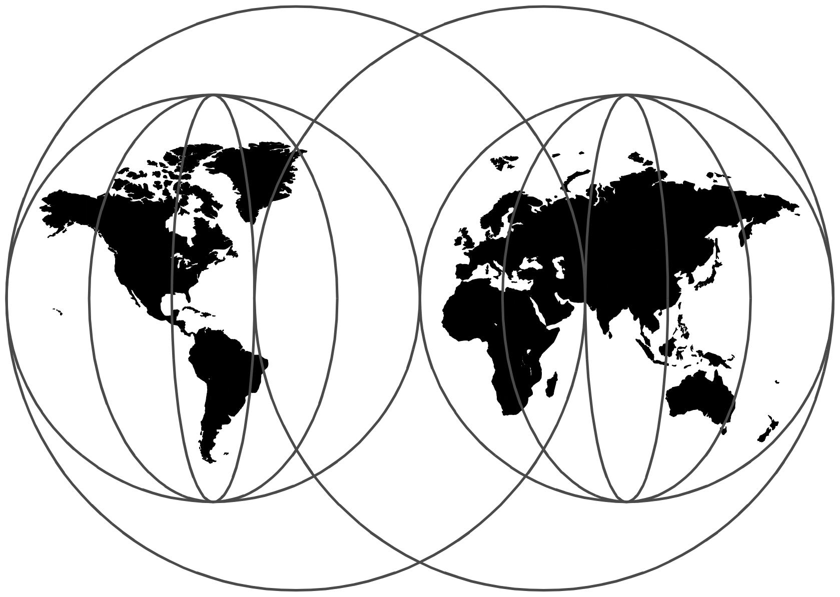
# Developing Distributed Transaction Applications with Encina

*Hanspeter Nagel, Bill Ruchte, Radoslav Nikolov, Pankaj Gupta*



**International Technical Support Organization**

http://www.redbooks.ibm.com

SG24-5241-00

SG24-5241-00

International Technical Support Organization

# Developing Distributed Transaction Applications with Encina

December 1998

**Take Note!**

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special Notices” on page 413.

**First Edition (December 1998)**

This edition applies to Encina Distributed transaction processing system version 2.5, for use with Operating Systems UNIX and Windows NT.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE  Building 80-E2
650 Harry Road
San Jose, California 95120-6099

# Contents

# Figures

# Tables

# Preface

This redbook is primarily intended for application developers, system architects, and IT managers to understand the principals about transaction application development using the Encina Software from IBM/Transarc.

The redbook consists of four parts. Part 1 gives a short introduction to Transaction Processing and how to position the different Encina modules.

Part 2 discusses the specifics of the different Encina APIs, explains how to use them and discusses some specific application design considerations.

Part 3 illustrates through a case study, how to design and develop a solution utilizing the Encina APIs. After an analysis and architecture description, we show how to design an Encina based solution, and how to develop the necessary code. The software developed through this phase is partly available on http://www.redbooks.ibm.com

Part 4 finally discusses the important issues about configuration, administration, and maintenance of Encina infrastructure. In addition it also covers the deployment tasks of Encina based applications and shows some useful tools.

In the case study we describe in part 3 we study a fictional customer who wants to enter the e-commerce market for direct sale. This gives him a market advantage against his competitors because of faster delivery process handling, higher degree of backoffice automation and better customer service through individualized marketing. While you read this case study, you may detect some needs already discussed in so many meetings....

## The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization San Jose Center.

**Hanspeter Nagel** is an Senior Systems Engineer at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on all areas of Transaction Systems. Before joining the ITSO, Hanspeter Nagel worked in the service organization of IBM Switzerland where he was responsible for DCE, Security, and Distributed Transaction Systems in several customer projects. You can reach him by e-mail at hnag@us.ibm.com.

**Bill Ruchte** is the Chief Technical Officer of Trifolium, an IBM Business Partner specializing in Encina application development.  Bill has been involved in a number of large distributed system development efforts using many different technologies.  Prior to founding Trifolium, Bill worked for Imonics and General Electric in system architecture and project management roles.  Bill is also an Adjunct Member of the Faculty at North Carolina State University.

**Pankaj Gupta** is a Technical Consultant with Transarc Corp. in USA. Pankaj Gupta has an extensive experience in all major client-server systems and Encina Technology and worked in several large projects.

**Radoslav Nikolov** is a Technical Consultant with Transarc Corp. in USA. Radoslav Nikolov has an extensive experiences in all major questions of multi-platform client-server systems and managed several large projects in this area.

Thanks to the following people for their invaluable contributions to this project:

Maggie Cuttler, Emma Jacobs, Elsa Barron, Mary Comianos, and Laymond M. Pon of the International Technical Support Organization, San Jose Center

Michael Lundblad
Transarc

Rose Palombo
Transarc

Tony Parente
Transarc

Douglas Ayers
Transarc

Doug Morin
Trifolium

Heiner Tschopp
IBM Switzerland

## Comments Welcome

**Your comments are important to us!**

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 443 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

| | |
|---|---|
| For Internet users | `http://www.redbooks.ibm.com` |
| For IBM Intranet users | `http://w3.itso.ibm.com` |

- Send us a note at the following address:

  `redbook@us.ibm.com`

# Part 1. Distributed Processing and Encina

In Part 1 after a brief introduction to distributed transactions, their components, and enabling standards, we provide an overview of the Encina technology and the different Encina modules. This part is primarily for readers who want to develop a general understanding of Encina.

# Chapter 1. Overview of Distributed Processing

In this chapter we provide an overview of distributed processing and the technologies related to Encina.

Several trends have dramatically changed the way information is processed. At the same time that the cost of personal computers and workstations has been steadily decreasing, their computing power has been increasing. In addition, local area networks and high-performance wide area networks are becoming cheaper and faster. Thus it has become quite cost effective to provide desktop computing power to users who can share information and resources with one another.

Businesses are also evolving. Separate organizations are merging, resulting in a combination of two completely different computer infrastructures. Previously autonomous entities within an organization are interacting with other entities within the organization or with other organizations, resulting in information exchange among entities that have different computers and different data representation methods.

The paradigm of computing is shifting from centralized computing to distributed computing, that is, computing that involves the cooperation of two or more machines communicating over a network. One advantage of this shift is that the machines participating in the system can range from personal computers to supercomputers; the network can connect machines in one building or on different continents.

Another advantage of distributed computing is resource sharing. If special-function hardware or software is available over the network, that functionality does not have to be duplicated on every computer system that needs to access it. For example, an organization could make a typesetting service available over the network, allowing users throughout the organization to submit their jobs to be typeset.

In addition to the cost-effectiveness of many small computers working together, having many units connected to a network is the more flexible configuration. If more resources are needed, another unit can be added in place, rather than bringing the whole system down and replacing it. The ability to replicate data and functionality has made distributed systems more reliable and available than centralized systems. For example, when a file is copied on two different machines, even if one machine is unavailable, the file can still be accessed on the other machine. Likewise, if several printers are attached to a network, even if an administrator takes one printer offline for maintenance, users can use an alternative printer to print their files.

The shift from centralized computing to distributed computing raises several new concerns. Because the data accessed could be anywhere in the network, naming conventions must be in place to identify the data and services must be available to locate data. Complicating this task is that the location of the data may be dynamic, that is, the data may be moving from one location to another over time. Additionally, the data could become unavailable because of network or computer failures

An other concern is security. The system that maintains the data must allow access to authorized external users and deny access to unauthorized external users. The computer systems in a distributed system must cooperate to exchange information and maintain data integrity.

## 1.1 Distributed Computing Environment

The Distributed Computing Environment (DCE) is a cross-platform, comprehensive, integrated set of services that support the development, use, and maintenance of distributed computing applications (see Figure 1 on page 5). The availability of a uniform set of distributed computing services anywhere in the network gives applications an effective means of harnessing the power inherent in networks of computers that may otherwise be unused. DCE is available on a broad range of platforms from desktop systems to UNIX workstations to mainframes and is an overall accepted standard.

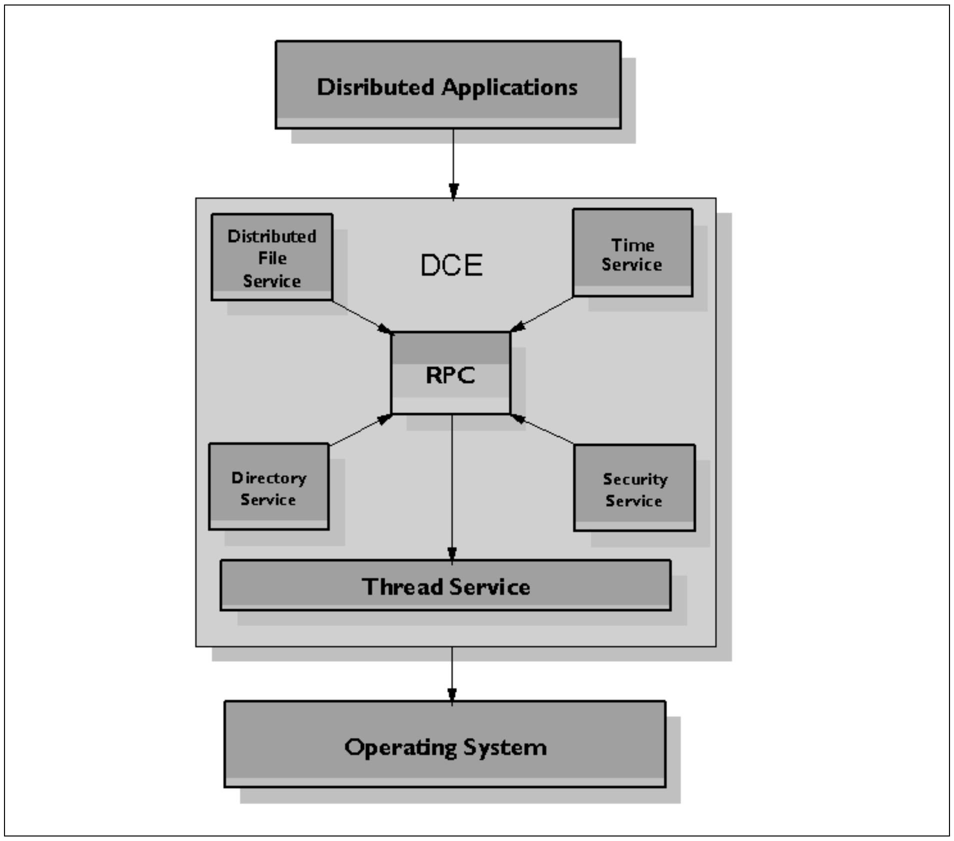


*Figure 1. DCE Architecture*

DCE offers a fully integrated, production-ready distributed computing environment based on an architecture designed to accommodate new distributed computing technologies in the future. It provides a communications environment that supports information flow from wherever information is stored to wherever it is needed, without exposing the network's complexity to the end user, system administrator or application developer.

The DCE architecture is a layered model that integrates a set of eight fundamental technologies from the most basic supplier of services (the operating system) to the highest level consumers of services (the applications). Security and management services are essential to all layers of the environment. To applications, DCE appears as a single logical system that can be organized into two broad categories of services: the DCE Secure Core and DCE Data Sharing services.

The DCE Secure Core services provide software developers with the tools to create end-user applications and system software products for distributed computing. These services include:

**Thread Service:** DCE supports multithreaded applications such as programs that use lightweight processes to perform many actions concurrently. DCE threads are based on the POSIX threading standard.

**Remote Procedure Call (RPC):** The DCE RPC is the fundamental communications mechanism, allowing direct calls to procedures on remote systems as if they were local procedure calls. This mechanism simplifies development of distributed applications by eliminating the need to explicitly program the network communications between the client and server. The DCE RPC mechanism masks differences in data representation on different hardware platforms, allowing distributed programs to work transparently across heterogeneous systems.

**Directory Services:** The DCE Cell Directory Service (CDS) is the mechanism for logically naming objects within a DCE cell (a group of client and server machines). Applications identify resources by name, without needing to know where the resources are located. DCE cells can also participate in a worldwide directory service, using the DCE Global Directory Service (GDS), which uses the Internet-style Domain Name Service (DNS).

**Security Service:** The DCE Security Service provides the mechanisms for writing applications that support secure communications between clients and servers. The Security Service enables processes on different machines to confirm one another’s identities, allows a server to determine whether a given user is authorized to access a particular resource, and supports several security levels for protecting messages as they travel across the network. The DCE Security Service is the foundation for enterprisewide security solutions, such as the IBM single signon product and IntelliSoft Corp.’s DCE/Snare.

**Time Service:** The DCE Distributed Time Service (DTS) synchronizes the clocks on different machines in a distributed system.

**Audit Service:** The DCE Audit Service provides the mechanism for detecting and recording critical events in distributed applications. The Audit Service logs audit records on the basis of specified criteria. An administrative command interface selects the events to be recorded on the basis of certain criteria. An event classification mechanism allows the logical grouping of a set of events for ease of administration.

**Naming Gateway:** The DCE Naming Gateway enables Microsoft Windows NT RPC clients to use the DCE Naming Service to locate servers without installing additional DCE software on the client machine.

DCE offers these features and benefits:

- Comprehensiveness: DCE encompasses all of the facilities necessary for building distributed applications. It integrates all of these services into a single, logical structure, enabling programmers and administrators to develop and manage distributed applications as easily as traditional, single-system programs.
- Powerful APIs: DCE provides the developer with an integrated set of high-level construct RPCs, threads, security, and naming services that mask the complexity and diversity of the network. This simplifies the development task and facilitates the porting of applications across platforms.
- High Performance: DCE is designed for high performance and availability. Critical DCE servers can be replicated to support large numbers of clients and ensure high availability.
- Interoperability: DCE provides a high level of interoperability and supports the construction of applications that interoperate seamlessly in multivendor environments. DCE software is Open Software Foundation (OSF) certified.
- Portability: DCE is network independent and operating-system independent and is backed as a standard by The Open Group (X/OpenLtd., OSF), numerous end users, and independend software vendors (ISVs), and most major computer vendors.

## 1.2 Common Object Request Broker

The Common Object Request Broker Architecture (CORBA) is a specification of the Object Management Group (OMG). The objective of the CORBA standard is to allow interoperability among diverse software and hardware products available from different vendors. CORBA enables various vendors to implement an application component with different underlying implementations as long as they conform to the interface of the component. Thus an application function can be invoked without regard to who has implemented it or where it is located. CORBA 1.1, introduced in 1991 by OMG, defined the interface definition language (IDL) and the application programming interfaces (API) that enable client/server object interaction within a specific implementation of an object request broker (ORB). The ORB is the glue that matches a client request to an appropriate server implementation. CORBA 2.0, adopted in December 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client-server relationship between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the client call and is responsible for finding a server object that can implement the request, pass it the parameters, invoke its method, and return the results. The client is independent of the object’s location, its programming language, its operating system, or any other system aspects that are not part of an object's interface. The ORB thus provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

The interface definition specifies the interface that an object supports. The object can then be implemented in any operating system and with any language, as long as it implements the required functionality and supports the defined interface. ORBs allow programmers to choose an appropriate operating system, execution environment, programming language, and algorithms to support an interface definition.

Additionally, ORBs allow the integration of existing components. In an ORB-based solution, developers simply model the legacy component, using the same IDL they use for creating new objects, and write wrapper code that translates between the standardized business and the legacy interfaces.

In summary, CORBA provides object-oriented standardization and interoperability. With CORBA, users access information transparently, without having to know on which software or hardware platform it resides or where it is located on an enterprise’s network. The communications heart of object-oriented systems, CORBA brings true interoperability to today's computing environment.

## 1.3 Web Server

A Web server maintains information that users on the World Wide Web can access. A user running a Web browser can access information stored on a Web server. Typically the web server stores and maintains the data, and the browser formats and displays the data. Web browsers also support the download and execution of applets, which are applications that can be downloaded from the Web server and executed on the Web browser (client). Thus, the World Wide Web supports a client/server environment.

With the growing popularity of the World Wide Web, existing (non-Web-based) applications have to be migrated so that they can be executed over the Web. Simple, straightforward application programs that run on low-end client machines offer large numbers of users an effective means to access Web servers and business-critical applications. Thus companies can leverage the Web, using smaller client machines across their enterprises for more efficient system operation.

The Internet DE-Light client, developed using the Java programming language, supports access to existing DCE and Encina applications through desktop PCs and workstations with Web browsers. The Internet DE-Light client (and its applications) provides the following key features:

- Portability of applications across a wide variety of platforms
- Consistent application look and feel because of a simplified high-level screen-definition interface
- Dynamic linking for automatic downloading of applets

The Internet DE-Light client enables Java applets running on any platform to invoke DCE- and Encina-based application servers located across the network. Applets download automatically from the Web server any time a browser accesses a Web page that requires DCE or Encina services. Specific installation of Internet DE-Light client components is not required on the client machine.

The client gateway protocol is simple and efficient and works well on low bandwidth channels like the dial-up lines common for many Web users.

With the Internet DE-Light client, Encina developers can leverage Java's benefits to create client applications that are visually consistent across platforms, easier to manage and more portable than before, and compatible with other DCE and Encina clients and existing application servers.

The Internet DE-Light client uses the secure sockets layer (SSL), an industry standard, to protect data as it travels between the client and gateway across the network. SSL provides data encryption and message integrity for a TCP/IP connection. Once at the gateway, DCE credentials are obtained for full-level security.

## 1.4 Transaction Processing

Transaction processing systems are widely used by enterprises to support mission critical applications. These applications need to store and update data reliably, provide concurrent access to data by hundreds or thousands of users, and maintain data integrity despite failures of individual system components.

A transaction is a tool for distributed systems programming that simplifies failure scenarios, and it 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.

Transactions provide ACID properties:

- Atomicity. A transaction's changes are atomic: either all operations that are part of the transaction occur or none occurs.
- Consistency. A transaction moves data between consistent states.
- Isolation. Even though transactions can execute concurrently, one transaction does not see 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 occur. 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.

In a distributed system, a transaction can access and update data across many different computers. To maintain the ACID properties of the transactions, these computer systems must cooperate with each other to guarantee data integrity and availability.

### 1.4.1 Transaction Processing Monitor

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. They also help provide 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, perform intercomputer communications, and design and manage the user interface.

### 1.4.2 Encina Monitor

The Encina Monitor run-time environment coordinates TP applications and resource managers and performs run-time administration tasks, such as load balancing and collecting diagnostics. In addition, this environment provides for other interactions with the execution environment, such as scheduling calls for later execution and retrieving information about users, transactions, and client/server bindings. The run-time environment provides support for coordinating a transaction in a distributed environment. It supports database access as well as access to data stored in mainframe systems. It provides a secure environment for running applications.

The monitor run-time environment also monitors the state of the system and the state of the server processes. In case of a failure of a server process, the monitor detects the failure and automatically restarts the server, increasing the availability of the system and providing additional robustness.

The monitor system administration interface is used to construct, initiate, control, and terminate a monitor system. The monitor is administered through monitor administrative and configuration interfaces.

Monitor applications are developed by using the monitor API in conjunction with other Encina interfaces, such as Tran-C. The monitor saves the programmer effort by performing some tasks, such as interaction with DCE RPC and security, on the application’s behalf. Thus, the Encina monitor simplifies the task of writing applications by automating several low-level tasks.

# Chapter 2. Overview of Encina

Whereas DCE offers great value as a set of consistent and interoperable cross-platform services, Encina extends these services by providing support for such essential services as distributed transactions, load balancing, an execution and systems administration infrastructure, and simplified development. Encina is based on a modular, layered architecture that builds on top of DCE. The lower layers of Encina extend DCE to form a set of core technologies that enable client/server transaction processing and the management of recoverable data.

A key benefit of Encina is its modular architecture, providing functionality in an organized way (see Figure 2 on page 13).

