_STATUS(_status); \
 _status = encina_RegisterAbortFormatter(&_abortFormatUuid, \
 (formatFunction)); \
 CHECK_STATUS(_status); \
END_MACRO
```

Figure 74. The REGISTER_ABORT_FORMATTER macro

After a function registers an abort code formatting function, subsequent calls to the **abortReason** function return a string formatted by the appropriate registered function. If the application needs to retrieve the code itself, it can use the **abortCode** function.

Appendix C. Source code for the example application

This appendix contains source code for the example application.

Source code for the order server

This section contains the code for the Monitor version of the order application server. It shows the initialization code and the code for the **OrderItem** function. The code for the functions called by the **OrderItem** function is given in subsequent sections.

```
#include <stdio.h>
#include <tc/tc_server.h>
#include <tpm/mon/mon_server.h>
#include "OrderInterface.h"
#include "billingInterface.h"
#include "order.h"
inModule("OrderServer");

int main (int argc, char **argv)
{
 unsigned32 status;
 char *serverName;
 int rmiId;

 /* XA switch. For actual operation, use the db_xa_switch for
 * your resource manager.
 */
 extern struct xa_switch_t db_xa_switch;

 /* Register the interface. */
 status = mon_InitServerInterface(
 MON_SERVER_INTERFACE(OrderInterface,1,0));
 CHECK_STATUS(status);

 /* Make the server recoverable. */
 status = mon_ServerRecoverable();
 CHECK_STATUS(status);

 /* Set up interaction with a resource manager. */
 status = mon_RegisterRmi(&db_xa_switch, RM_NAME, &rmiId);
 CHECK_STATUS(status);

 /* Initialize Enina and the server. */
 status = mon_InitServer();
 CHECK_STATUS(status);

 /* RQS initialization. */
 status = ConnectToShippingServer();
 CHECK_STATUS(status);
 /* Begin listening for RPCs */
 status = mon_BeginService();
 CHECK_STATUS(status);

 /* Clean-up actions. */
 DisconnectFromShippingServer();

 exit(0);
}
/*****************************************************************
 * OrderItem() -- Starts a transaction and places an order.
 *****************************************************************
```

```
 */
error_status_t OrderItem(idl_ulong_int stockNum,
   idl_ulong_int numOrdered,
   idl_ulong_int customerId)
{

 error_status_t returnStatus;
 idl_long_int costPerItem, totalCost, priority;

 transaction{
           PlaceOrder(stockNum, numOrdered, &costPerItem);
           totalCost = numOrdered * costPerItem;
      if (totalCost > 1000)
           priority = HIGH_PRIORITY;
      else
           priority = NORMAL_PRIORITY;

      PlaceItemOnQueue(stockNum, numOrdered, customerId);
      BillForItem(customerId, totalCost);
 }onCommit{
      fprintf(stderr, "We committed.\n");
      returnStatus = SUCCESS;
 }onAbort{
      fprintf(stderr, "We aborted. %s\n", abortReason());
      returnStatus = ORDER_FAILED;
 }

 return returnStatus;

}
```

Source code for RQS interactions

This section shows the soruce file contain the functions for obtaining a handle the RQS server, queueing a shipping request, and freeing the handle.

```
#include <stdio.h>
#include <rqs/rqs.h>
#include "order.h"
#include "orderRqs.h"
inModule("PlaceItemOnQueue");

/* Global Variable */
static rqs_serverHandle_t rqsHandle;

/*
 * ConnectToShippingServer -- gets a handle to an RQS server.
 */
error_status_t ConnectToShippingServer(void)
{
 rqs_status_t status;

 status = rqs_GetServerHandle(RQS_SERVER_NAME, &rqsHandle);
 if (status != RQS_SUCCESS)
 return RQS_FAILURE;
 else
 return SUCCESS;
}
/*
 * PlaceItemOnQueue -- Queues a shipping request. Aborts
 * the transaction if queueing operation fails.
 */
void PlaceItemOnQueue(idl_ulong_int stockNum,
 idl_ulong_int numOrdered,
 idl_ulong_int customerId, short priority)
{
```

```
 char *queueName;
 char *elementType = SHIPPING_TYPE;
 rqs_elementId_t elementId;
 rqs_status_t status;
 char *itemToQueue, *itemCursor;
 idl_ulong_int itemLength = sizeof(customerId)
                          + sizeof(stockNum)
                          + sizeof(numOrdered);


 /* Allocate the item to queue and pack fields into it. */
 itemToQueue = malloc(itemLength);

 /* Start copying at start of itemToQueue buffer */
 itemCursor = itemToQueue;
 memcpy(itemCursor, &customerId, sizeof(customerId));
 itemCursor += sizeof(customerId);
 memcpy(itemCursor, &stockNum, sizeof(stockNum));
 itemCursor += sizeof(stockNum);
 memcpy(itemCursor, &numOrdered, sizeof(numOrdered));

 /* Now queue item */
 if (priority == HIGH_PRIORITY)
    queueName = "priorityShippingQueue";
 else
    queueName = "normalShippingQueue";

 status = rqs_Enqueue (rqsHandle, queueName,
                                       elementType, itemLength,
                                       itemToQueue, NULL, /* Ignore work accum. */
                                        &elementId);
 if (status != RQS_SUCCESS) {
      abort("Enqueue failed.");
 }
}
/*
 * DisconnectFromShippingServer -- Frees the server handle. Called
 * during Monitor termination.
 */

void DisconnectFromShippingServer(void)
{
 rqs_FreeServerHandle(rqsHandle);
}
```

The following example shows the source code used for dequeueing shipping requests.

```
#include <rqs/rqs.h>
#include "order.h"
#include "orderRqs.h"
/* Global Variable */

static rqs_serverHandle_t rqsHandle;


error_status_t ConnectToShippingServer(void)
{
 rqs_status_t status;

 status = rqs_GetServerHandle(RQS_SERVER_NAME, &rqsHandle);
 if (status != RQS_SUCCESS)
      return RQS_FAILURE;
 else
      return SUCCESS;
}
```

```
/*
 * DequeueFromShippingQueueSet -- Dequeues and item from the
 * shipping queue set. Aborts the transaction if
 * dequeue attempt fails.
 */
error_status_t DequeueFromShippingQueueSet(idl_ulong_int *stockNum,
                                           idl_ulong_int *numOrdered,
                                           idl_ulong_int *customerId)
{
 rqs_status_t status;
 char *itemCursor;
 rqs_elementDescriptor_t *itemDequeued;
 /* Dequeue the element */

 status = rqs_QSDequeue(rqsHandle, SHIPPING_QUEUE_SET,
                                       rqs_deleteElement,
                                       TRUE, /* Wait for an element to dequeue. */
                                       &itemDequeued);

 if (status != RQS_SUCCESS)
      return DEQUEUE_FAILED;

 /* Break the item into its three components. */
 itemCursor = itemDequeued->value;
 memcpy(customerId, itemCursor,sizeof(*customerId));
 itemCursor += sizeof(*customerId);
 memcpy(stockNum, itemCursor, sizeof(*stockNum));
 itemCursor += sizeof(*stockNum);
 memcpy(numOrdered, itemCursor, sizeof(*numOrdered));

 rqs_Free (itemDequeued);
 return SUCCESS;
}



void DisconnectFromShippingServer(void)
{
 rqs_FreeServerHandle(rqsHandle);
}
```

The following example shows the simple client that interacts with the dequeueing code shown.

```
#include <stdio.h>
#include <order.h>
#include <orderRqs.h>
#include <trdce/trdce.h>
#include <tc/tc_server.h>
error_status_t ConnectToShippingServer(void);
error_status_t DequeueFromShippingQueueSet(idl_ulong_int *stockNum,
                                           idl_ulong_int *numOrdered,
                                     idl_ulong_int *customerId);
void CloseConnectionToQueue(void);

inModule("DequeueClient");

int main(void)
{
 idl_ulong_int numOrdered;
 idl_ulong_int stockNum;
 idl_ulong_int customerId;

 error_status_t status;

 /* Encina/Tran-C Initialization. */
 preInitTC();
```

```
 status = trpc_InitWithTrdce();
 CHECK_STATUS(status);
 tc_InitTRPC();
 postInitTC();

 status = ConnectToShippingServer();
 if (status != SUCCESS) {
      printf("Couldn't connect to server.\n");
      exit(1);
 }

 transaction{
      DequeueFromShippingQueueSet(&stockNum, &numOrdered, &customerId);
 }onCommit{
      printf("Stock number: %u\n", stockNum);
      printf("Number ordered: %u\n", numOrdered);
      printf("Customer ID: %u\n", customerId);
 }onAbort{
      fprintf(stderr, "We aborted. %s\n", abortReason());
 }
 quiesceTC();
 exitTC(0);

}
```

Source code for RDBMS interactions

The complete **PlaceOrder** function is shown in this section. Note that this code uses standard SQL, not embedded SQL particular to any specific RDBMS. Modifications may be required for some RDBMSs.