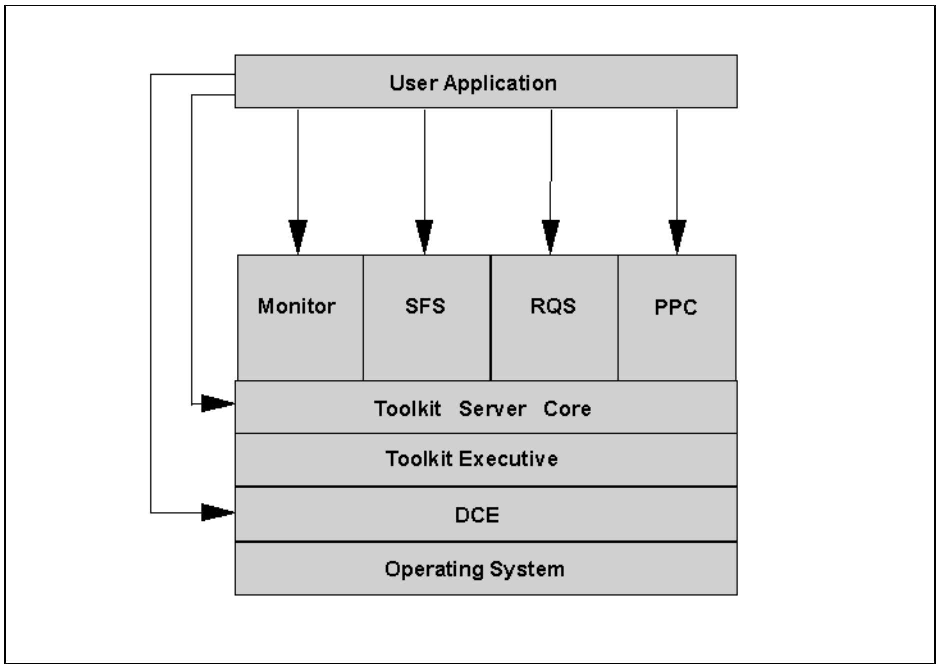


*Figure 2. Encina Overview*

The Encina toolkit provides services for failure recovery, integrating with relational databases through the industry-standard XA interface, and distributed transactions. The Structured File Service (SFS) provides a record-oriented file system that is very similar to VSAM on mainframes. The Peer-to-Peer Communications (PPC) component provides Logical Unit 6.2 (LU 6.2) connectivity services to applications and data on mainframes and other enterprise platforms. The Recoverable Queuing Service (RQS) provides full-featured, fault-tolerant queuing services. The Encina Monitor simplifies development, provides execution support for starting and stopping servers, fault detection and correction, and single-image systems management. While Encina uses a modular architecture, Encina-based applications remain efficient and perform well because of the specific way in which Encina uses shared libraries and links into application programs.

## 2.1 Product Suite

Encina is a modular system that is layered on top of the DCE services. The Encina product suite consists of the following modules:

- Encina Toolkit Executive
- Encina Toolkit Server Core
- Encina Structured File Server
- Encina Recoverable Queuing Service
- Encina Monitor
- Encina PPC Executive
- Encina PPC Gateway/SNA

Encina 2.5 additionally provides expanded support for object-oriented programming by supporting the OMG's Object Transaction Service (OTS) and Object Concurrency Control Service (OCCS) on top of a commercial ORB.

### 2.1.1 Encina Toolkit

As shown in Figure 2 on page 13, the modules of the Encina Toolkit are grouped into two major components:

- **Toolkit Executive:** The Executive provides services that permit a process to initiate, participate in, and commit distributed transactions. These services include transactional extensions to DCE RPCs that ensure transactional integrity over distributed computations transparently. The Executive also supports nested transactions, a feature that provides failure containment and simplifies application development.
- **Toolkit Server Core:** Built on the Executive, the Server Core provides facilities for managing recoverable data that is accessed and updated transactionally. These facilities include a locking library to serialize data access, a recoverable storage system to allow transactions to roll back or roll forward after failures, and an X/Open XA interface to permit the use of XA-compliant resource managers.

The modular nature of the Toolkit provides a flexible environment for developing transaction processing applications or moving existing transaction processing applications from one computer system to another. In addition, the Toolkit provides interfaces in the C programming language, which makes it easily used on many systems.

### 2.1.2 Encina Structured File Server

The Encina SFS is a record-oriented file system that provides transactional integrity, log-based recovery, and broad scalability. It supports very large databases, and it is fully capable of participating in two-phase commit protocols, allowing multiple SFS servers to be used in a single transaction.

SFS uses structured files, which are composed of records. It provides X/Open ISAM-compliant and VSAM-like interfaces. 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.

### 2.1.3 Encina Recoverable Queuing Service

The 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 transactional at a later time.

### 2.1.4 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. In conjunction with resource managers, the Monitor 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 interoperates with existing computing resources such as IBM mainframes running CICS.

### 2.1.5 Encina Peer-to-Peer Communication

PPC Services enable Encina transaction processing systems to interoperate with systems, typically mainframes, that have Systems Network Architecture (SNA) LU 6.2 communication 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.

## 2.2 Encina Connectivity

Encina provides connectivity among various components. As mentioned before Encina supports client/server computing in a distributed environment. The transactional application may span multiple systems, using the DCE facilities for name lookup, RPCs and security. Encina provides the mechanisms for multiple systems to interact with each other to ensure the consistency of data and maintain the ACID properties of transactions.

Additionally Encina supports connectivity with external components. An example of an external component is a database back-end system that is used to store data. In this section we describe how Encina provides connectivity to several such systems.

### 2.2.1 Encina - Web

The DE-Light product is a set of APIs and a gateway server that you can use to extend the power of the DCE and Transarc Encina to personal computers and other systems that are not running as DCE clients.

You can use DE-Light to build clients that require less overall effort to create, involve fewer administrative resources, and generate less network traffic than standard DCE or Encina clients. Yet these DE-Light clients can still take advantage of the benefits of load balancing, scalability, and server replication that were formerly available only to full DCE and Encina clients. In addition, DE-Light enables you to access DCE and Encina from systems that do not support DCE but do support Java.

DE-Light is available as a separate product and consists of the following three components:

- Java API: Used to develop DE-Light Java client applications (also known as "Internet DE-Light clients").
- C API: Used to develop DE-Light C clients for the Microsoft Windows NT and Windows 95 environments.
- Gateway server: Enables communications between Internet DE-Light clients and DCE or Encina servers.

In the DE Light environment, clients use a simplified RPC protocol to communicate with a DE-Light gateway. The DE-Light gateway, a server process running within the DCE cell, translates the simplified RPCs into full DCE RPCs or Encina transactional RPCs (TRPCs) and communicates with DCE or Encina servers on behalf of the clients. The gateway also translates communications from these servers into responses for the clients, thus completing the communications loop between clients and servers.

Communications between Internet DE-Light clients and the DE-Light gateway can use the TCP/IP, HTTP, or HTTPS transport protocols. Internet DE-Light clients can use the Internet as the transport mechanism for secure commercial applications. Clients need only connect to the Internet through a local service provider to securely access DCE and Encina servers.

DE-Light Java clients can consist of either:

- A java applet residing on a Web server or a stand-alone java application
- A Java applet embedded in a HyperText Markup Language (HTML) document residing on a Web server. With this type of client, a user contacts a Web server through a Java-enabled Web browser, and the browser automatically downloads the DE-Light Java client applet along with the HTML document.

The DE-Light Java client applet then contacts the appropriate DE-Light gateway. The gateway is a separate server that resides on a DCE client machine. As shown in Figure 3 on page 17, the Web client can now communicate with DCE or Encina servers through the DE-Light gateway.

In previous versions of Netscape and Microsoft Internet Explorer, the gateway had to reside on the same machine as the Web server. With Netscape Version 4.0 and Microsoft Internet Explorer Version 4.0, the use of signed applets removes this restriction.

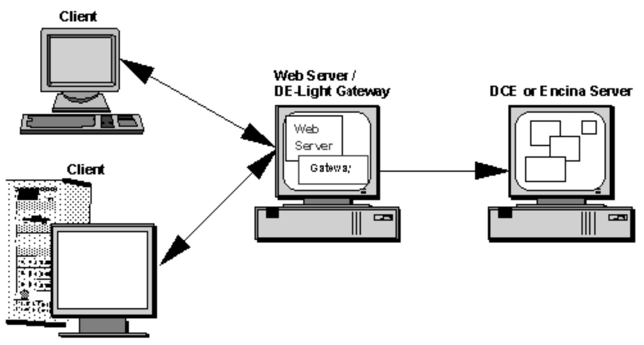


*Figure 3. DE-Light: Going through a Gateway*

Alternatively, the application can be written as a stand-alone Java application (see Figure 4 on page 18). This DE-Light Java client communicates with a DE-Light gateway at a known TCP/IP or HTTP address.

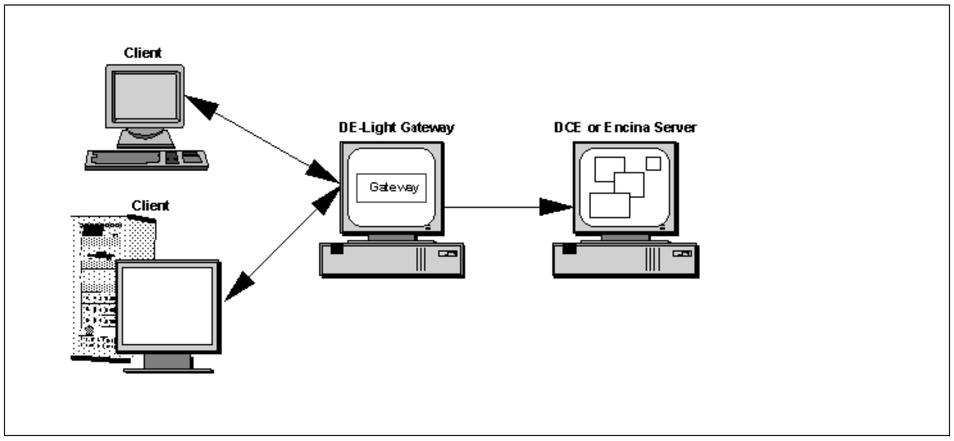


*Figure 4. DE-Light: Stand-alone Java Application*

DE-Light C clients use simplified RPCs to communicate with DE-Light gateways. These simplified RPCs are translated by the gateway into full DCE RPCs and TRPCs and are sent along to the appropriate DCE or Encina servers. The gateway then translates the RPCs and TRPCs and returns any output and status codes to the clients.

A traditional DCE or Encina application uses the DCE RPC mechanism to communicate directly with DCE or Encina servers. The client application is compiled with IDL information, which defines all of the DCE RPCs or Encina TRPCs that clients understand. Each computer that runs these clients must have installed and configured a full client implementation of DCE.

A DE-Light client accesses DCE or Encina servers through a proxy, or intermediary: the DE-Light gateway. The DE-Light gateway runs on a DCE client machine or server host machine and acts as an RPC client of the DCE or Encina servers; it transmits and translates RPC requests from the DE-Light client to the application servers and returns responses from those application servers to the DE-Light client.

Because the DE-Light gateway acts as a DCE or Encina client, it must be provided with information about the server interfaces with which it communicates. The gateway acts as an IDL interpreter; it loads IDL or transactional IDL (TIDL) files that provide information about the RPCs and TRPCs clients use for communications. Because you can load and unload

RPC definitions while the gateway is running, you can easily reconfigure the gateway to support updated or new DE-Light applications.

### 2.2.2 Encina - OS/390 Interoperability

Frequently a newly designed Encina application has to access existing legacy applications and data on a mainframe. Most legacy applications are developed using CICS or Information Management System (IMS) on MVS. Communication between distributed systems, such as AIX or Windows NT, and MVS is through either SNA or TCP-IP.

#### 2.2.2.1 The Application Support Server

The Application Support server (AS) enables a DCE client application located anywhere in the DCE environment to access the resources of CICS and IMS on OS/390 (see Figure 5 on page 19). To call a CICS or IMS application program, the client program can use the DCE RPC.

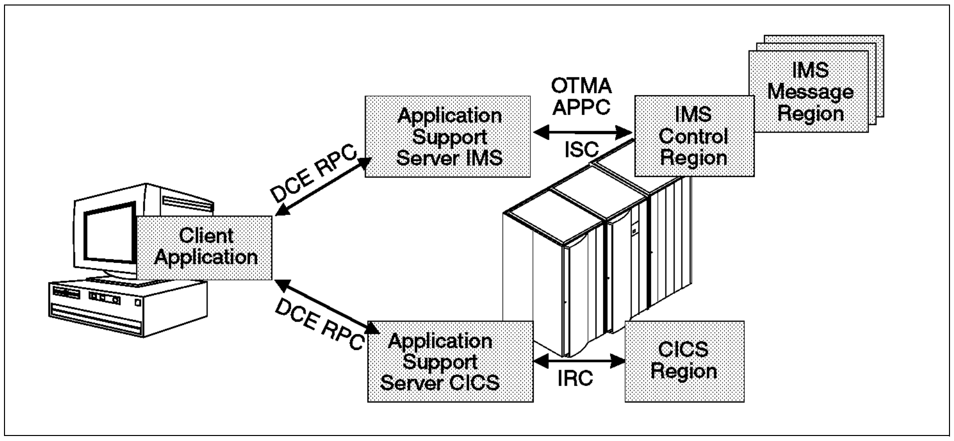


*Figure 5. OpenEdition DCE AS CICS and IMS Application Support Servers*

The OS/390 environment for accessing CICS and IMS consists of an AS server and OS/390 OpenEdition DCE. The AS server supports client programs written in C and CICS and IMS application programs written in COBOL or C.

The DCE RPC runtime handles the conversion of COBOL and C data types. Components of the OS/390 OpenEdition DCE handle conversion of EBCDIC and ASCII data types, if needed.

#### 2.2.2.2 The OS/390 Encina Toolkit Executive

The OS/390 Encina Toolkit Executive enhances DCE RPCs with transactional semantics by implementing a two-phase commit protocol that synchronizes related pieces of work taking place in different processes. The protocol guarantees that all processes successfully complete the work or that the work is not performed at all. For example, changes to data either all fail or all succeed. The goal is to ensure that each participant in a transaction, that is, each resource manager updating data in the transaction, takes the same action (commits or aborts). The OS/390 Encina Toolkit Executive provides the necessary Encina interfaces such as TRAN, TRPC, Tran-C, and ThreadTid to develop these transactional interfaces to IMS.

#### 2.2.2.3 Encina-IMS Connection

As shown in Figure 5 on page 19, the AS server passes data to and from IMS. To pass the data, the AS server must communicate with IMS. You can use one of three methods of communication between IMS and the AS server:

- Open Transaction Manager Access (OTMA)
- APPC/IMS
- IMS intersystem communications (ISC).

***OTMA***

OTMA is a transaction-based, connectionless, client-server protocol for OS/390 in an MVS sysplex environment. It uses the MVS cross-system coupling facility (XCF) as its transport layer. OTMA, although similar to the ISC and APPC adapters, provides transactional capability when used with AS IMS server of the OS/390 Encina Toolkit Executive. Because the OTMA adapter supports the transactional RPC from Encina, it is the most interesting one from an Encina perspective. Figure 6 on page 21 shows how an Encina application can access IMS through TRPC and OTMA.

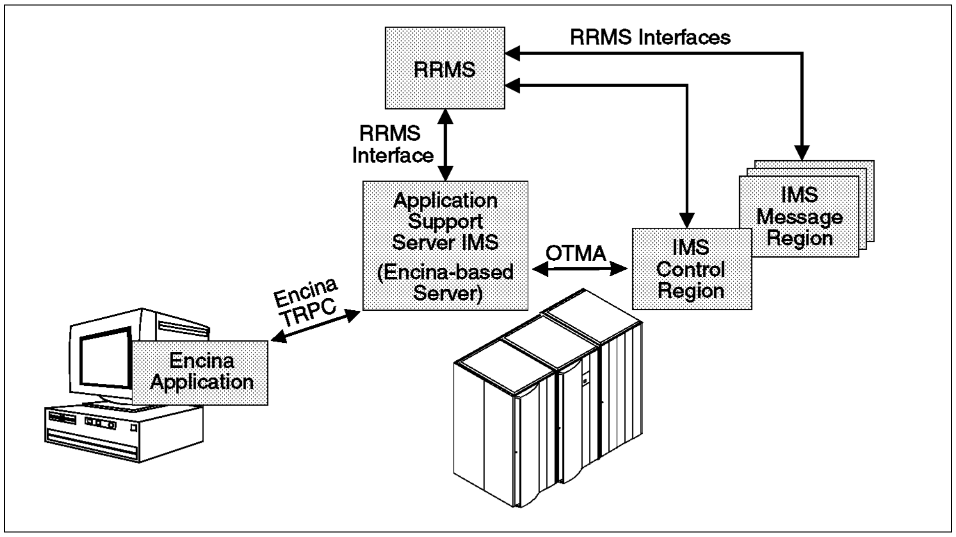


*Figure 6. Transactional Access from an Encina Client to IMS*

***APPC/IMS***

APPC/IMS is a part of the IMS Transaction Manager (TM) and enables IMS application programs to communicate with other programs through APPC. APPC/IMS is based on APPC/MVS. AS for IMS with APPC supports implicit communication between APPC/IMS and IMS. Therefore IMS transactions communicate with APPC through the IMS message queue. Implicit support lets APPC conversations call existing, unmodified IMS application programs that use data language 1 (DL/I) calls for message handling, without the explicit use of APPC verbs themselves.

***ISC***

The AS server appears to the IMS subsystem as a pool of ISC parallel sessions. Each ISC parallel session is defined as a logical terminal to IMS. The number of sessions that can run concurrently is the lesser of the maximum number of ISC parallel sessions defined by the IMS administrator for the AS server and the maximum number of sessions defined in VTAM.

#### 2.2.2.4 TP Considerations for AS Servers for IMS with OTMA

An Encina client application begins a transaction by calling `tran_Begin()`. The same application can later call `tran_End()` to try to commit the transaction. Any participant in the transaction can call `tran_Abort()` to abort the transaction.

Application programs can also use Tran-C, which simplifies transaction demarcation, concurrency control, and exception handling. It provides the transaction construct, which lets you bracket a transaction to delimit it, and it provides clauses for successful completion or failure.

The AS server has a timeout clock that cancels a conversation with IMS if either IMS or the client (between conversational transaction RPCs) does not respond within an installation-defined interval (the default is 60 seconds). A conversation that is canceled by the AS server is referred to as an *orphaned conversation*. If you expect the default interval to be exceeded while processing your client program or IMS transaction, you can contact your administrator to adjust the interval to accommodate your transactions by using the _ASU_EXPIRE environment variable. The same timeout value applies to all transactions handled by an AS server.

The following restrictions apply to client programs that use the OTMA server for transactional RPCs:

- For a transactional RPC, the application does not issue commits from IMS transactions, but it can issue rollbacks from IMS transactions. These affect the outcome of the entire distributed sync point.
- Errors that the AS server for IMS detects are reflected back to the client through the op_rc and return code values. The client program should examine these in addition to DCE and Encina aborts.
- An IMS message region processing an IMS transaction, called as a result of a transactional RPC, stays active until the global outcome is resolved. Any locks this IMS transaction holds are retained during this time.
  If an IMS transaction is called multiple times within a sync point, IMS must be able to schedule each invocation in a separate message region.
- If multiple transactions access the same IMS, IMS resource locks are transferred to treat all resources as part of the same commit unit. However, if non-IMS resources such as DB2 are involved, resource requests from different IMS transactions may cause contention.
- Encina allows transactions to be embedded within other transactions. These embedded transactions are called *nested transactions* or *subtransactions*. The OTMA interface in the AS server for IMS does not support nested transactions.
- If an application creates its own application identifier rather than using the default Encina-generated identifier, the application identifier must be 40 characters or fewer.
- Transactional IDL files can import IDL files but not other files containing TIDL constructs.

### 2.2.2.5 Encina CICS Connection through TCP-IP

The AS server coexists with all other ways of accessing CICS. The AS server and CICS are connected through the CICS external interface, which is supported by the multiregion operation (MRO) facility of CICS. Each RPC from a client program is handled as a CICS task.

A client can make RPC calls to a COBOL or C program that is part of a distributed program link (DPL) request, part of an EXEC CICS LINK local call, or uses the COMMAREA to pass input and output parameters.

**Note**

You can access application programs written in other languages, such as Assembler and PL/I, if the parameters for the other language match the COBOL data types and data alignments in the interface definition. The parameters in the target application programs must follow the same rules that apply to interlanguage calls from COBOL.

### 2.2.2.6 Mapping the COMMAREA to RPC Parameters

You have to define the RPC parameters in the same order and using the same data types as the fields in the COMMAREA. You can group the parameters in different ways:

- Use a structure to define the entire COMMAREA and then specify the structure as a single RPC parameter. You must specify the single parameter as in, out.
- Define each field in the COMMAREA as a separate RPC parameter.
- Use a structure to group some fields and define other fields separately.

You must balance the convenience of defining the entire COMMAREA as one RPC parameter with the possibility of negative performance when large amounts of data are passed between the client and the server. When you define only one RPC parameter, all fields in the COMMAREA are marshalled and unmarshalled. When you define some RPC parameters as input and some as output, only the required parameters are marshalled and unmarshalled.

**Note**

1. The output parameters must be large enough to contain the maximum length you expect to be returned. You cannot define parameters to be of varying length.
2. If the transaction program redefines the COMMAREA, the program must restore the COMMAREA before it passes results back to the client program.

#### 2.2.2.7 Encina-CICS Connection through SNA

Encina PPC distributed program link (DPL) provides a way for Encina applications to communicate with CICS applications (running on a mainframe, for example) by using a mechanism that is conceptually similar to an Encina TRPC. DPL allows Encina applications to interact with CICS DPL applications (that is, CICS applications that use the EXEC CICS LINK command) and other Encina DPL applications.

In PPC, DPL allows Encina applications to act as either the linking or the linked-to program. Conceptually, linking to a remote program is like making a TRPC to the remote program. Therefore, from an Encina point of view, the linking program is a client, and the linked-to program is a server. Figure 7 on page 25 shows how Encina and CICS systems communicate using DPL. DPL is always used within the context of a transaction. The DPL client program initializes and allocates a conversation with the DPL server. It then begins a transaction. Within the transaction, it makes a Dynamic_Program_Link (CMDPLINK) call to ship a function to the DPL server. The DPL server executes the function and then returns control to the DPL client. Note that the DPL server always has "sync on return." Therefore every DPL call that has successfully completed on a server must initiate a commit on that server.

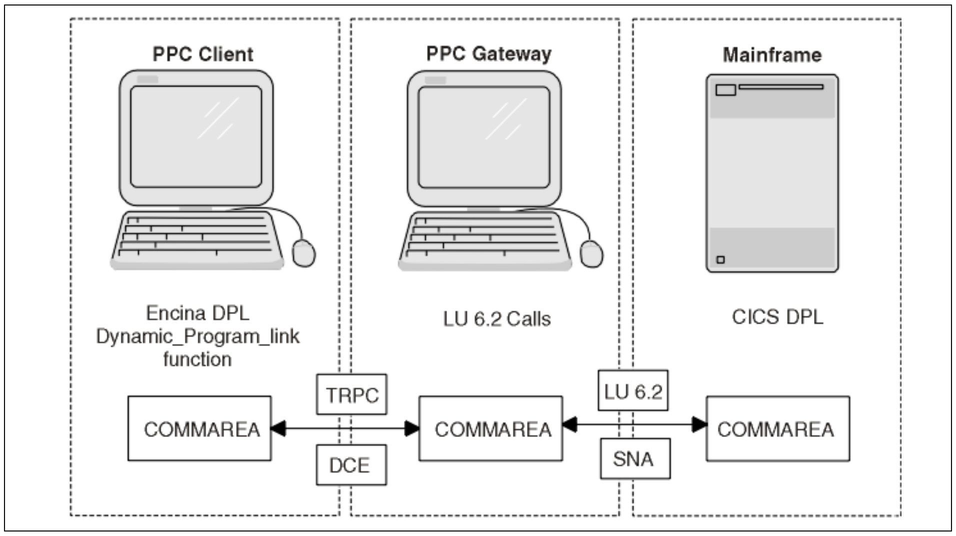


*Figure 7. DPL Access to CICS through the PPC Gateway*

The PPC gateway server is the bridge between the Encina and CICS applications, seamlesly linking the DPL client to the DPL server. Encina PPC and DCE specify the logical and physical connections between the Encina application and the PPC gateway server. SNA LU 6.2 specifies the logical and physical connections between the PPC gateway server and the mainframe. The PPC library handles the details, including syncpoint processing, conversation deallocation, and security.

Data is passed between the client and the server through the COMMAREA. The client and server must agree on the format and size of this buffer. The client specifies the format of binary or string data that is passed back and forth in the COMMAREA.

You do not have to include any special logic in the application to initialize multiple conversations within the same transaction. DPL can reuse conversations within a transaction. A PPC conversation is allocated for the first Dynamic_Program_Link call during a transaction. The conversation is reused for subsequent calls that have the same PPC allocation parameters. It is automatically deallocated when the transaction commits. Encina PPC is discussed in more detail in Chapter 3.1.6, “Encina Peer to Peer Communication” on page 44.

## 2.3 Encina Resources

Encina facilitates access to transactional data. Several data storage systems, most notably relational databases such as DB2 and queuing systems such as MQSeries also support the notion of transactions. Encina interacts with these resource managers, using the X/Open XA interface specifications.

The X/Open XA interface specifies a bidirectional interface between a transaction manager (in this case Encina) and a resource manager. The transaction manager drives the commitment of transactions and the data recovery, while the resource manager acts as a participant.

Encina provides a module called TM-XA. This module supports the XA interface by using the support of other Encina toolkit modules such as TRAN and LOG. (We explain these modules in Chapter 3.) Applications using the TM-XA Service provide the list of resource managers that use the XA interface and set the transaction context before calling the resource managers. The service handles commit coordination and recovery of the resource managers automatically from the application’s point of view. The TM-XA Service uses TRAN to implement these features.

The X/Open XA interface specification also 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.

### 2.3.1 Encina and Database Access

The X/Open XA interface specification describes a protocol by which Encina must interact with the resource manager. Most of the interfacing tasks are supported automatically by the Encina runtime environment, and the programmer does not have to be concerned with the details of the communication protocols between Encina and the resource manager.

The important step from the point of view of application development is to register the resource manager with Encina. You must register the resource manager with the Monitor environment, using the `mon_RegisterRmi` function during Monitor initialization, before the call to the `mon_InitServer` function. You must supply the following information:

- The resource manager's XA switch. This switch is provided by the relational database management system (RDBMS) vendor.
- The name of this instance of the resource manager. This name must match a name already configured with the Monitor. To register the

resource manager, the Monitor uses the information already configured about it.

The `mon_RegisterRmi` function returns an ID, which can be used in those structured query language (SQL) calls that take an interface ID. With some RDBMSs, this ID enables the application to work with multiple instances of the RDBMS.

Typically, the RDBMS library exports the XA switch. You need only declare an external variable of the appropriate name, as specified in the RDBMS documentation. For example, the DB2 library exports its XA switch under the *db2xa_switch* variable. Oracle exports its XA switch under the name **xaosw**. The linker initializes this variable; you can use it in the call to the `mon_RegisterRmi` function.

After the resource manager has been registered, you can access the RDBMS. Embedding SQL code directly in an application is the simplest and most common way to access RDBMSs programmatically.

Although the embedded SQL interface is straightforward, 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.

Applications 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.

The SQL communications area (SQLCA) provides a standard error handling mechanism. The SQL communications area 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 application developers is the sqlcode field, which contains the return code of the most recently executed SQL statement.

You include the *sqlca* structure in a program by using an embedded SQL include statement. After each SQL call, an application checks the sqlcode field to determine whether the call was successful. If the call fails, you abort the transaction.

### 2.3.2 Encina and RQS Access

The 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.

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. Enqueuing and dequeuing are performed from within the scope of a user transaction. The RQS guarantees that, once 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 because of system failures, media failures, or failed dequeue attempts. If the dequeuing 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. The RQS provides a number of field types that can be used when defining an element type.

Because the RQS uses the same underlying Encina toolkit, the toolkit preserves the transactional semantics between the RQS data and the other application data. You do not have to do any special coding to maintain the transactional data integrity.

## 2.4 Encina++

Encina++ is a set of interfaces for programming Encina applications in C++ or Java. Encina++ provides an object-oriented model for the development of client and server application programs in a distributed transaction processing environment.

Encina++ supports the development of object-oriented applications that are based on the DCE, CORBA, or both.

Encina++ contains several different APIs. Some of these APIs are common to both DCE and CORBA, and some are designed for only one or the other. The applications you write can use some or all of these interfaces; which interfaces you use depend on the requirements of a particular application.

The Encina++ programming interfaces common to both DCE and CORBA provide client and server support and transaction demarcation capabilities.

The Encina C++ interface defines C++ classes and member functions that enable the creation and management of client/server applications and provide support for the underlying environment.

The Transactional-C++ (Tran-C++) interface defines C++ constructs and macros as well as classes and member functions for distributed transaction processing. This interface provides an object-oriented alternative to the Encina Transactional-C (Tran-C) interface.

The OMG Object Transaction Service (OMG OTS) interface also defines C++ classes and member functions for distributed transactional processing. This interface implements the OMG OTS specification.

Although the underlying functionality is the same, there are two implementations of the common interfaces. These implementations, referred to as Encina++/DCE and Encina++/CORBA, address the differences in DCE and CORBA. Important differences include client and server stub generation and binding methods.

To write Encina++/DCE applications, you must use Encina’s TIDL compiler to generate stub files for communications between Encina++ clients and servers, adding transactional semantics to remote procedures. Using TIDL, you define which functions in the interface are transactional. To write Encina++/CORBA applications, however, you use a CORBA IDL compiler specific to the ORB being used. Using the CORBA IDL, you define entire interfaces rather than individual functions as transactional.

Encina++/DCE clients bind to exported server objects by using constructors defined in the client stubs generated by the TIDL compiler. In Encina++/CORBA applications, however, clients use a binding method that is specific to the ORB being used (or clients can use a CORBA Object Naming Service, if available).

The Encina++ programming interfaces that are supported only for DCE provide object-oriented access to two types of Encina servers offering specialized services:

- The Recoverable Queuing Service C++ interface (RQS++) defines C++ classes and functions for enqueuing and dequeuing data transactionally.
- The Structured File Server C++ interface (SFS++) defines C++ classes and functions for manipulating data stored in record-oriented files while maintaining transactional integrity.

In addition, you can use Encina's data definition language (DDL) compiler to generate stub files for RQS++ and SFS++ applications.

Encina++ provides two programming interfaces that are supported only for CORBA: one defines a locking mechanism for CORBA-based servers, and the other allows you to develop Java transactional clients that can communicate with Encina++ servers:

- The OCCS interface defines C++ classes and functions that enable multiple clients to coordinate access to shared resources. This interface implements the OMG Concurrency Control Service Proposal.
- The Java OTS client interface defines Java classes and functions that enable Java client applications to begin and control distributed transactions. This interface implements the OMG OTS specification.

Encina++ offers the following features for object-oriented, distributed transaction processing applications:

- Initialization of clients and servers
- Transparent and explicit binding

- Object registration and binding
- Integration of XA-compliant databases
- Transactional and nontransactional threads
- Integrated exception handling

Encina++ enables you to develop several different types of client and server applications in C++ as well as Java clients that access Encina++ servers.

# Chapter 3. Encina Components

Encina is a family of products for developing, executing, and administering distributed transaction processing systems. A distributed system consists of multiple software components that run in separate, independent processes on different machines in a network. For example, a distributed operation can require applications running on UNIX workstations and Windows NT machines and data residing in a database on a mainframe. Even though the software is distributed across a network, it can be accessed reliably by multiple users as if it were running on a single system. The Encina family consists of base components, object components, Web components, and client components. In this chapter we take a close look at these different components. In subsequent chapters we describe in more detail various Encina components as they are used to develop a transactional client/server application.

## 3.1 Base Components

In this section we provide a brief overview of Encina's base components.

### 3.1.1 Encina Toolkit

The Encina Toolkit provides low-level services required for distributed transaction processing systems. The Toolkit services implement a complete transaction paradigm: nested, distributed, concurrent transactions with recoverable storage. These transactions can be used to maintain the consistency of data on a network in the face of communication failures, system failures, and disk failures.

The Toolkit services provide the foundation on top of which Encina's extended services are built. These extended services include higher-level facilities (such as the Encina Monitor) that expand the Toolkit functionality to provide a comprehensive environment for developing distributed transaction processing applications.

The Encina Toolkit comprises several modules, implemented as function libraries. Each module provides a different service. The Toolkit libraries provide all of the functions required for transaction processing system development. The modules of the Encina Toolkit are grouped into two major components:

- Toolkit Executive. The Executive provides services that permit a process to initiate, participate in, and commit distributed transactions. These services include transactional extensions to DCE RPCs that ensure transactional integrity over distributed computations transparently. The Executive also supports nested transactions, a feature that provides failure containment and simplifies application development.

- Toolkit Server Core. Built on the Executive, the Server Core provides facilities for managing recoverable data, that is, data that is accessed and updated transactional. These facilities include a locking library to serialize data access, a recoverable storage system to allow transactions to roll back or roll forward after failures, and an X/Open XA interface to permit the use of XA-compliant resource managers.

The modules that make up the Toolkit Executive provide the functionality necessary to write client applications. Client applications start transactions and make transactional RPCs to Encina server applications. The primary low-level modules of the Executive include Distributed Transaction Service (TRAN), Transactional Remote Procedure Call (TRPC), and Thread-to-Tid Mapping Service (ThreadTid). TRAN coordinates multiple transactions, guarantees that transactions either commit or abort, and manages the delivery of information about transaction outcome to the participants of the transaction. TRPC carries additional information to identify the transaction on whose behalf it is executing. A TRPC is similar to a standard RPC and is discussed in detail in Chapter 3.1.3, “Transactional Remote Procedure Calls Service” on page 37. ThreadTid maintains the association between a thread and a transaction identifier (TID). It allows applications to determine which transaction is associated with a particular thread.

The easiest way to build a Toolkit client application is to use Tran-C, a C-language programming interface designed to simplify the development of Encina transactional applications. Tran-C defines functions and constructs that provide high-level interfaces to functionality that is commonly used in transactional programming. The use of low-level Toolkit modules is not usually necessary, as Tran-C provides most of the functionality offered by the low-level components of the Toolkit Executive. There is little performance penalty for using Tran-C in preference to using the low-level modules directly.

In some cases, however, aspects of the high-level interface provided by Tran-C can conflict with the requirements of an application. For example, Tran-C's exception handling facility can cause a conflict if another language (such as C++) that has its own exception handling facility is used in the same application. Under such circumstances, you might decide to use the low-level modules directly instead of using Tran-C.

Using the lower-level interfaces (such as TRAN) to begin and end transactions provides increased flexibility but can result in programs that are longer, less readable, and more difficult to maintain than Tran-C.

You can also use the lower-level interfaces of the Toolkit in conjunction with Tran-C if you use them with caution. In particular, you must be very careful when calling low-level Executive interfaces that perform activities normally performed by Tran-C. For example, if you begin a transaction with Tran-C’s transaction statement and attempt to commit it by calling the TRAN tran_End function, the resulting behavior is undefined.

Situations in which you might want to use lower-level interfaces from within a Tran-C application include requesting callbacks for specific states in the execution of a transaction and setting application-specific properties. These situations are fairly rare; in most cases, if you are developing client applications only, you do not need to use the interfaces of the low-level components.

The modules that make up the Encina Server Core, in combination with the modules of the Executive, provide the functionality necessary to write recoverable server applications that manage persistent data and accept TRPCs to access that data. The primary low-level modules of the Server Core used to write server applications include Recovery Service (REC), Log Service (LOG), Volume Service (VOL), and Lock Service (LOCK). The Server Core also includes the Transaction Manager-XA Service (TM-XA), which you use to access and use XA-compliant resource managers in your transactions. The REC Service guarantees the consistency of the permanent data used by a distributed transaction service. It uses log records to undo or re-create transactions. The LOG Service provides efficient and stable storage for recording the actions of programs as they update recoverable data. It ensures that accurate records of transactions are retained across system shutdowns and restarts.The VOL Service provides a logical interface to underlying physical storage. It enables volumes and files to span multiple physical devices. The LOCK Service permits synchronization of accesses to data. It enables transactions to lock resources before accessing or modifying them. The TM-XA Service implements the transaction manager side of the X/Open XA interface for coordinating distributed transactions with XA-compliant resource managers.

The easiest way to build a Toolkit server application is to use Tran-C, REC, and the high-level interface to LOCK provided by Tran-C. However, it is important to remember that most of the underlying modules used by higher-level Toolkit modules must still be initialized explicitly. For example, although REC performs most of the calls to VOL interface functions required by server applications, the Toolkit server application must still initialize VOL.

TM-XA allows XA-compliant databases (resource managers) to be accessed from within Encina applications. In most cases, developers who must integrate an application with external databases build an intermediary server that allows multiple clients to access one or more external databases through TM-XA. Although client applications can use TM-XA directly, TM-XA must be used in conjunction with LOG because the client of an XA-compliant database must be recoverable. This restriction makes it impractical for end-user applications to use XA-compliant databases directly because each end-user process would require its own log file.

An alternative to writing a Toolkit server is to write a Monitor application server, using the Encina Monitor and Tran-C. The Monitor's high-level interfaces allow you to write server applications without needing to know all the low-level details of the Toolkit. In addition, the Monitor provides facilities for load balancing, scheduling, and so on, and it integrates support for XA-compliant resource managers.

### 3.1.2 Encina Monitor

The Encina Monitor provides the means to develop, run, and administer transaction processing applications. Monitor applications, which are distributed client/server applications, use the Monitor API. The Monitor, in conjunction with resource managers, provides an environment in which to maintain large quantities of data in a consistent state. It controls which users and clients access specific data through defined servers in specific ways. The Monitor provides an open, modular system that is scalable and interoperates with existing computing resources such as IBM mainframes running CICS. It supports interoperation among a number of components: the operating system, the OSF DCE, the Encina Toolkit, third-party RDBMSs such as Informix and Oracle, third-party front ends (user interfaces) such as JYACC's JAM, and networks.

The Monitor provides three functional areas for a transaction processing system:

- The run-time environment
- The system administration facility
- The application development environment

The Monitor run-time environment coordinates transaction processing (TP) applications and resource managers and performs run-time administration tasks, such as load balancing and collecting diagnostics. In addition, it provides for other interactions with the execution environment, such as scheduling calls for later execution and retrieving information about users, transactions, and client/server bindings.

The Monitor system administration interface is used to construct, initiate, control, and terminate a Monitor system. The Monitor is administered through Monitor administrative and configuration interfaces.

Monitor applications are developed using the Monitor API in conjunction with other Encina interfaces, such as Tran-C. The Monitor saves the programmer effort by performing some tasks, such as interacting with DCE RPC and security, on the application's behalf.

Clients can use screen- or form-development tools to develop user interfaces. The Monitor does not provide these tools but supports their integration. Servers can be developed using Tran-C or other transactional interfaces such as that provided by the Monitor API. The Monitor API provides functionality to manage client and server application programs in a distributed transaction processing environment.

### 3.1.3 Transactional Remote Procedure Calls Service

The Encina TRPC service enhances the DCE RPC package offered by the OSF. DCE RPC is a remote procedure call system that implements nontransactional RPCs for use by Encina Toolkit components.

TRPC provides the same basic interface as DCE RPC. However, there is one significant difference. TRPC implements transactional and nontransactional RPCs; DCE RPC implements nontransactional RPCs only. To accommodate the transactional nature of TRPC, the TIDL was developed. TIDL simplifies the writing of code using TRPCs. It is an extension of the DCE IDL. The TIDL compiler is named *tidl*.

TRPC assumes a multithreaded environment. A thread package is a prerequisite for DCE RPC, the underlying communication paradigm for TRPC.