```
void PlaceOrder (idl_ulong_int stockNum,
                            idl_ulong_int numOrdered,
                            idl_ulong_int *costPerItem);
{
 EXEC SQL INCLUDE sqlca;
 EXEC SQL BEGIN DECLARE SECTION;
 unsigned long stockNumber;
 unsigned long pricePerItem;
 unsigned long quantityAvailable;
 EXEC SQL END DECLARE SECTION;
 stockNumber = stockNum;


 /* Determine the number in stock and the cost. */
 EXEC SQL SELECT num_available, item_cost
      INTO :quantityAvailable, :pricePerItem
      FROM inventory
      WHERE stock_num = :stockNumber;
 if (sqlca.sqlcode != SQL_SUCCESS){
      abort("Database lookup failed.");
 }
 *costPerItem = pricePerItem;

 if (quantityAvailable < numOrdered){
      abort("Insufficient stock.");
 }

 /*
 * Update the database. This update will be
 * backed out if it later turns out we do not
 * have sufficient funds in the billing database.
 */

 quantityAvailable -= numOrdered;
```

```
 EXEC SQL UPDATE inventory
      SET num_available = :quantityAvailable
      WHERE stock_num = :stockNumber;

 if (sqlca.sqlcode != SQL_SUCCESS){
      abort("Database update failed.");
 }


}
```

Source code for the billing server and PPC interactions

The source code for the billing serverand for the **BillForItem** function are shown in this section.

```
#include <stdio.h>
#include <tc/tc_server.h>
#include <tpm/mon/mon_server.h>
#include <ppc/cpic.h>
#include "OrderInterface.h"
#include "BillingInterface.h"
#include "order.h"
inModule("BillingServer");

int main (void)
{
 unsigned32 status;
 int rmId;
 extern struct xa_switch_t db_xa_switch;

 /* Register the interface. */
 status = mon_InitServerInterface(
 MON_SERVER_INTERFACE(BillingInterface,1,0));
 CHECK_STATUS(status);

 mon_InitServer();

 status = InitializePpc();
 if (status != SUCCESS){
      fprintf(stderr, "Unable to initialize PPC.\n");
      exit(1);
 }
 mon_BeginService();

 exit(0);

}
error_status_t InitializePpc(void)
{
 char *luName = LU_NAME;
 char *sideInfoFile = SIDE_INFO_FILE_NAME;
 CM_RETCODE returnCode;
 returnCode = cpic_Init(luName);
 if (returnCode)
      return INIT_FAILED;

 returnCode = cpic_ReadSideInfo(sideInfoFile);
 if (returnCode)
      return INIT_FAILED;

 return SUCCESS;
}
void BillForItem(idl_ulong_int cutomerId,
 idl_ulong_int amount)
{
```

```
 unsigned long status;
 CONVERSATION_ID conversationId;
 status = AllocateConversation(conversationId);
 if (status != SUCCESS)
      abort("Conversation Allocation Failed.");
 status = DebitCustomerAccount(conversationId, customerId, amount);
 if (status != SUCCESS)
      abort("Data Exchange Failed.");
 status = DeallocateConversation(conversationId);
 if (status != SUCCESS)
      abort("Conversation Deallocation Failed.&cdq;);
}
error_status_t AllocateConversation(CONVERSATION_ID conversationId)
{
 CM_RETCODE returnCode;
 char *smDestName = "CHECK_ACCOUNT";
 char *syncLevel = CM_SYNC_POINT;
 /* Initialize the conversation and set the synclevel. */
 Initialize_Conversation(conversationId, symDestName,
 &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;
 Set_Sync_Level(conversationId, syncLevel, &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;
 Allocate(conversationId, &returnCode);
 if (returnCode != CM_OK)
      return ALLOCATE_FAILED;

 return SUCCESS;
}
error_status_t DebitCustomerAccount(CONVERSATION_ID convId,
 idl_ulong_int customerId,
 idl_ulong_int amount)
{
 char debitInfo[MAX_SEND_LENGTH];
 int debitInfoLen = 0;
 REQUEST_TO_SEND_RECEIVED requestToSendReceived;
 CM_RETCODE status;
 /* The data is sent immediately rather than buffered. */
 SEND_TYPE sendType = CM_SEND_AND_FLUSH;
 /* Pack up the buffer to send to the peer. */
 sprintf(debitInfo, "%lu %lu", customerId, amount);
 debitInfoLen = strlen(debitInfo);

 Set_Send_Type(convId, &sendType, &status);
 if (returnCode != CM_OK)
      return CPIC_ERROR;

 Send_Data(convId, debitInfo, &debitInfoLen,
                     &requestToSendReceived,&status);
 if (returnCode != CM_OK)
      return SEND_ERROR;

 return SUCCESS;
}
error_status_t DeallocateConversation(CONVERSATION_ID conversationId)
{
 CM_RETCODE returnCode;
 Set_Deallocate_Type(conversationId, CM_DEALLOCATE_SYNC_LEVEL
                                        &returnCode);
 if (returnCode != CM_OK)
      return DEALLOCATION_FAILURE;

 Deallocate(conversationId, &returnCode);
 if (returnCode != CM_OK)
```

```
      return DEALLOCATION_FAILURE;

 return SUCCESS;

}
```

TIDL and TACF files for the application servers

The TIDL file for the sample application is shown in the following example.