An RPC is a programming paradigm similar to the well-known procedure call mechanism. Both transfer control and data within a program. When a remote procedure is called, the parameters of the call are passed over the network to the environment where the call is actually executed. Meanwhile, the calling environment waits for the results of the procedure execution. Typically the calling environment is a program that is referred to as a client. The environment where the call is executed is referred to as a server. When the server (the called environment) finishes executing the procedure, it ships the results back to the client (the calling environment), which then resumes execution as if returning from a local procedure call.

TRPCs are initiated from within the scope of a transaction. Each TRPC does work on behalf of a transaction. The TIDL preprocessor adds additional parameters to user-specified operations in an interface definition file. These additional parameters are used to carry the TRAN state and data. TRPCs carry the TRAN state and data along with the regular parameters of the RPC and pass the parameters to the appropriate TRAN.

Although transactional applications commonly use TRPCs, some transactional applications may also use nontransactional RPCs. Therefore, TRPC provides a mechanism that allows an RPC to retain its nontransactional semantics. TRPCs nontransactional RPCs are an enhanced version of the DCE RPC.

TRPC consists of two components: a preprocessor and a library of functions. The TIDL preprocessor, tidl, preprocesses interface definition files. It produces a set of files that must be compiled and linked appropriately with client and the server programs. It also produces an interface definition file that must be processed by IDL.

The library of functions provides the relevant communication support for the TRAN. TRPC also exports interface calls to the application developer. The functions can be categorized as follows:

- Functions to support the TRAN communication interface
- Functions used to initialize the TRPC run-time interface, provide information about communication protocols and end points, and register callbacks
- Functions that wrap certain DCE RPC run-time functions that manipulate RPC handles

In Section 4.3, “TRPC” on page 66 we discuss the basics of how to program TRPC.

#### 3.1.3.1 Encina Threading Model

A single server program typically services requests made by multiple client programs. To expedite the handling of multiple requests, most modern client/server development environments provide a mechanism for intraprocess multitasking. Encina supports the use of multiple parallel execution sequences, called *threads*, within the single address space of a process. A thread, also known as a lightweight process, is an execution environment within an address space. Threads that belong to the same parent process normally do not interfere with each other. In most cases, it is not necessary to protect portions of a process's address space from various threads of that process.

Within a threaded environment, interthread communication through shared data structures is so simple that access to some portions of shared memory must be restricted. Mutual exclusion facilities (mutexes) are used to restrict access. A mutex is a synchronization object used to ensure that only one thread can execute in a particular section of code or access a particular portion of memory at a single time. A mutex has essentially two states: locked (no other thread can execute the particular piece of code or access the agreed upon memory) and unlocked (access is available to any thread). Mutexes can be used to prevent incompatible accesses from occurring, such as when one thread is reading some data while another thread is modifying it.

A transactional environment has many different types of access to shared memory, not all of which need to be mutually exclusive. For example, multiple threads can require read access to a portion of shared memory. Although it would be bad if some other thread modified the memory, all threads wanting simply to be able to read the data are compatible with each other. In this situation, mutexes fail to provide an appropriate mutual-exclusion mechanism. To overcome this shortcoming, the Encina Toolkit provides a locking mechanism that provides multiple locking modes for portions of shared memory. The Tran-C interface provides transactional mutexes and locks for use in situations where finer control of access to shared memory is required.

The Encina threading model is based on the DCE thread package, which supports the Posix pthread specification. DCE supports multiple threads within a process, allowing a client application to concurrently issue several RPC calls, each of which executes within a single thread of control. Similarly a server process can be a multithreaded process, allowing the server to concurrently process multiple client requests. Programs must be carefully written in a multithreaded environment, with appropriate mutexes to protect shared variables.

An Encina server may consist of multiple threads. The number of threads within a processing agent can be configured administratively. An Encina server may be linked to other modules, such as a database. In this case, depending on the ability of the database to support or not support multiple threads, the interface between the Encina server and the database is configured appropriately by the Encina administrator. Depending on the configuration, Encina appropriately manages its threads by allowing or disallowing them to concurrently access the database.

### 3.1.4 Encina SFS

The Encina SFS is a record-oriented file system that provides transactional integrity, log-based recovery, and broad scalability. Many operating systems support only byte stream access to data where all input and output data, regardless of its source, is treated as an unformatted stream of bytes. SFS uses structured files, which are composed of records. The records themselves are made up of fields, and the field layout is defined when the file is created. The SFS file system is based on X/Open ISAM standards. SFS is ideally suited for applications that manage large amounts of record-based data--for example, inventory records, customer orders, and employee files. In a typical model, an SFS server application receives requests from one or more SFS clients for access or modification to data. The way the records in a file are arranged is referred to as the file organization. The records in a file can be organized in one of three ways:

- Entry-sequenced
- Relative
- B-tree clustered

The records in an entry-sequenced file are stored in the order in which they are written to the file. New records are always appended to the end of the file. When records are deleted from an entry-sequenced file, the space formerly allocated to those records is not automatically reclaimed or reused. The only way to reclaim this space is by using the `sfs_ReorganizeFile` function. Entry-sequenced files can contain fixed-length or variable-length records. An updated record must be no longer than the original record.

A relative file is an array of fixed-length slots. Records can be inserted in the first free slot found from the beginning of the file, at the end of the file, or in a specified slot in the file. Relative files are often used when records will be accessed directly, by record number. Because all of the slots in a relative file are the same size, SFS can calculate the position of a specific record, identified by record number, by multiplying the record number by the record slot size. The records in a relative file can consist of fixed or variable-length fields, but the size of each slot in the file is that of the maximum record size (which is calculated from the record specification when a relative file is created). The records in a relative file can be updated or deleted in place. Any slots freed when records are deleted can be reused by subsequent insertions.

A B-tree clustered file is a tree-structured file where records with adjacent index values are clustered together to reduce the cost of searching for ranges of records. The clustered file organization used in clustered SFS files is automatically maintained by the SFS server. The records in a clustered file are ordered on the basis of the contents of the primary index. Because the SFS may move records to maintain clustering when new records are inserted or deleted, there is no practical way to maintain direct references to individual records. The records in a clustered file can be fixed or variable in length. Records in clustered files can be updated or deleted. Disk space freed by record deletions is automatically reused.

An SFS program is a client to an SFS server and data is accessed by making calls to an SFS server. An SFS client must take the following actions to process records in a file:

1. **Initialize:** SFS is automatically initialized when the program calls an SFS function for the first time.
2. **Open files:** Any file that you are going to use must first be opened. When a program opens a file, it specifies the name of the file that it wants to access and how it wants to access it.
3. **Processing operations:** Programs have several options for handling records. SFS can read or write records as a single buffer; the program is then responsible for packing the field into or unpacking the fields from that buffer. Alternatively, SFS can place some or all of the fields into their own buffers. Programs also have the option to read or update only some fields in a record. Records can be accessed randomly or sequentially. The server maintains the transactional integrity of the data and provides transactional access to the data that supports fine granularity of access and top level and nested transactions.
4. **Close files:** A program should close any files after it has finished using them, either explicitly or automatically at the end of a transaction.

Storing data in SFS files provides the following advantages:

- Transaction protection: SFS provides both transactional and nontransactional access to data stored in an SFS file. SFS uses the services of the Encina Toolkit to recover from server problems, network outages, and media failures. With transactional access, the state of the SFS file after recovery reflects data changes from all committed transactions. Any transactions in progress at the time of the failure are either completed or undone.

- Record-oriented files: SFS files can be created with entry-sequenced, relative, or clustered organizations. Files can be accessed through primary or secondary indexes.
- Flexible storage management: An SFS file's data and its indexes can reside on different volumes, and thus on different disks. This independence gives you greater flexibility in controlling availability and performance.
- Import/export capability: SFS files can be stored and retrieved from a file, disk, or tape device and can be transferred between SFS servers.

SFS is a non-hierarchical file system and its files are independent of the operating system file system. SFS files can be accessed through sfsadmin commands, and applications can access files through SFS functions. SFS is simple and fast, but it is limited in flexibility when compared to RDBMSs. See the *Encina Administration Guide Volume 2* for a detailed discussion of SFS administration and file management.

### 3.1.5 Encina Recoverable Queuing Service

Encina RQS is layered on top of the basic Toolkit Executive and Server Core components. RQS enables applications to transactionally enqueue and dequeue data. You can develop applications that transactionally update data in a resource manager like a database and enqueue or dequeue data from a queue, with the guarantee that both operations will either succeed or abort.

One advantage of the queuing model is that applications can off-load some work to be done at a later time. This deferred mode of computing is in contrast with the RPC style of communication where an application invokes a service to do the processing as soon as it can. This model is particularly advantageous when work can be deferred to be processed during offpeak hours. The queuing model is also natural for work flow applications, where the work flow can be modeled by queuing and dequeuing on a queue.

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 an FIFO manner. Encina supports queues that may contain elements of different data types. An element key is a sequence of one or more fields of an element type used to retrieve the element.

An RQS server tracks a variety of statistics on queue activity, such as processed and new element count, mean waiting time, and physical queue size, for a collection period and for the lifetime of the queue. It can also track the work of a queue. The work is the volume of business represented by a queue, for example, the total cost of a product or the number of ordered books.

An application can requeue an element to another queue for subsequent processing by another application. Requeuing is the process of moving an element from one queue to another. A client application can request that an element be requeued to a related queue, for example, moved from an "order" queue to a "shipping" queue. When an application dequeues an element it indicates its intent to requeue that element by identifying it as an orphan. An orphan is an element that has been dequeued but not yet requeued. Orphan is usually a transitory state; elements do not remain orphans beyond the scope of a primary transaction.

Each queue is maintained by one and only one RQS server. All interactions with that queue are handled by the server. An RQS server may contain multiple queues and requests to access the data are sent to the server, which processes the request. RQS provides support for both programmatic access and administrative tasks.

Applications that select from several different queues when processing dequeue requests can use queue sets to simplify the selection process. A queue set is a collection of queues. A queue can belong to more than one queue set. A queue that belongs to a queue set can be accessed either as part of that queue set or individually.

#### 3.1.5.1 Recoverable Queuing Service Locking

Locking guarantees the consistency of elements and queues in RQS. RQS supports locking for the duration of an operation or for the duration of a transaction.

Queues are traditionally FIFO data structures where the head and tail of the queue are locked during enqueue and dequeue operations. This is known as *strong FIFO behavior*. Strong FIFO behavior can severely limit concurrency in a distributed transactional system. Suppose that every transaction that wanted to dequeue from a particular queue had to wait while a concurrently dequeuing transaction performed its work and committed. Such a system would provide minimal concurrency. To maximize concurrency, RQS modifies the semantics of FIFO behavior during enqueue and dequeue operations. The head and tail of a queue are not locked during these operations. RQS servers dequeue and enqueue elements from queues according to their ability to obtain locks on those elements.

**Note**

One consequence of locking is that elements are not necessarily processed in strict FIFO order. Consider two transactions that concurrently dequeue from a queue. Each obtains a lock on the first element that it can lock in the queue, rather than on the head of the queue itself. Suppose that one transaction obtains a write lock on the first element, and a second transaction obtains a write lock on the second element. If the first dequeuing transaction aborts, the second transaction effectively dequeues an element that was not at the head of the queue. If the second transaction subsequently commits, it has processed an element in the queue out of FIFO order because the first element remains in the queue while the second has been dequeued.

You can read more about locking in the *Encina RQS Programming Guide*.

### 3.1.6 Encina Peer to Peer Communication

Encina PPC Services provide the ability to transactionally access data stored on mainframes and Perform a distributed two-phase commit of data stored across UNIX servers and mainframes. Thus mainframe applications can participate in an Encina transactional application, and Encina applications can participate in mainframe transactional applications. The Encina PPC Services supports a two-phase commit sync protocol (sync level 2) to commit the transaction that accesses data on the mainframe and the UNIX server.

The PPC Services use the SNA LU 6.2 communication interface for communication and support bidirectional communication. The PPC Services provide functionality to bridge systems that run different network protocols (TCP on one system and SNA on the other system). The PPC Services support both the X/Open Common Programming Interface Communications (CPI-C) and the IBM Systems Application Architecture (SAA) CPI-C. The PPC Services also support the SAA Common Programming Interface Resource Recovery (CPI-RR).

The PPC Services are implemented as a PPC Executive and a PPC gateway product. The PPC Executive is a library that is run within the Encina cell. The PPC gateway is a server that acts as a gateway between the DCE and SNA communications protocols and allows Encina applications to communicate with LU 6.2 applications.

Figure 8 on page 45 shows a typical configuration of the PPC Services where an Encina PPC Executive application runs in a DCE cell and communicates with a PPC gateway server running on the same DCE cell. The PPC gateway server communicates with the mainframe through a SNA LU 6.2 connection program. The PPC Services enable you to develop Encina applications that act as either the coordinator or the subordinate in a transaction between an Encina system and a mainframe host. Encina application programmers use the CPI-C API for coding the PPC component. The PPC gateway translates the CPI-C conversations from TCP/IP to LU 6.2.

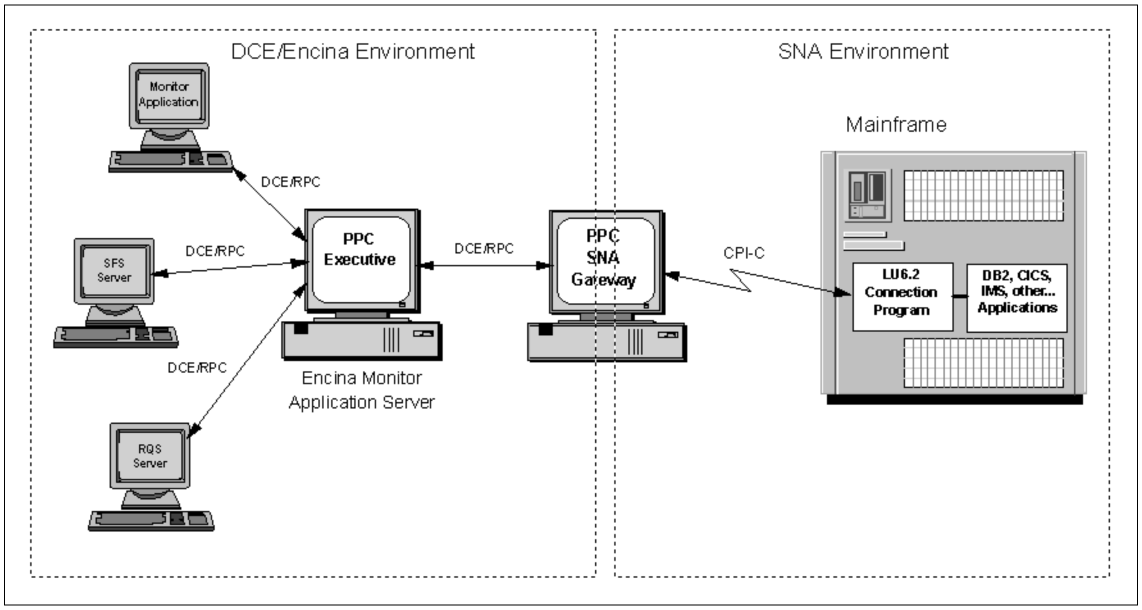


*Figure 8. Encina PPC Services*

The two-way communication between two application programs over an LU 6.2 session (connection between two LUs) is called a *conversation*. The two programs are partners in a conversation and exchange information. Each LU 6.2 session can carry one conversation at a time. To establish a conversation, one program allocates it. Therefore the program specifies the LU, the mode, and the transaction program with which it wants to communicate. The program that allocates the conversation is called the *allocator*. The *acceptor* is the recipient of an allocator’s conversation request and accepts the conversation. To end a conversation, one side deallocates it, and its counterpart receives notification of the deallocation.

Conversations involving PPC Executive applications are classified by the conversation allocator and acceptor. The PPC uses following three conversation types:

- The **Encina-to-SNA** conversation is allocated by a PPC Executive application and accepted by an LU 6.2 application running on a mainframe. The PPC Executive application requests that a gateway server allocate the conversation. The gateway server allocates the conversation on behalf of the PPC Executive application.
- The **SNA-to-Encina** conversation is allocated by an LU 6.2 application at a mainframe to a PPC Executive application through a gateway server. The LU 6.2 application at the mainframe allocates a conversation to a gateway server. The gateway server forwards the conversation to the correct PPC Executive application.
- The **Encina-to-Encina** conversation is between two PPC Executive applications. Encina-to-Encina conversations do not use the gateway server.

A PPC Executive application that allocates a conversation must know the identity of the desired acceptor. An application's conversation partner is usually defined in the application's side information file. The information in a side information file is accessed by a symbolic destination name.

Figure 9 on page 47 summarizes what happens when a PPC Executive application allocates a conversation.

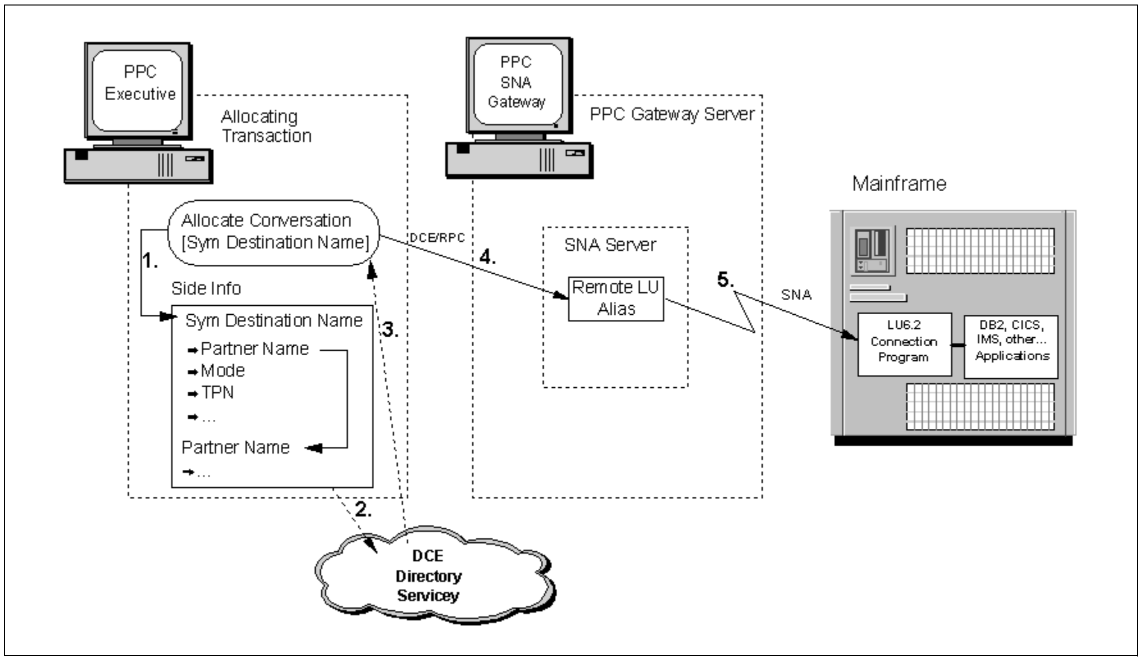


*Figure 9. Encina-to-SNA Conversation*

1. When a PPC Executive application wants to allocate a conversation to a SNA host, it first uses the symbolic destination name to determine information about the peer to which it wants to connect.
2. The application must perform a DCE Directory Service lookup to locate the remote LU alias (a name assigned to an LU in the SNA Server package on the gateway server) for that SNA host.
3. If the lookup is successful, the DCE Directory Service returns the end point of a gateway server that services the SNA host.
4. A PPC Executive conversation is allocated to the appropriate gateway server.
5. An LU 6.2 conversation is allocated from the gateway server to the host LU.

For Encina-to-SNA conversations, the remote LU alias for a SNA host and the set of gateway servers that offer connectivity to it are stored in the DCE Directory Service. The SNA connection between a gateway server and the SNA host containing the remote LU is configured in the underlying SNA Server package before the gateway server is started. When the gateway server is started or configured, it automatically registers its remote LUs in the DCE Directory Service.

**Note**

Because of the PPC gateway server's role, the gateway machine must be physically secure. Applications that can directly access a SNA link are implicitly trusted by the SNA host. If the gateway machine is physically insecure, it is possible for an untrusted user to start a process on the gateway server machine, which could then directly access a SNA link.

## 3.2 Encina++

Encina++ is a set of interfaces for programming Encina applications in C++ or Java. Encina++ provides an object-oriented model for the development of client and server application programs in a distributed transaction processing environment. Encina++ simplifies application development by providing high-level interfaces to the Encina Toolkit development tools: the Encina Monitor, the Encina RQS, and the Encina SFS.

Encina++ supports the development of object-oriented applications that are based on the DCE, CORBA, or both. Encina++ contains several different APIs. Some of these APIs are common to both DCE and CORBA, and some are designed for only one or the other. The applications you write can use some or all of these interfaces; which interfaces you use depends on the requirements of a particular application.

### 3.2.1 Encina++ Programming Model

The Encina++ classes support a client/object programming model in which clients access objects instead of servers. Servers export one or more interfaces (classes) and one or more instances of each class (objects). The client application can access objects exported by servers without you knowing how the objects available in the system map to servers.

Clients can bind to objects exported by servers. They can bind to individual objects when the objects are known, or they can bind to a class when the objects are not known or when all objects of a specific class provide the same capabilities. Typically, you specify a name for an object. Although each object created has a universal unique identifier (UUID), naming an object allows clients to bind to the object by name instead of by UUID.

In Encina++, an IDL is used to specify the interfaces to objects in the form of remote procedures. The remote procedures are used for communications between the client and server applications. The interface compiler generates files that include client stub and server stub classes for each interface. These stub classes give the client and server a slightly different view of the same interface.

Before RPCs can be made between a client and server, the server must be available to receive requests from clients. Creating an instance of the server stub class within a running server causes the object to be exported to the namespace so that a client can locate and bind to it. The instance is referred to as a *server object*.

### 3.2.2 Encina++/DCE Programming

Encina++ /DCE supports the development of transactional, object-oriented applications for DCE. Encina++ /DCE applications use the DCE RPC mechanism for communications between clients and servers. This dependency on DCE RPCs affects interface definition, binding, and exception handling.

In the DCE, the TIDL must be used to specify object interfaces in the form of remote procedures. The TIDL compiler generates C++ stubs that include client stub and server stub classes for each interface.

### 3.2.3 Encina++/CORBA Programming

Encina++ /CORBA supports the development of transactional, object-oriented applications for the CORBA environment. Encina++ /CORBA applications rely on an ORB for communication between clients and servers. This dependency on an ORB affects interface definition, binding, and exception handling.

In the CORBA environment, the CORBA IDL must be used to specify the interfaces to objects. The operations defined by an object's interface are used for communication between the client and server applications. The CORBA IDL compiler generates stub files that include client stub and server stub classes for each interface.

An ORB-specific binding method can be used to bind the client to the server. For Orbix, the IDL compiler generates a client stub class that corresponds to the interface definition. The generated class contains a static member function named *_bind*: Calling the _bind function creates an instance that is bound to an object at the server. This instance of the client stub class is referred to as a *client proxy* object.

When the binding function call is made on the client proxy object, the object is bound to a corresponding remote object, referred to as a *server object*. The client then communicates with the server object through the client proxy object.

### 3.2.4 Encina SFS++

The SFS++ interface consists of a set of C++ classes for creating Encina SFS applications. Together with other parts of Encina++ such as Tran-C++, SFS++ enables you to develop object-oriented Encina applications. The SFS++ classes contain functions that invoke the most commonly used features of SFS.

SFS++ encapsulates SFS features into a set of classes.

The DDL provides a means for defining the data objects that are used by SFS++ (and RQS++) to represent elements, records, and keys. Each type of record or element is specified as an interface in a DDL file. Keys can be specified for the data types.

To use DDL, you define the data objects you need for your SFS++ programs in a DDL file. The DDL file is then processed by the ddl command. This command generates header and source files containing C++ classes based on the data objects specified in the DDL file.

SFS++ applications use objects of the generated classes as records for file input and output. The same classes can also be used for creating SFS files. Objects of the key classes generated by DDL can be used to access records in SFS files and to add secondary indexes to SFS files.

Each SFS file can store records of only one record type. The record type defines the data type for each field of the record. SFS++ applications use DDL to define SFS record types. From these definitions, the ddl command then generates a record class, derived from the Pos::Object class. The record class represents the record type and contains constructors to create and initialize record objects of that class.

### 3.2.5 Encina RQS++

The RQS++ interface consists of a set of C++ classes for creating Encina RQS applications. Together with other parts of Encina++ such as Tran-C++, it enables you to develop object-oriented RQS applications.

RQS++ encapsulates RQS features into a set of classes.

The Rqs::Server class is an abstraction of an RQS server. It provides methods for creating and deleting element types, queues, and queue sets. Before performing any other RQS++ operations, an application must create an Rqs::Server object by using the class constructor. The application must specify the name of an actual running RQS server when it creates the Rqs::Server object.

After creating the Rqs::Server object, the application can then use Rqs::Server class member functions to create Rqs::Queue and Rqs::QueueSet objects.

The Rqs::Queue class is an abstraction of an RQS queue. It provides member functions for enqueuing and dequeuing elements, controlling access to a queue, and getting cursors for sequentially scanning elements in a queue. Before an application can enqueue, dequeue, or requeue elements to a queue, it must create an object of the Rqs::Queue.

RQS++ applications use objects of the classes generated by DDL as the elements that are enqueued, dequeued, and requeued. The generated classes can also be used to define element types at the RQS server.

## 3.3 Encina DE-Light Web Components

The explosive growth of the Internet has created a need for a product that allows access to Encina applications from the World Wide Web. The increasing use of PCs has created the need for an easy way to build PC clients using GUIs to access Encina applications without using DCE on the client PC. The DCE Encina Lightweight Client (DE-Light) product addresses these two needs.

DE-Light is a set of APIs and a gateway server that enable you to extend the power of the DCE and Encina to PCs and other systems not running as DCE clients. You can use DE-Light to build clients that require less overall effort to create than standard DCE and Encina clients and still take advantage of the benefits of distributed transactions, security, load balancing, scalability, and server replication formerly restricted to full DCE and Encina clients.

DE-Light clients use simplified RPCs to communicate with a DE-Light gateway. The clients do not use DCE for communication with the gateway. The gateway translates the simplified RPC into full DCE RPC or Encina TRPC and communicates with DCE or Encina servers on behalf of the clients. The gateway also translates communications from these servers into responses for the clients, thus completing the communications loop between clients and servers.

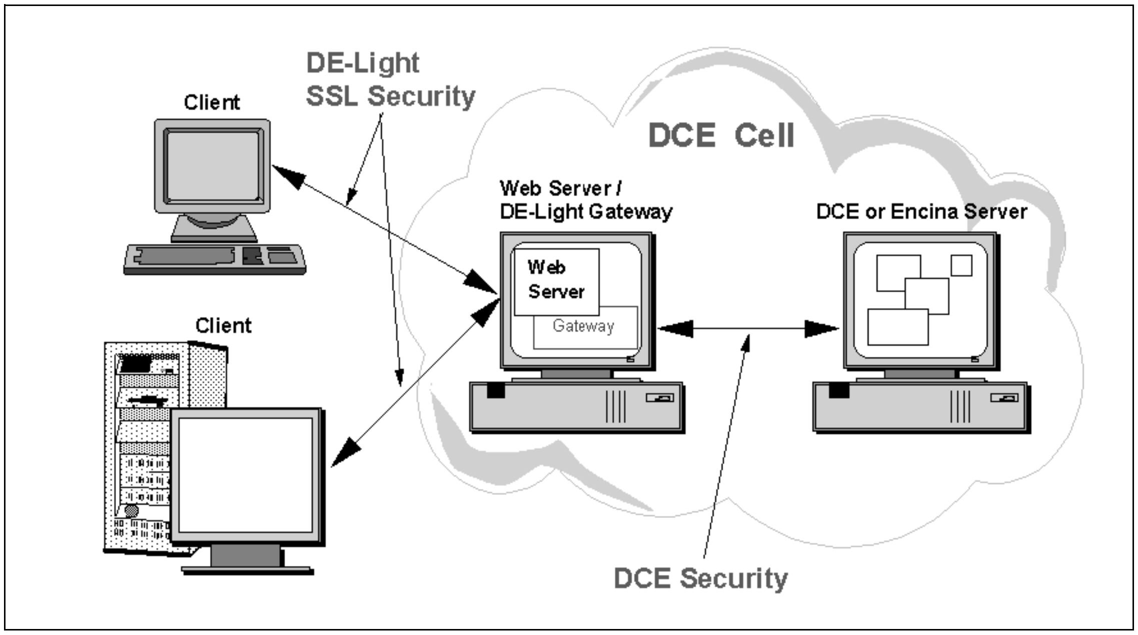


*Figure 10. DE-Light Architecture Overview*

The DE-Light package consists of three components: gateway server, C API, and Java API. The gateway server provides the RPC translation to the DE-Light clients. The two APIs provide a set of calls to be used by C and Java programs for accessing DCE and Encina servers through the simplified RPC.

## 3.3.1 DE-Light Gateway Server

A standard DCE or Encina client application uses the DCE RPC mechanism to communicate directly with DCE or Encina servers. The client application is compiled with IDL information, which defines all of the DCE RPCs or Encina TRPCs that clients understand. Each of these standard clients must run a full client implementation of DCE.

A DE-Light client accesses DCE and Encina servers through a proxy: the DE-Light gateway. The DE-Light gateway acts as a client of the DCE or Encina server. It transmits RPC requests from the DE-Light client to the servers and responses from the servers to the DE-Light client. Because the gateway acts as a DCE or Encina client, it must be loaded with information about the servers with which it communicates.

The gateway acts as an IDL interpreter. It loads IDL or TIDL files that provide information about the simplified RPCs and TRPCs clients use for communications. These RPCs and TRPCs correspond directly to the RPCs and TRPCs supported by DCE and Encina servers. Because you can load and unload RPC definitions while the gateway is running, you can reconfigure it easily to support updated or new DE-Light applications.

### 3.3.2 DE-Light C API

The DE-Light C API enables you to build clients for use in Microsoft Windows 3.1 environments that do not run DCE. These C clients are “light," that is, they require very little disk storage or memory. They are also installed easily by users, along with the client library they require for operation.

In the Microsoft Windows environment, DE-Light C clients use simplified RPCs sent through a TCP/IP connection to communicate with DE-Light gateways at known end points. These simplified RPCs are translated by the gateway into full DCE RPCs and Encina TRPCs, and they are sent along to the appropriate DCE or Encina servers. The gateway then translates the RPCs and TRPCs returning from the servers into simplified RPCs, which are sent back to the clients.

DE-Light C clients consist of client binaries, the drpc.dll library (which contains the DE-Light functions), and a drpc.ini file (which sets a number of initialization parameters, including the end point for the gateway).

Because the DE-Light C clients are intended for use within secure intranets, the security capabilities for C clients are less robust than those for Java clients. Although the connection between clients and gateways is not secure, passwords are sent to the gateway in a “scrambled” form, rather than in clear text.

### 3.3.3 DE-Light Java API

The DE-Light Java API is used to build Java applets and stand-alone Java applications that connect with DCE and Encina servers through the DE-Light gateway. DE-Light Java clients, like any Java clients, have certain advantages over C clients:

- Java clients are supported on virtually any computer.
- They can use the Internet as the transport mechanism for secure commercial applications.
- They require minimal administration. Clients download the DE-Light Java applets as required from standard Web servers.

- They can use DE-Light security, which uses the SSL, in communications with the gateway. DE-Light security is available only when you use Java applets with Web browsers that support SSL.
- They have a consistent look and feel, despite differences in computing platforms.

DE-Light Java clients can be implemented in two ways: as Java applets and as stand-alone java applications.

You can create a DE-Light Java applet embedded into a hypertext markup language (HTML) document on a Web server. With this type of client, a user contacts a Web server through a Java-enabled Web browser, and the browser automatically downloads the DE-Light Java client applet along with the HTML document (see Figure 11 on page 54).

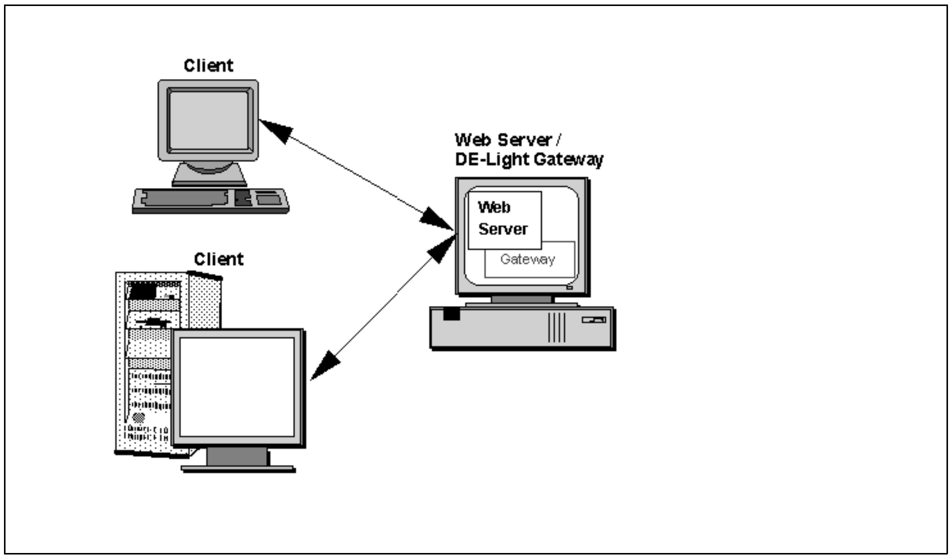


*Figure 11. Downloading DE-Light Java Applets*

The Java client applet then contacts the appropriate DE-Light gateway and performs its function. The DE-Light gateway is a separate server that must reside on the same machine as the Web server. The Web client can now communicate through the DE-Light gateway with DCE or Encina servers (see Figure 12 on page 55).

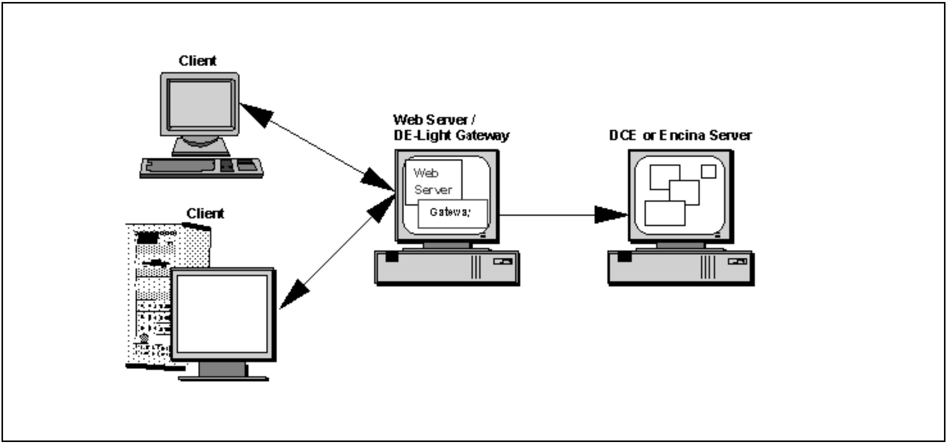


*Figure 12. Java Clients Accessing Encina Servers*

Alternatively, you can build a stand-alone Java application. This stand-alone client communicates with a DE-Light gateway at a known TCP/IP or HTTP address.

Communication between either type of client and the DE-Light gateway can use either the TCP/IP or HTTP transport protocols. The gateway then translates the simplified remote RPCs from the client into DCE RPCs or Encina TRPCs and transmits them to the appropriate DCE or Encina servers. The gateway also translates RPCs and TRPCs returning from the servers into simplified RPCs and sends them back to the client.

You can configure both types of DE-Light Java clients to use SSL connections to the DE-Light gateway server. The gateway server itself belongs to the DCE cell that contains the DCE or Encina servers and therefore uses DCE security to access those servers.

## 3.4 EncinaBuilder

EncinaBuilder streamlines Encina client application development by integrating Encina for Windows with the PowerBuilder Enterprise application development environment. By combining the power of Encina with the ease of use of PowerBuilder, organizations can more rapidly deploy Encina-based solutions throughout the enterprise.

EncinaBuilder automates the process of developing Encina clients. With EncinaBuilder, you can:

- Automatically generate Windows-based applications that access business functions offered by Encina application servers
- Use PowerBuilder's graphical, object-oriented development tools to create and customize Windows-based Encina client applications
- Rapidly deploy Encina clients by streamlining the development process

PowerBuilder applications can now scale to the enterprise level by relying on Encina's three-tier client/server architecture to support larger numbers of users and integrate a wide variety of data and resources. EncinaBuilder enables PowerBuilder users to:

- Develop and deploy large-scale client/server applications, using the Encina Monitor and other Encina products
- Transactional access and update heterogeneous data resources, including RDBMSs, record-oriented files, queues, and mainframe systems
- Capitalize on the robust, scalable execution and management environment of the Encina Monitor
- Integrate services of the OSF's DCE into PowerBuilder applications

With EncinaBuilder for Windows, Encina client applications are developed easily and rapidly with PowerBuilder Enterprise application development tools. Encina application servers make business functions available to client programmers through well-defined interfaces. The interfaces are described in a TIDL file.

Developers use EncinaBuilder's GUI to select and transform Encina application server interfaces defined in the application server's TIDL file into PowerBuilder callable objects. Once EncinaBuilder has populated the PowerBuilder library, you can develop Encina for Windows client applications, using standard features of the PowerBuilder graphical client/server development environment.

In addition to generating PowerBuilder objects, EncinaBuilder creates a fully functional PowerBuilder application that can be used immediately to invoke the functions offered by the Encina application server. These sample applications are customized with standard PowerBuilder features to create graphical Encina client applications.

EncinaBuilder provides a library containing PowerBuilder user objects for common transaction functions, including begin, commit, and rollback. The library also contains objects encapsulating Encina Monitor functions, such as setting authorization and authentication defaults, and application management functions, such as initializing and terminating applications.

# Part 2. Encina Components Related to Application Development

In Part 2 we discuss in detail the Encina transaction model and the Encina components related to application development and explain how to use and program the components and connect Encina applications to the Internet. Part 2 is particulary important for readers who are not familiar with Encina application development and some of its details. It demonstrates how all Encina modules work together, how interfaces must be defined, and how the different application modules should be coded. Reading this part will prepare you well for Part 3, where we show, through a case study, how to use the Encina components.

# Chapter 4. Encina Transaction Model

In this chapter we provide background information about transaction processing and transactional applications. We also describes programming concepts with Encina TRPCs and their security implications.

## 4.1 Transactions

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 can run across different computers, using different programs (executables) hosted by different operating systems. Although a transaction may be distributed, made up of several different programs, and working with different data sources, any client recognizes only the transaction call and not the complexity behind it. Thus is it a perfect mechanism for three-tier distributed computing models.

### 4.1.1 Atomicity Consistency Isolation Durability (ACID)

Transactions have four critical properties that have to be understood:

- Atomicity. A transaction's changes are atomic; either all operations that are part of the transaction occur or none occurs.
- Consistency. A transaction moves data between consistent states.
- Isolation. Even though transactions can execute concurrently, one transaction does not see 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 withdrawn 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 occur. 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.

A distributed transaction is one that runs in multiple processes, usually on several machines. Each process works for the 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.

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, a process 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 coordinator commits the transaction by writing a commit record to its stable storage and then informs all the participants that the transaction has committed. If any participant indicates that it cannot prepare, the transaction coordinator aborts the transaction by writing an abort record to its stable storage and then informs all participants that the transaction has aborted.

### 4.1.2 Nested Transactions

Using transactions does not always allow applications the granularity of error isolation that may be desired. If the transaction aborts, all changes are rolled back. Although this may be sufficient for a large number of transactions, for more complicated transactions, you may want a finer granularity of error isolation. For example, you may not want to undo all parts of a transaction because of an error in one operation.