```
[
uuid(0016a2ce-526e-1eba-bda1-9e620a3aaa77),
version(1.0)
]
interface OrderInterface
{
import "tpm/mon/mon_handle.idl";

[nontransactional] long OrderItem(
                           [in] unsigned long stockNum,
                           [in] unsigned long numOrdered,
                           [in] unsigned long customerId);

}
```

The TACF file for the sample application is shown in the following example.

```
[explicit_handle (encina_handle_t  void)]
interface OrderInterface
{
 [comm_status, fault_status] OrderItem();
}
```

The TIDL and TACF files for the billing server are shown in the following examples.

```
[
uuid(002513cc-da57-1ec0-8a0f-9e620a3aaa77),
version(1.0)
]
interface BillingInterface
{
import "tpm/mon/mon_handle.idl";

[transactional] void BillForItem(
           [in] unsigned long customerID,
           [in] unsigned long amount);

}
```

```
[explicit_handle (encina_handle_t void)]
interface BillingInterface
{
}
```

Application include files

The following example shows the **order.h** header file.

```
/* order.h */
#include <stdio.h>
#include <dce/rpc.h>
#include <dce/dce_error.h>
#include <encina/encina.h>
#include <encina/afac.h>
#include <tc/tc.h>
#define SUCCESS 0
```

```
#define ORDER_FAILED 1
#define INIT_FAILED 2
#define ALLOCATE_FAILED 3
#define CPIC_FAILURE 4
#define SEND_ERROR 5
#define DEALLOCATION_FAILURE 6

/* Order priorities */
#define HIGH_PRIORITY 1
#define NORMAL_PRIORITY 0


/* Boolean constants */
#define FALSE 0
#define TRUE 1

/* Macro delimiters */
#define BEGIN_MACRO do {

#define END_MACRO } while (0)

/* FATAL -- Failure.Print error message and exit the program.*/

#define FATAL(args) \
BEGIN_MACRO \
 printf args; \
 exit(1); \
END_MACRO

/* CHECK_STATUS: Make sure status is 0; print error msg &
 * exit if it isn't. */

#define CHECK_STATUS(status) \
BEGIN_MACRO \
 char _errorMsg[ENCINA_MAX_STATUS_STRING_SIZE]; \
 if (status) { \
 encina_StatusToString(status, \
 ENCINA_MAX_STATUS_STRING_SIZE, \
 _errorMsg); \
 FATAL(("%s(%d): %s\n", __FILE__, __LINE__, _errorMsg)); \
 } \
END_MACRO
```

The following example shows the include file used by the RQS part of the sample application.

```
#define SHIPPING_TYPE "shippingType"
#define RQS_SERVER_NAME "/.:/order_cell/server/rqsShippingServer"
#define SHIPPING_QUEUE_SET "shippingQueueSet"
#define DEQUEUE_FAILED 10
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

AIX	C-ISAM
CICS	CICS/400
CICS/6000®	CICS/ESA
CICS/MVS	CICS/VSE
Database 2™	DB2

DB2 Universal Database™	DFS
Domino™	Encina
IBM	IMS™
Informix	Lotus®
MQSeries	MVS
MVS/ESA	Notes®
OS/2	RACF®
SecureWay	SupportPac™
System/390	TXSeries
VisualAge®	VTAM®
WebSphere®	

Domino, Lotus, and LotusScript are trademarks or registered trademarks of Lotus Development Corporation in the United States, other countries, or both.

ActiveX, Microsoft®, Visual Basic, Visual C++, Visual J++, Visual Studio, Windows, Windows NT®, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Pentium® is a trademark of Intel™ Corporation in the United States, other countries, or both.

This software contains RSA encryption code.

Other company, product, and service names may be trademarks or service marks of others.

Index

P

Q

R

S

T

U

V

X

Printed in USA

SC09-4486-02

Spine information:

TXSeries™ Writing Encina Applications Version 5.1 SC09-4486-02