In such cases you can decompose a transaction into a number of subtransactions. The failure of one transaction does not affect the other subtransactions in a transaction. If a subtransaction fails, it can be simply restarted without having to roll back or restart the other subtransactions. As an example, for the transfer application described above, the withdrawal operation and the deposit operation can be written as subtransactions within the transfer transaction. Therefore if the deposit operation fails, it can be retried without having to roll back either the entire transaction or the withdrawal operation.

Standard tree terminology is used to describe relationships between nested transactions. A nested transaction is considered an indivisible operation within its enclosing transaction. Nested transactions have the same transactional properties, although with somewhat weaker guarantees. Subtransactions can be used to achieve safe parallelism and finer-granularity failure isolation.

A transaction that is not nested within another is called a *top-level transaction*. A subtransaction is a child of the enclosing (parent) transaction. A parent may have several children; those children are siblings. The ancestor and descendant relationships are the recursive closures of the parent and child relationships. A top-level transaction and its descendants are collectively known as a *transaction family*, or simply a *family*.

Subtransactions are simply another form of operation that can be performed within a transaction. Whenever an application can perform work on behalf of a transaction, it can instead create a subtransaction to do that work. A subtransaction must be strictly nested within the enclosing transaction; it must be completed (committed or aborted) before the enclosing transaction can complete. Should the enclosing transaction abort, all effects of the subtransaction are also undone.

The transaction properties of atomicity, serializability, and permanence apply to top-level transactions; somewhat weaker properties apply to individual subtransactions. Subtransactions retain the atomicity and consistency properties. The isolation property is essentially the same: a subtransaction cannot observe the effects of another subtransaction that may subsequently abort independently of it. The permanence property applies only to top-level transactions; a subtransaction is not permanent until its top-level ancestor commits. These properties permit subtransactions to be used to achieve safe parallelism within a transaction and to isolate failures within subtransactions.

A transaction can create several child transactions, each of which performs part of the required work. The parent transaction does not perform any work until all those children complete, so it does not observe their partial effects. A child transaction can observe the previous effects of the parent transaction; however, a transaction cannot observe any effects of a sibling until that sibling completes. This isolation property permits the child transactions to operate in parallel without interfering with one another.

Nested transactions offer several features. They:

- Enable an application to isolate errors in certain operations
- Allow an application to treat several related operations as a single atomic operation
- Operate concurrently

Nested transactions, like any other transactions, incur a performance cost. Therefore, you should use them only when necessary.

A nested transaction begins 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 the nested transaction is created, it is completely independent of the transaction that created it. The Tran-C top-level 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 in Tran-C is simply to use a transaction block within the scope of an existing transaction. Tran-C automatically makes the new transaction a subtransaction of the existing transaction.

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.

Most commercial RDBMSs, the XA interface, and Encina PPC Services do not support nested transactions.

## 4.2 Rollback

If a transaction is aborted before it can be committed, the effects of the transaction must be undone or rolled back. All components of a global transaction must be rolled back in that case.

The Encina TRAN service implements a two-phase commit protocol to drive the commitment and recovery of a transaction. When a transaction begins, the event is logged in a log. Other events such as the prepare phase and the commit or abort of a transaction are also logged in the log.

Encina recovery is invoked whenever the system is restarted. For example, if a machine crashes and is restarted, Encina recovery takes place when Encina is restarted on the machine. When Encina is restarted, it goes through its log. It aborts transactions that were started but not prepared. For such transactions, Encina applies undo records from the log to undo their effects.

If a transaction accesses other recoverable data, for example, RDBMSs or data stored in a mainframe that is accessed through PPC, Encina must roll back the changes made to the data by the transaction that is being rolled back.

For RDBMSs, Encina uses the XA interface to coordinate the rollback. Encina makes xa_rollback calls to the database for each transaction that is to be rolled back. It is the responsibility of the database to roll back the data for such transactions. Only those transactions that attempted to do any XA work for that database are aborted.

For PPC, the connection between the Encina application and the mainframe application must be syncpoint sync level 2. Sync level syncpoint is the highest level of synchronization provided by LU 6.2. It provides transactional conversations, which use the two-phase commit protocol, and rollback and resynchronization capabilities.There are two ways in which an LUW can be backed out (aborted):

- Through the CPI-RR backout (SRRBACK) function
- Through an asynchronous abort. An asynchronous abort results when an application explicitly calls one of the Encina abort functions (such as abort or tran_Abort). An asynchronous abort can also be due to an underlying failure, such as a TRPC failure.

Any peer can initiate a backout by calling the backout (SRRBACK) function. All other peers receive notification of this backout. The status code received and the action the application must take in response to receiving this backout request depend on which function returned the backout notification.

An asynchronous abort occurs whenever a transaction aborts by some mechanism other than an application issuing a backout (SRRBACK) function call. For example, if an application calls tran_Abort, it may cause an asynchronous abort at any CPI-C site participating in the transaction. An asynchronous abort can also occur as a result of a failure in the underlying RPC mechanism. Asynchronous aborts are detected by the PPC Services.

If the application asynchronously aborts the transaction, all sync level syncpoint conversations are deallocated with a DEALLOCATE_ABEND type return code. All of the application's peers receive a CM_DEALLOCATED_ABEND return code (such as CM_DEALLOCATED_ABEND_SVC).

If an asynchronous abort occurs when the application is issuing a CPI-C or CPI-RR function call, that call returns a product-specific error with the detail set to PPC_ASYNC_ABORTED. It is possible that the asynchronous abort will be detected during a CPI-RR function and that the function will receive a status code making it appear as if the transaction backed out. When it appears that the transaction naturally backed out, the PPC Services begin a new transaction.

Any peer can initiate backout processing. The remote peer initiates backout processing by calling Backout (SRRBACK).The PPC Executive application can initiate a backout either in the server function or in the client (after the TRPC returns). To do so, it can call any of the Encina functions that initiate an asynchronous abort.

For a distributed transaction, Encina selects a two-phase commit coordinator for the transaction. The coordinator is responsible for logging the commit/rollback status of the transaction. Whenever a participant in a transaction performs recovery, it must contact the coordinator to obtain the status of transactions that are undecided. Only the coordinator can determine the outcome of such transactions. For example, if a back-end database fails and is restarted, it contacts the TM-XA service for the outcome of in-doubt transactions. The TX-XA service contacts the transaction coordinator for such transactions and then informs the database of the transaction outcome. The database must then roll back the effects of aborted transactions.

In general, PPC transactions are distributed transactions: one application starts the transaction, which is spread to other applications, on other hosts. When a transaction is distributed over multiple hosts, one participant in the transaction must be recoverable to allow the transaction to commit. This is because the Encina DTS requires the coordinator of a transaction to be recoverable. If the PPC Executive application is using the PPC SNA/Gateway (that is, if it is an Encina-to-SNA or SNA-to-Encina conversation, not an Encina-to-Encina conversation), the gateway server can act as the coordinator because it is recoverable.

If no application involved in the transaction is recoverable, the transaction is be aborted. Note that this occurs only when the transaction is distributed across multiple hosts. If the application has local-only transaction processing, the Encina DTS arranges for the transaction’s outcome status to be committed.

To make the server transactional you must:

1. Make the server recoverable.
2. Modify the initialization steps so that your server initializes the parts of Encina needed for transactions. Depending on the component you use, you may also have to modify the way the server terminates.
3. Delimit (indicate the beginning and end of) the transaction, specifying which operations are to be part of the transaction.

There is a bootstrap problem when restarting a server and recovering its data. To recover data, the services employed by the Recovery Service (REC), which is a lower-level module of the Encina Toolkit, requires some information to reestablish their state relative to the application. However, this information cannot be part of the recoverable data for the application because it must be present to recover the application’s data. For example, the Volume Service needs to know which volumes it created for the application before it can make those volumes available for recovery. To solve this problem REC coordinates with the Log Service, which maintains persistent data, so REC can provide a single block of restart data that it can immediately pass to an application.

## 4.3 TRPC

The Encina TRPC Service enhances the DCE RPC package offered by the OSF. DCE RPC is a remote procedure call system that implements nontransactional RPCs for use by Encina Toolkit components.

TRPC provides the same basic interface as DCE RPC. However, there is one significant difference. TRPC implements transactional and nontransactional RPCs; DCE RPC implements nontransactional RPCs only. To accommodate the transactional nature of TRPC, the TIDL was developed.

An RPC is a programming paradigm similar to the well-known procedure call mechanism. Both mechanisms transfer control and data within a program. When a remote procedure is called, the parameters of the call are passed over the network to the environment where the call is actually executed. Meanwhile, the calling environment waits for the results of the procedure execution. Typically the calling environment is a program that is referred to as a client. The environment where the call is executed is referred to as a server. When the server (the called environment) finishes executing the procedure, it ships the results back to the client (the calling environment), which then resumes execution as if returning from a local procedure call.

TRPCs are initiated from within the scope of a transaction. Each transactional RPC does work on behalf of a transaction. The TIDL preprocessor adds additional parameters to user-specified operations in an interface definition file. These additional parameters are used to carry the TRAN state and data. TRPCs carry the TRAN state and data along with the regular parameters of the RPC and pass the parameters to the appropriate TRAN.

Although transactional applications commonly use TRPCs, some may also use nontransactional RPCs. Therefore, TRPC provides a mechanism that allows an RPC to retain its nontransactional semantics. TRPC's nontransactional RPCs are an enhanced version of the DCE RPC.

TRPC consists of two components: a preprocessor and a library of functions. The TIDL preprocessor, tidl, preprocesses interface definition files. It produces a set of files that must be compiled and linked appropriately with the client and the server programs. It also produces an interface definition file that must be processed by IDL.

### 4.3.1 Interface Definitions

A TIDL file contains the interface definition for a server. During the design phase, the application designers decide which functions the server will export, what their arguments will be, and so forth. In a TIDL file, you must specify the following about each function (see Figure 13 on page 68):

- The arguments to the function and their types
- Whether the function is to be invoked transactional or nontransactional
- Whether each argument is input, output, or both

```
[
    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 13. Sample TIDL file*

The error_status_t type is defined by DCE for error status. It is equivalent to the unsigned long type, but it also specifies to DCE that it is a status parameter or return value. This fact is used by DCE if you specify that DCE use a status code to return errors it detects to your client rather than generating exceptions.

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 OPS.tidl
```

With Monitor transparent binding the client automatically binds to a server exporting an interface with this UUID when it makes an RPC. With explicit binding, you would specify this UUID when obtaining a binding handle to the server.

You must edit the skeleton produced by the uuidgen command to add the interface name and the prototype for the function that makes up the interface.

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. The TACF is used to control:

- The way binding occurs
- The way errors and exceptions are reported

Unless you specify otherwise in a TACF, binding is explicit: the client must obtain a binding handle to the server and use this handle in all RPCs. If the client is using DCE automatic or implicit binding or the Monitor's transparent binding, you must specify that use in a TACF or an ACF (see Figure 14 on page 69).

The TACF also allows communication 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 communication 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. You can use a TACF to specify that such exceptions are to be returned as status codes, which the client can handle in the same way that it handles other status codes.

```
/* TACF for OrderInterface */

    [implicit_handle (mon_handle_t handle)]
    interface OrderInterface
    {
      [comm_status, fault_status] OrderItem();
     }
```

*Figure 14. Sample TACF File*

You compile the TIDL file, using the TIDL compiler, `tidl`, which produces a number of files, including an IDL file. The IDL file must in turn be compiled by the IDL compiler, `idl`, which produces the necessary stub files (the code that turns the local procedure call into an RPC) and the OPS.h file. The TIDL compiler automatically compiles the TACF for the interface if a TACF file exists.

To implement TRPCs, the tidl command accepts an interface definition file and produces the following:

- **IDL interface definition file** - This file contains modified versions of the operations in the TIDL interface definition file. Each operation in the IDL interface has extra parameters to transmit and receive TRAN data and callback data. The TIDL interface definition file is slightly different from the IDL interface definition file. IDL accepts the modified interface definition file and produces the native client and server stubs.

- **Shadow client stubs** - These stubs are invoked when the client initiates an RPC. The shadow client stubs then invoke the native client stubs produced by IDL.
- **Shadow manager functions** - These functions are invoked when the RPC run time invokes manager functions within the server stubs. The shadow manager functions eventually call the user-provided manager functions.
- **Header file** - Application programs need this file to get the type definitions of the operation parameters and to return values. Both client and server programs must include this file. The header file produced by tidl automatically includes the header file produced by idl.

Figure 15 on page 70 shows the sample file structure of the OPS.

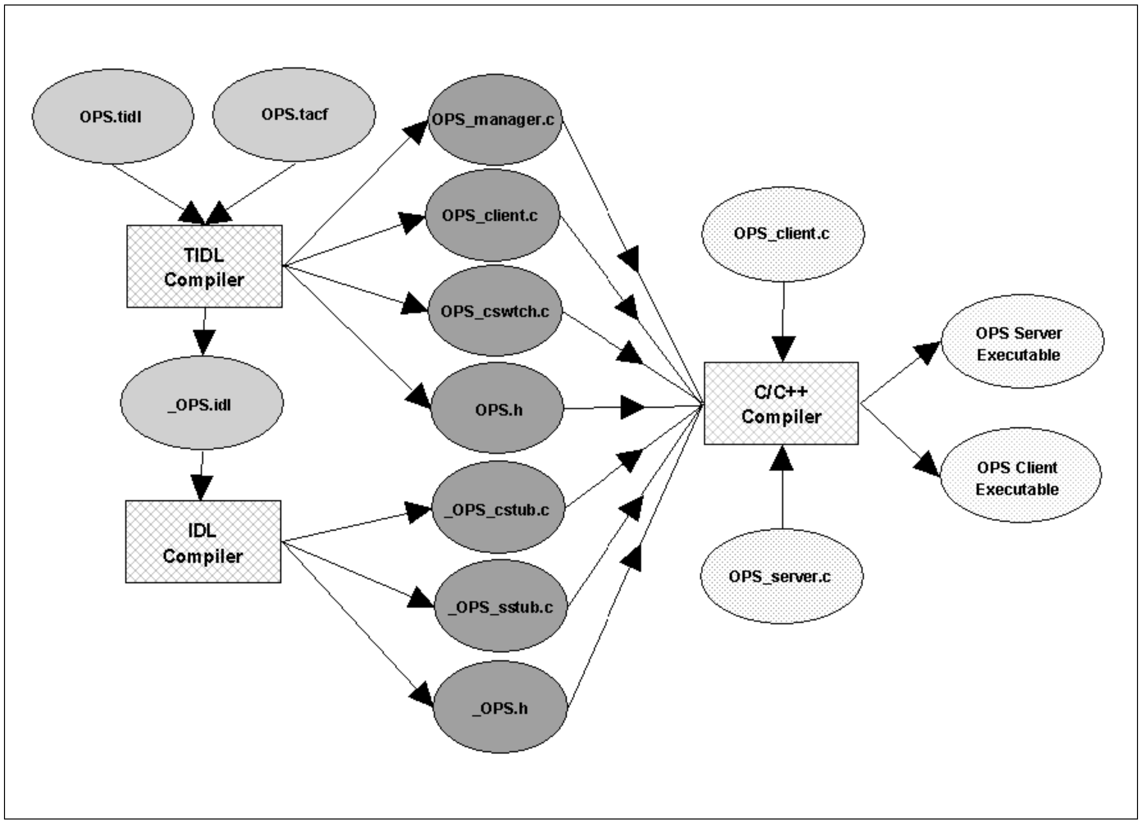


*Figure 15. OrderProcessingSystem Sample File Structure*

Here is a description of the files in the OrderProcessing System file structure shown in Figure 15 on page 70:

- **OPS.tidl:** The TIDL file contains the interface description to be processed by Encina's TIDL compiler. In addition to stub and header files, the compiler generates an IDL file (in our example, OPS.idl) to be used as input to the DCE IDL compiler.
- **OPS.tacf:** The TACF specifies which of the operations defined in the associated TIDL file are to be exported. Use of TACF enables the same TIDL file to contain multiple definitions for an operation. Operations can be exported selectively for different applications by modifying the TACF file.
- **OPS_client.c:** This file contains the shadow client stubs that TIDL produces. It must be compiled and linked with client programs. It contains code that calls the TRPC run time and invokes the modified operations.
- **OPS_manager.c:** This file contains the shadow manager stubs that TIDL produces. It must be compiled and linked with the server. It contains code that calls TRAN, invokes the callbacks, and calls the user-provided manager function.
- **OPS_cswtch.c:** This file must normally be compiled and linked with the client programs. If an application that uses TIDL is to be used as both a client and a server of an interface, it must *not* link with this file. Such an application must invoke the RPC through the entry point vector initialized in OPS_client.c.
- **OPS.h:** The client and server programs must include this file instead of the header file produced by IDL. The jill.h file includes the header file produced by IDL.
- **_OPS.idl:** This is the output interface definition file that must be proceeded by the IDL compiler. TIDL also prefixes an underscore ( _ ) to each operation name in the interface definition file. Each operation in this file also has some additional parameters for transmitting and receiving transaction service data and callback data.
- **_OPS_cstub.c:** This file contains the client stubs that marshal the input parameters and unmarshal the output parameters. It is compiled and linked with the client programs.
- **_OPS_sstub.c:** This file contains the server side stubs that unmarshal the input parameters and marshal the output parameters. It is compiled and linked with the server programs.
- **_OPS.h:** This file is expected to be included in the client and the server programs when raw DCE RPC is used. The header file produced by TIDL (OPS.h, in our example) automatically includes this file.

### 4.3.1.1 DCE-Only RPC Interfaces

TIDL allows an interface attribute to be defined for a DCE-only RPC interface UUID. The `dceOnlyRpcUuid` attribute, as shown in Figure 16 on page 72, can be specified in the interface attribute list of the TIDL file to supply an interface UUID for a DCE-only RPC interface. The supplied UUID is used in the DCE-only RPC IDL file generated by TIDL.

```
[uuid(003549e0-a1f5-1f1a-ba3e-9e620912aa77),
      dceOnlyRpcUuid(00224c0a-a1f7-1f1a-a4c1-9e620912aa77),
      version (1)]
```

*Figure 16. Sample DCE-Only RPC Interface*

If a dceOnlyRpcUuid interface is specified, the TIDL compiler produces a `_dceOnlyRpc.idl` intermediate file, which is proceeded by the DCE/IDL compiler. In our example this file would be called `OPS_dceOnlyRpcUuid.idl`

This process produces the necessary files to be linked to client and server programs to use RPC-only calls. For details about the process, see the *Encina Transactional Programming Guide*.

## 4.4 Security

Encina uses the DCE security features to provide security for transactional applications. 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, it 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 granular 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 least) at 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 according to 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.

Each Monitor cell that is started with DCE security has an access control list (ACL) that controls administrative access to the cell. In addition, there are ACLs that control administrative access to the node managers and managed servers.

When a Monitor cell is first started, the principal named as the exclusive authority is granted exclusive access to the cell manager. If you are using Enconsole, the default exclusive authority is the encina_admin user. Enconsole automatically grants full administrative permissions to the encina_admin_group and then clears the exclusive authority.

If you are not using Enconsole, the exclusive authority must create entries on the cell manager ACL. For example, the read (r), write (w), and administer (a) permissions for the Monitor cell can be granted by adding the appropriate entry to the cell manager ACL. After permissions have been set up, the exclusive authority must be cleared by using the tkadmin clear exclusiveauthority command.

If security is enabled, servers can have ACLs that grant permissions to principals to perform administrative tasks such as listing and aborting transactions. In the absence of an ACL for a specific server, the cell manager ACL is used.

To create an ACL for a managed server, use the dcecp acl modify command. The ACLs for administering managed servers are /.:/cellname/ecm/server/servername, where cellname is the name of the Monitor cell and servername is the name of the server, for example, /.:/branch1/ecm/server/merchandise. To issue the dcecp acl modify command, you must have the Encina Monitor administer (a) permission, and the cell manager must be running with authorization enabled.

If security is enabled, application interfaces and the functions within them can have ACLs that grant execute (x) permission to client principals. If a function has an ACL, that ACL is used, and the interface ACL is ignored. Otherwise, the ACL for the interface is used.

The ACLs for interfaces are /.:/cellname/ecm/interface/interface_name, where interface_name is the name of the interface exported by the Monitor application server-for example, /.:/branch1/ecm/interface/merchandise.

The ACLs for functions are /.:/cellname/ecm/interface/interface_name/function_name, where interface_name is the name of the interface and function_name is the name of a function in that interface, for example, /.:/branch1/ecm/interface/merchandise/write_checks.

The Encina Monitor automatically uses a number of the DCE security features. You do not have to modify the application to use security in the Monitor environment. You specify the level of security to use when starting the application server. You can use two security features:

- RPC authentication. The administrator who starts the server specifies an RPC authentication level. The Monitor rejects any RPCs from clients that do not specify at least that level.
- 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 both cases, the program does not have to do anything. The server never sees the rejected RPCs.*

Each RQS server has an ACL governing access to it. When an RQS server is first started, the principal named as the exclusive authority is granted exclusive access to the server. If you are using Enconsole, the default exclusive authority is the encina_admin user. Enconsole automatically grants full administrative permissions to the group encina_admin_group and then clears the exclusive authority. A user holding the create (c) permission on a server can create queues, queue sets, and element types in the server. Any user who creates a queue or queue set is automatically granted full permissions on the queue or queue set. For instance, when a queue is created, its ACL initially consists of an entry that grants the creator all permissions on the queue.

You can use ACLs to protect SFS objects against unauthorized access. Each SFS server that is started with DCE security has an ACL that controls access to the server and, indirectly, to the files that the server manages. In addition, each SFS file has an ACL that controls access to it.

Users with the administer (A) permission on an SFS server automatically acquire the administer (A), exclusive open (E), and query (Q) permissions on all files managed by the server and create (C) and query (Q) permissions on the server. Users with the administer (A) permission on an SFS file automatically acquire the exclusive open (E) and query (Q) permissions on that file. Note that the creator of a file automatically gets full permissions (A, D, E, I, Q, R, and U) on the file.

For sfsadmin commands that modify a file or its indexes, file permissions are checked first, followed by server permissions. Because the administer (A) permission on a server indirectly grants full file permissions, lack of file permissions does not necessarily prevent file modification commands. Therefore, the administer (A) permission on a server should be granted cautiously.

Each PPC gateway server using DCE security has an ACL that controls access to the server. The ACL contains entries that grant specific access permissions to specific users or groups.

In addition, for SNA-to-Encina conversations, the PPC gateway server requires Monitor execute (x) permission on PPC Executive applications in the Monitor to schedule conversations from LU 6.2 applications. The PPC gateway server principal must be on the ACL for the LU name registered for the PPC Executive application or on the ACL for the individual transaction program name (TPN).

When a PPC gateway server is first started, the principal named as the exclusive authority is granted exclusive access to the server. If you are using Enconsole, the default exclusive authority is the encina_admin user.

Enconsole automatically grants full administrative permissions to the group encina_admin_group and then clears the exclusive authority.

If security is enabled for a PPC gateway server, you must add entries to the gateway server ACL to give principals access to it. You must have the PPC administer (a) permission on the gateway server ACL to add an entry to the ACL. Because users with administer (a) permission can grant anyone access to the server, you should grant the administer (a) permission cautiously.

# Chapter 5. Using Encina Components

In this chapter we describe the Encina components in detail and explain how to use them for application development.

## 5.1 Encina Monitor

The Encina Monitor provides features to simplify programming and automate some of the basic tasks that a client/server application must perform. The Monitor also provides a number of run-time and administrative benefits, including load balancing and centralized administration.

The Encina Monitor provides three function areas to a transaction processing system:

- The run-time environment
- The system administration facility
- The application development environment

The Monitor run-time environment coordinates TP applications and resource managers and performs run-time administration tasks, such as load balancing and collecting diagnostics. In addition, this environment provides for other interactions with the execution environment, such as scheduling calls for later execution and retrieving information about users, transactions, and client/server bindings.

The Monitor system administration interface is used to construct, initiate, control, and terminate a Monitor system. The Monitor is administered through Monitor administrative and configuration interfaces.

Monitor applications are developed by using the Monitor API in conjunction with other Encina interfaces, such as Tran-C. The Monitor saves the programmer effort by performing some tasks, such as interaction with DCE RPC and security, on the application's behalf.

### 5.1.1 Run-time Environment

The Monitor run-time environment is not a single process. It consists of a number of processes that together build the platform for TP applications. This run-time environment is also called the *Monitor cell*. A Monitor cell is always part of a DCE cell and uses DCE services. A DCE cell can contain one or multiple Monitor cells. The processes that make up the Monitor cell can also be distributed over different nodes (computer systems). These nodes are called *Monitor managed nodes*. Figure 1 on page 78 shows a simple Encina Monitor configuration.

A Monitor managed node is capable of simultaneously running one or more client or application servers and one or more Monitor system processes. Because of the value of the enterprise data entrusted to such a computer system, a Monitor managed node usually is a highly trusted computer system. Application servers must run on managed nodes. Clients do not have to run on managed nodes.

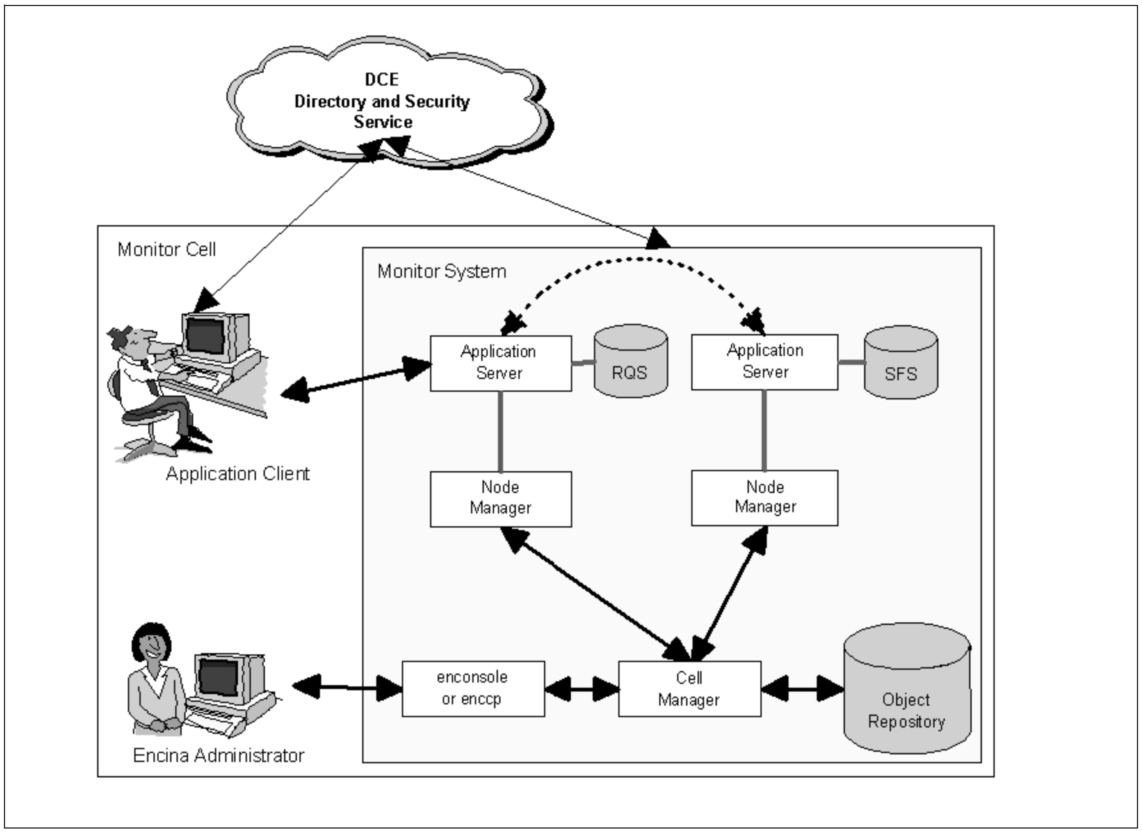


*Table 1. Simple Encina Monitor Configuration*

A Monitor cell contains the following components:

- **Cell manager** - The part of the Monitor that monitors and controls the node managers and data repository within the cell. The data in the cell manager’s data repository describes the application servers configured to run on each node in the cell and which users are authorized to use which services. In addition to storing authorization data, the cell manager stores information about active users, application servers, and the services in a cell. The cell manager communicates with node managers, clients, and DCE services.
- **Node manager** - The part of Encina that controls all application servers on a single managed node on behalf of the cell manager. The node manager starts and monitors the application servers running on a single managed node. Each managed node in the Monitor system must have a node manager. The cell manager and its node managers monitor the system constantly, detecting, reporting, and restarting application servers that have failed. Therefore the node manager periodically polls its application servers to confirm that they are still responding to requests. When an application server fails to respond, the node manager notifies the cell manager of the failure and automatically restarts the application server. Node managers must be Monitor managed nodes; they cannot run on public nodes.
  *Programmer intervention is not required to run the node manager. The system administrator need only supply the necessary configuration information.*
- **Application server** - Application servers process client requests and run on managed nodes, usually interacting with one or more resource managers in a manner specific to the resource manager. Application servers can be recoverable or nonrecoverable. Recoverable servers that fail can be restored to a consistent state, and transactions are guaranteed either to commit or abort cleanly rather than being left in intermediate states. A nonrecoverable server does not log and recover its data; if it fails, any of its data that has not already been committed as part of a distributed transaction is simply lost. An application server consists of one or more processes called *processing agents* (PAs) that receive and handle client requests for services. Processing agents are multithreaded processes that execute requests received from clients. A single application server with several processing agents only has to be configured and started once, and its processing agents can be managed as a single group. The Monitor automatically parcels out client requests among the various processing agents, and you do not have to worry about each processing agent individually. An application server can consist of up to MON_MAX_PA_COUNT (64) PAs.

- **Application client** - The part of the application through which a user interacts with the Monitor system, making requests for services exported by active application servers. Clients can be implemented in several ways. They can be designed for single or multiple simultaneous users and have command- or forms-based interfaces. The choice of interface type depends on the expected demand, the expertise of the users, and other user interface design issues. Clients are the only components that do not have to run on managed nodes.
- **Resource manager** - The resource manager is a component that manages a shared resource, such as RDBMSs, MQSeries, or file system servers. The Encina SFS and RQS are also examples of resource managers.
  The Encina Monitor supports two types of interaction with resource managers. A native resource manager, such as the SFS, is one that interacts with the Monitor through components of the Encina Toolkit (TRAN, TRPC, or PPC, for example). Such resource managers provide transactional consistency transparently to the Monitor. An XA-compliant resource manager such as DB2 or MQSeries is one that interacts with the Monitor using the X/Open XA interface. This type of resource manager is registered with the Monitor and interoperates with the Monitor through the Toolkit's TM-XA component.

### 5.1.2 Application Development Environment

The Monitor application development environment consists of the application development tools, languages, and libraries that enable programmers to develop application programs for the Monitor. The development environment has three major components:

- Tran-C, a collection of C language functions and macros that is used to construct application servers and client applications
- The RPC mechanism, which is used to define and execute client/server interactions. We discuss this mechanism in Chapter 4.3, “TRPC” on page 66
- The Monitor application development libraries

#### 5.1.2.1 Transactional-C Programming Language

Encina's Tran-C is a C interface to the Encina Toolkit that simplifies the development of transactional applications by greatly reducing the number of necessary calls to the underlying Encina Toolkit. Tran-C consists of macros and library functions that embrace the commonly used functionality of the Encina Toolkit, eliminating the need for direct access to the Toolkit module interfaces in most cases. Tran-C does not incorporate seldom-used routines from other Toolkit modules (for example, Transaction Service routines that return information about the various relationships of transactions) or Recovery Service routines.

Tran-C provides a simpler application development environment than that provided by the Toolkit modules themselves. When directly using the interfaces provided by the Toolkit modules, you must make all of the required calls to each module and execute those calls in the correct sequence. For example, most Toolkit modules have their own initialization calls. These calls must be executed in the correct sequence, because some modules depend on services provided by others and must therefore be initialized after those other services. The Tran-C development environment is slightly more restrictive than that provided by the Toolkit interfaces but is much easier to use. It provides transactional extensions at the level of added language constructs, rather than by adding function libraries. The difference between developing applications by using the Toolkit or Tran-C is analogous to the difference between writing applications in assembler and a high-level language such as C.

Tran-C adds a number of important constructs to the C programming language, where constructs are groups of associated phrases that together make up an entity affecting the flow of control in a program. The most important of these is the transaction construct, which consists of a transaction clause, an onCommit or suspend clause, and an onAbort clause. Each clause contains a statement, which can be a compound statement (a collection of statements in brackets). Each transaction construct must contain an onAbort clause. The onCommit and suspend clauses are optional and, if specified, must precede the onAbort clause. Figure 17 on page 82 shows a "Hello World" Tran-C example demonstrating how the transaction construct can be implemented.

```
#include <tc/tc.h>
     inModule("helloworld");

     void main()
     {
       int i;
       inFunction("main");
       initTC();
       for(i=0;i<10;i++) {
         transaction {
             printf("Hello World - transaction %d\n", getTid());
             if (i % 2)
                 abort("Odd Numbered Transactions are aborted...");
         }
         onCommit
             printf("\t(Transaction committed)\n");
         onAbort
            printf("Aborted in module : %s\n\t%s\n", abortModuleName(),
                                   abortReason());
       }
     }
```

*Figure 17. "Hello World" Tran-C Example*

The program aborts all odd numbered and commits all even numbered transactions. Figure 18 on page 82 shows an extract of the program's output.

```
Hello World - transaction 65536
             (Transaction committed)
     Hello World - transaction 1
     Aborted in module : helloworld
             Odd Numbered Transactions are aborted...
     Hello World - transaction 2
             (Transaction committed)
     Hello World - transaction 3
     Aborted in module : helloworld
             Odd Numbered Transactions are aborted...
     Hello World - transaction 4
             (Transaction committed)
     Hello World - transaction 5
     Aborted in module : helloworld
             Odd Numbered Transactions are aborted...
```

*Figure 18. Output of the "Hello World" Tran-C Example*

The core of the "Hello World" program is a simple printf statement, enclosed within the Tran-C transaction construct. Because the transaction construct is executed 10 times, each iteration of the printf statement is actually executed by a different transaction. In a transaction processing environment, each transaction has a unique name, the TID. The Tran-C run-time system automatically manages passing, generating, and manipulating the TIDs in Tran-C programs. Ordinarily, a TID is transparent because it is seldom necessary to know it for any given transaction.

This sample program illustrates the use of the Tran-C transaction construct in a very simple situation. If you need more information about how to incorporate RPCs or construct nested transactions, we recommend that you read Transarc’s *Encina Transactional Programming Guide.*

#### 5.1.2.2 Application Programming Interface

The Monitor contains a collection of functions to support the development of distributed transaction processing applications, such as functions allowing clients and application servers to register with the Monitor system and obtain a binding for a given service. In addition to the functions needed to operate in an Encina Monitor system, the Monitor is compatible with a number of the Encina Toolkit components, so calls to the lower level components can be made.

### 5.1.3 Client/Server Application Development

The technical steps to develop a Monitor client/server application are:

1. Define the interface.
2. Write the server.
3. Build the server.
4. Write the client.
5. Build the client.

We assume that the reader is familiar with the necessary analysis steps needed in order to start with the interface definitions. Below we concentrate on the technical part of the development process.

#### 5.1.3.1 Defining the Interface

As discussed in Chapter 4.3, “TRPC” on page 66, an interface is a description of a group of functions to be provided by an application server and the arguments those functions take. An interface is exported by an application server and typically has an ACL specifying who may use the interface. Interfaces are specified with TIDL. Both TIDL and IDL are used in developing

Monitor applications; TIDL generates, as part of its output, the input to IDL. To define an interface, first create a file that describes the interface. TIDL files are usually named *interface-name.tidl.* Figure 19 on page 84 shows an extract of the Order Processing Interface of our sample application, OPS.

```
[ uuid(000b4aa0-7911-14cf-845a-00a024c008a6), version(1.0) ]

interface OrderProcIF {

   import "tpm/mon_handle.idl";
   import "Common.idl";

   /* is the server alive (ready to receive RPCs) */
   [nontransactional] RpcReturn ping();

   /* create an order */
   [nontransactional] RpcReturn createOrder(
   [out] long* orderId);

   /* list all the products that can be ordered */
   [nontransactional] RpcReturn listProducts(
   [out] ProductListPtr* prodList);

   /* add quantity of a product to order */
   [nontransactional] RpcReturn orderItem(
   [in]  long orderId,
   [in]  long productId,
   [in]  long quantity);

   /* review order using orderId, returns the original order,
   and up-to-date status */
   [nontransactional] RpcReturn reviewOrder(
   [in]  long orderId,
   [out] OrderInfoPtr* ordInfo);
}
```

*Figure 19. Sample TIDL for OrderProcIF*

The function descriptions are similar to standard C prototypes. The [in] and [out] labels refer to the type of argument; input parameters are used but not modified, and output parameters are placeholders for information to be returned. A third option, [in, out], describes parameters that act as both input and output.

With transparent binding, there are three additional requirements for defining the interface:

1. The TIDL file must import the mon_handle.idl file.
2. The functions specified in the TIDL file must not specify handles as parameters.
3. You must provide a TACF that specifies that the binding handle is an implicit handle.

With transparent binding, the client relies on the Monitor to create the binding and obtains a handle to an application server that exports a requested interface. The Monitor selects a server from all servers in the cell that export the requested interface; the application cannot specify a particular application server. Thus the Monitor can balance the workload among all available servers that export an interface. Figure 20 on page 85 shows an example of the required TACF for the Order Processing Interface. This input file is required only with transparent binding.

```
[implicit_handle (mon_handle_t handle)]

interface OrderProcIF
{
}
```

*Figure 20. Sample TACF for OrderProcIF*

#### 5.1.3.2 Writing the Server

Once the interface is defined, you must write the server and the client. Writing the server consists of the following steps:

1. Implementing the manager functions
2. Writing the code needed for server initialization and server termination

Steps typically constitutes the bulk of the work. In our Order Processing Interface example we produced the orderprocif_manager.c file, which has to be linked together with the server main program. Functions such as ping_msr are defined in this file and conform to the interface definition with regard to the function’s input and output. Figure 21 on page 86 and Figure 22 on page 87 show an extract of orderprocif_manager.c describing the ping_msr function.

```
RpcReturn ENCINA_RPC_CALLING ping_msr
#ifdef IDL_PROTOTYPES
   (
      handle_trpcHandle_,
      trpc_byteData_t  applAndAddress_,
      idl_ulong_int  applAndAddressLength_,
      trpc_callbackData_t  inCallbackData_,
      idl_ulong_int  numOfInCallbackData_
   )
#else
   (
      rpcHandle_,
      applAndAddress_,
      applAndAddressLength_,
      inCallbackData_,
      numOfInCallbackData_
   )
   handle_trpcHandle_;
   trpc_byteData_t  applAndAddress_;
   idl_ulong_int  applAndAddressLength_;
   trpc_callbackData_t  inCallbackData_;
   idl_ulong_int  numOfInCallbackData_;
#endif
{
   trpcStub_call_tcallHandle_;
#define inMsg_ ((idl_byte *) 0)
#define inMsgLength_ ((idl_ulong_int) 0)
#define outMsg_ ((idl_byte *) 0)
#define outMsgLength_ ((idl_ulong_int *) 0)
#define remoteAddrString_ ((idl_byte *) 0)
#define remoteAddrStringLength_ ((idl_ulong_int) 0)
   RpcReturn  returnValue_;

   trpcStub_EnterManagerStub(1, &callHandle_);
```

*Figure 21.  Extract of orderprocif_manager.c (Part 1)*

```
   TRY
      trpcStub_ManagerPrologue(1, &callHandle_, rpcHandle_,
                               OrderProcIF_v1_0_s_ifspec,
                               interfaceName_,
                               "ping",
                               applAndAddress_,
                               applAndAddressLength_,
                               inMsg_, inMsgLength_,
                               inCallbackData_,
                               numOfInCallbackData_);

      returnValue_ = ping();


      trpcStub_ManagerEpilogue(&callHandle_,
                               outMsg_, outMsgLength_);
   CATCH_ALL
      trpcStub_ManagerCatchClause(&callHandle_, THIS_CATCH);
   ENDTRY

   return(returnValue_);
#undef inMsg_
#undef inMsgLength_
#undef outMsg_
#undef outMsgLength_
#undef remoteAddrString_
#undef remoteAddrStringLength
}
```

*Figure 22. Extract of orderprocif_manager.c (Part 2)*

An application server consists of one or more processes called PAs that receive and handle client requests for services. Processing agents are multithreaded processes that execute requests received from clients. They allow parallel processing without the administration overhead of running several identical application servers. A single application server with several processing agents only has to be configured and started once, and its processing agents can be managed as a single group. The Monitor automatically parcels out client requests among the various processing agents; the programmer need not worry about each processing agent individually.

The Monitor application server program runs under the control of the Monitor and executes manager functions in response to the client requests. The

Monitor automatically handles the initialization of DCE, Toolkit, and Tran-C. These components must not be initialized by the Monitor application server main program. To enable the application server program to the Monitor, you must program the following steps:

1. Perform any application-specific initialization that does not require DCE or Encina, such as setting up the application environment.
2. Register any interfaces it will use by calling the `mon_InitServerInterface` function. For example:
   ```
   mon_status_t mst;
   mst = mon_InitServerInterface(MON_SERVER_INTERFACE(OrderProcIF,1,0));
   ```
3. Register any PPC transaction program names (TPNs) by calling the `mon_RegisterTPN` function. For example:
   ```
   mon_status_t mst;
   mon_ppcSchedMgr_t HandleDBQuery_MVS
   mst = mon_RegisterTPN("UDBQUERY_MVS", HandleDBQuery_MVS);
   ```
   The `HandleDBQuery_MVS` function is called when a PPC scheduling request is received by this processing agent for the transaction `MVSQUERY`.
4. Register any XA resource managers by calling the `mon_RegisterRmi` function. For example:
   ```
   mon_status_t mst;
   struct xa_switch_t *xaSwitchP;
   char *rmName;
   int *rmiID
   mst = mon_RegisterRmi(xaSwitchP, rmName, &rmiID);
   ```
   The function causes an X/Open XA-compliant resource manager to be registered. The `xaSwitchP` identifies the resource manager where `rmName` defines the instance. The value of `rmiID` can be used in SQL calls that take an interface ID. This function must be called during server initialization, before the `mon_InitServerfunction` is called, if it is called at all.
5. Register any Queued Request Facility (QRF) manager functions by calling the `mon_RegisterQrfManagerFunction` function. When the QRF manager function is registered, entries in specified RQS queues are automatically routed to the appropriate application servers to be handled by the registered manager functions.
6. Request initialization of TX instead of Tran-C (which is the default) by calling the `mon_ServerUsesTx` function. The mon_ServerUsesTx function causes the Monitor to call the tx_open function as part of server initialization. Thus the application server need not explicitly invoke the tx_open function.

7. Enable the environment retrieval routines by calling the `mon_RetrieveEnable` function. The mon_RetrieveEnable function turns environment retrieval on or off. If the flag is TRUE (nonzero), the data that is returned from the various environment retrieval functions is forwarded to other components; if it is FALSE (zero), the data is not forwarded. Any calls to the environment retrieval functions yield a null pointer. The environment retrieval functions are disabled by default.
   **When environment retrieval is turned on, the cost of each RPC initiated by the client or application server is significantly higher than when environment retrieval is not enabled.**
8. Set the scheduling policy by calling the `mon_SetSchedulingPolicy` function. This function is used to specify whether a server permits shared access. If exclusive access is desired, no call is necessary, but one may be made. Once this call is made, the specified access occurs automatically in all PAs for the server. Because all PAs use the same executable, it is not possible to have some PAs provide shared access and others provide exclusive access.
9. Use the `mon_InitServer` function to initialize the server and the underlying Encina and DCE components. After this function returns, the server can start transactions and make RPCs to other servers. This function is optional, if the server does not call this function, initialization occurs automatically when the `mon_BeginService` function is called. However, there are certain tasks that may not be performed until after Encina has been initialized, such as making transactional RPCs and initializing shared memory. There may be situations when you may want to perform these actions before the application server receives any RPCs. Such actions must be performed between the call to the `mon_InitServer` function and the call to the `mon_BeginService` function.
10. After initialization is complete, the server calls the `mon_BeginService` function. This function registers the interfaces that were exported by the server (using the `mon_InitServerInterface` function), enabling the server to begin accepting RPCs. It does not return as long as the server is active. It returns when the server is shut down, either administratively or through a call to the `mon_TerminateServer` function.
11. To shut down the application server, the `mon_TerminateServer` function may be called from within the program, causing the application server to shut down after a short period to allow transactions to complete. However, servers are usually shut down by system administrators using enccp or enconsole.

#### 5.1.3.3 Writing the Client

Unlike application servers, which usually run for a long time and may require administrative intervention to be shut down, clients are typically run by a user and terminated when the user no longer needs them. A typical client application initializes, issues calls to application servers, cleans up, and exits. Clients must include the mon_client.h header file and the header file generated by tidl. Clients that use the JAM user interface must also include mon_jam.h.

A client can incorporate an interactive user interface but can also use other methods for generating the requests. Clients are the only components that do not have to run on managed nodes. Clients can be implemented in several ways. They can be designed for single or multiple simultaneous users and can have command- or forms-based interfaces such as JAM. The choice of interface type depends on the expected demand, the expertise of the users, and other user interface design issues.

***Client Initialization***

The first action a client application must perform is initialization of the Monitor. The mon_InitClient function initializes the DCE, Toolkit, and Tran-C as well as the Monitor itself. Clients must not initialize these components themselves. The `mon_InitClient` function must be called once in each client, any other Monitor functions are called. Application servers that also act as clients do not need to call this function.

The `mon_InitClient` function has the following syntax:

```
mon_status_t mon_InitClient(

        char *clientName,

        char *cellName)
```

The first argument, clientName, names the client. This name is used by various Monitor environment retrieval functions to identify the client but not for authorization. The second argument, cellName, is the name of the Monitor cell.

***Client - Server Binding***

Before a client can make an RPC to an application server, it must bind to the application server that exports that service. There are two ways in which a client can bind to a server: transparent binding and explicit binding.

With transparent binding, the client relies on the Monitor to create the binding and obtain a handle to an application server that exports a requested interface. The Monitor selects a server from all servers in the cell that export the requested interface; the application cannot specify a particular application server. Thus the Monitor can balance the workload among all available servers that export an interface.

With explicit binding, the client specifies the application server to which it will bind. The client obtains a binding handle to that server and explicitly uses that handle in subsequent RPCs.

Explicit binding gives the client application a higher degree of control over the binding process but is more complex than transparent binding. Transparent binding accomplishes binding without specific application code and is the preferred method.

When a client using transparent binding makes an RPC to a server, it obtains a server handle to one of the server's PAs. If the PA is busy, the client waits until the PA becomes available. This is also called the *no reservation* mode and is the default for transparent binding. A client can, however, choose to reserve a PA on a short-term basis by calling the `mon_AcquireReservations` function or issue a long-term reservation when the application uses explicit binding only. The short-term reservation lasts for the duration of a transaction.

Servers can specify three types of client access for processing agents: exclusive, shared, and concurrent shared. With exclusive access only one client can reserve the PA at any given time, and only one RPC from that client can be active at any time; other RPCs are queued, and clients cannot interfere with each other. This is the default behavior when clients reserve PAs. With shared access more than one client can reserve the PA, but only one RPC can be active at a time. With concurrent shared access more than one client can use the PA, and multiple RPCs from those clients can be active simultaneously. For more information about how to use PAs, see the *Encina Monitor Programming Guide.*

### *Security*

If an application uses security, clients must be able to perform the appropriate operations to obtain principals and encrypt RPCs. As part of their configuration, application server interfaces have information about security. The client can control the DCE security level of the RPC in several ways:

- Choose to do nothing. RPCs are made with the level set by the server.
- Set default values for authentication. These values are used for all transparent binding RPCs until new values are set.

- Register a callback function that is called when each RPC is made by a client if transparent binding is being used. The application itself then sets the protection and authorization levels in the callback function. The DCE `rpc_binding_set_auth_info` function is usually used to set these security attributes. Security callbacks can be used, for example, to specify different protection levels for different interfaces or principals. A security callback is a function that is security related and executed before any RPC is sent.
- Use the DCE Security Service functions to handle these issues if explicit binding is used. See the *OSF DCE Application Development Guide* for details.

The `mon_SecuritySetDefaults` function changes the default security levels for all outgoing RPCs from the calling client. If there is a security callback, these values are not used; otherwise, they are used until changed:

```
mon_status_t mon_SecuritySetDefaults(

         unsigned32 authnLevel,

         unsigned32 authzSvc)
```

The authnLevel argument specifies the DCE protection level and the authzSvc argument specifies the DCE authorization level. You can specify either `rpc_c_authz_none` or `rpc_c_authz_dce`.

The following call sets the highest level of authentication:

```
mon_SecuritySetDefaults(

             rpc_c_protect_level_pkt_privacy,

             rpc_c_authz_dce)
```

The following call turns off security for the client:

```
mon_SecuritySetDefaults(

             rpc_c_protect_level_none,

             rpc_c_authz_none)
```

**Note**

Servers may have more stringent security requirements than clients. Clients cannot dictate security; they can only indicate which security level they will use.

The `mon_SecurityRegisterCallback` function registers a function to perform any application-specific security work and then return either TRUE or FALSE, depending on whether the RPC should proceed. The `mon_SecurityRegisterCallback` function takes one argument, the function to register. It has the following syntax:

```
mon_status_t mon_SecurityRegisterCallback(

        mon_secCallback_t appl_SecurityCallback)
```

The function pointed to by the appl_SecurityCallback parameter is called once before each RPC that uses transparent binding. The function is responsible for setting any authentication and authorization attributes. If a registered callback exists, `mon_SecurityRegisterCallback` replaces it.

#### *Terminating the Client*

The `mon_ExitClient` function replaces the C `exit` function in clients. When this function is called, all unprepared transactions are aborted. The syntax for this function is:

```
void mon_ExitClient(

        int status)
```

The `mon_ExitClient` function takes one argument, an integer value to return to the calling environment (the shell, for instance). This value corresponds to the argument that the exit function takes.

## 5.2 Encina SFS

The Encina SFS is a record-oriented file system that provides transactional integrity, log-based recovery, and broad scalability. Many operating systems support only byte-stream access to data: all input and output data, regardless of its source, is treated as an unformatted stream of bytes. SFS uses structured files, which are composed of records. The records themselves are made up of fields. For example, each record may contain the information about an employee, with fields for the name, employee number, and salary. Although SFS looks like a database, it is not a database system but must be seen as an enhanced, structured data storage and retrieval environment. In general, SFS is the transaction component that allows transaction programs to access common data in a transactional manner (ACID properties), regardless of whether the programs run on one system or are distributed across different platforms.

All data in SFS files is managed by the SFS server. Programs that require access to this data must submit their requests to that server, which retrieves the requested data or performs the specified operation.

SFS provides both data processing and administrative functions. The data processing functions provide the standard operations used to access and modify data: read, insert, update, delete, lock, and unlock. The administrative functions enable programs to create, query, and modify SFS files and volumes, copy files, and delete files.

For system administrators, SFS also provides a system administration tool, sfsadmin, which provides a command-line interface for the functionality provided by the SFS administrative functions.

SFS brings the following benefits to transaction-oriented applications:

- **Transaction protection.** SFS provides transactional access to data stored in a file. Files managed by SFS are thus fully recoverable from server problems, network outages, and media failures. SFS automatically keeps a record of any changes made to the data stored in SFS files. SFS ensures that any changes that were in progress when system problems occurred are either completed or completely undone.
- **Record-oriented files.** SFS supports record-oriented file types. SFS organizes the records into fields and provides both record-level and field-level access to data. Access to the records is through indexes, enabling an application to easily and quickly access records on the basis of one or more fields in the record.
- **Support for distributed computing and open systems.** SFS provides a consistent mechanism for requesting access to structured data across multiple platforms. The client/server model used by SFS allows applications to be easily and transparently distributed on the network.
- **Ease of porting existing applications.** SFS enables you to port existing structured file or database applications. SFS provides a logical rather than physical data model, minimizing portability problems across systems with different byte-ordering or other concerns.
- **ISAM compatibility.** The Encina Transactional Indexed Sequential Access Method (T-ISAM) library provides an X/Open ISAM-compliant

method of accessing data stored in SFS. The T-ISAM library contains all of the C-ISAM functions required for C-ISAM application source compatibility.

- **COBOL record interface.** The SFS External File Handler (EXTFH) supports the use of Micro Focus COBOL with SFS. Existing COBOL applications, using standard COBOL I/O statements, can be made to access SFS files; the native COBOL I/O calls are transparently mapped to SFS calls. See the *Encina COBOL Programming Guide* for more information.
- **Compatibility with database systems.** The Encina Transaction Manager-XA (TM-XA) Service enables SFS applications to interact with database applications that support the X/Open XA interface.
- **Compatibility with the RQS.** Many SFS definitions are compatible with the RQS. For example, the field types used by SFS have corresponding types in RQS. Thus, applications can easily use both SFS and RQS.

SFS is built on top of the Encina Toolkit Server Core and the Encina Toolkit Executive. It provides a higher-level set of functions for the manipulation of structured data without requiring that the user be familiar with most of the details of DCE or the Encina Toolkit. The transactional services required by SFS are mostly transparent to the user. Because SFS uses those elements of the Encina Toolkit that enable recovery in the event of failure, the programs you write do not have to handle such failures; that is, your program need not include any code to enable recovery or to directly access the Encina Toolkit components used for recovery.

A program that uses SFS is a client to an SFS server. If that client processes records in a file, the program must perform the following steps:

1. **Perform the initialization.** Calling an SFS function automatically performs the internal initialization required by the SFS client library functions. Therefore, SFS itself requires no special initialization. However, SFS uses lower levels of the Encina Toolkit, which you must initialize:
   - TRPC
   - Tran-C (see Figure 23 on page 96)
   - Any other Encina components you are using
2. **Open any SFS files the program is going to use.** Any file that you are going to use must first be opened. When a program opens a file, it specifies the name of the file that it wants to access and how it wants to access it. Section 5.2.4, “Opening an SFS File” on page 104 describes this process in detail.

3. **Perform I/O on the files.** Programs have several options as to how to handle records. SFS can read or write records as a single buffer; the program is then responsible for packing the field into or unpacking the fields from that buffer. Alternatively, SFS can place some or all of the fields into their own buffers. Programs also have the option to read or update only some fields in a record. Records can be accessed randomly or sequentially.
4. **Close the files.** The Open File Descriptor (OFD) can be closed at any time, regardless of whether any transaction in which it was involved has actually completed. The `sfs_CloseOfd` function closes an OFD. This function is not needed for OFDs for which `autoClose` was specified when the file was opened because the OFD is automatically closed when the transaction is resolved.

```
void InitWithTranC()
     {
         trpc_status_t status;

         preInitTC();
         status = trpc_InitWithTrdce();
         ENCINA_STATUS_CHECK(status);
         tc_InitTRPC();
         postInitTC();

     }
```

*Figure 23. Sample SFS Initialization Using Tran-C*

### 5.2.1 File Names

The full name of an SFS file is specified by giving the name of the server on which it resides, followed by the name of the file itself. For example, if the server name is `/.:/encina/sfs/my_sfs_server` and the file name is `my_first_file`, the full file name would be `/.:/encina/sfs/my_sfs_server/my_first_file`. The name in this form is referred to as a fully qualified CDS file name. You pass this fully qualified CDS name to any function that takes a file name as an argument, such as `sfs_OpenFile`.

### 5.2.2 File Structure

SFS files are record-oriented files; user data is organized as a collection of records. A record is a grouping of related information with a predefined size and a predefined number of fields that hold specific parts of the record's information. These fields can be of various predefined data types. The field layout of a record is defined when the file is created. The way the records in a file are arranged is referred to as the *file organization*. The records in a file can be organized in one of three ways:

- Entry-sequenced
- Relative
- Clustered

The records in an entry-sequenced file are stored in the order in which they are written into the file. New records are always appended to the end of the file. When records are deleted from an entry-sequenced file, the space formerly allocated to those records is not automatically reclaimed or reused. The only way to reclaim this space is by using the `sfs_ReorganizeFile` function. Entry-sequenced files are often used when records in the file will be accessed in the order in which they are written to the file. This type of file organization is frequently used for log files, audit trail files, or for any other files that keep time-sequenced records of events. Each record in an entry-sequenced file has an entry sequence number (ESN), which corresponds to the order in which it was inserted into the file. The ESN is not part of the record.Therefore you specify only the name of the primary index when creating an entry-sequenced file in SFS.

A relative file is an array of fixed-length slots. Records can be inserted in the first free slot found from the beginning of the file, at the end of the file, or in a specified slot in the file. Relative files are often used when records will be accessed directly, by record number. Because all of the slots in a relative file are the same size, SFS can calculate the position of a specific record (identified by record number) by multiplying the record number by the record slot size. The primary index of a relative file is based on the relative slot number (RSN), which represents the number of the slot occupied by a record. The first relative slot number in a file is slot number 0, and the highest slot number cannot exceed the maximum number of records specified when the file was created. The RSN is a physical part of the user's data record. When a relative file is created, at least one field must be of type sfs_unsignedInt32. This field must be specified as the primary index.

A clustered file (also called a *B-tree-clustered* file) is a tree-structured file in which records with adjacent index values are clustered together, to reduce the cost of searching for ranges of records. The clustered file organization used in clustered SFS files is automatically maintained by the SFS server. Records in a clustered file do not have a numeric record index such as an ESN or RSN. The primary index can be based on any field or combination of fields. The records in a clustered file are ordered according to the contents of the primary index. Because the SFS may move records to maintain clustering when new records are inserted or deleted, there is no practical way to maintain direct references to individual records. Disk space freed by record deletions is automatically reused.

Table 2 on page 98 compares the three SFS file organizations.

*Table 2. SFS File Organizations*

| | **Entry-Sequenced** | **Relative** | **Clustered** |
|---|---|---|---|
| Data structure | Sequence | Array | Tree |
| Maximum number of records | 2 **36-10 | 2 **32-10 | 2 **64-10 |
| Storage associated with each record | Fixed or variable length | Fixed | Fixed or variable length |
| Record update limitations | Must be less than or equal to size of record being updated | Must be less than or equal to maximum record size | Must be less than or equal to maximum record size |
| Can space occupied by deleted records be reused? | Not without reorganizing the file | Yes | Yes |
| Primary index | Implicit on entry sequence number (ESN) | Explicit on relative slot number (RSN) | Explicit on any field or fields (specified at create time) |
| Optimized for which type of access | Chronological | Direct access by RSN | Access through primary key value |

## 5.2.3 Creating an SFS File

Each record in an SFS file is made up of one or multiple fields. The field definitions from record to record throughout the file are consistent, and the fields must be defined when the file is being created. The fields in an SFS file are specified in an array of record field specifications. Each element of the array is a data structure of type `sfs_recordFieldSpec_t`. Each structure specifies the contents of one field in the record. The following information about a field can be specified:

- The `fieldName` field specifies the name of the field.
- The `fieldType` field specifies the data type of the field.

- The `fieldSize` field specifies the size of the field. The field size applies only to strings, byte arrays, and variable-length byte arrays. For strings and byte arrays, the field size is the actual size of the field; for variable-length byte arrays, it is the maximum size of the field.
- The `collatingLanguage` field is currently unused and must be set to NULL.

Figure 24 on page 99 shows an example how to create an SFS record template.

```
/* CreateFile- Creates inventory file if it does not already exist. */

void CreateFile(sfsFileName, sfsVolumeName)
    char *sfsFileName, *sfsVolumeName;
{
    sfs_fileSpec_t        fileSpec;
    sfs_recordFieldSpec_t fieldSpecArray[2];
    sfs_indexFieldSpec_t  indexFieldSpecArray[1];
    sfs_status_t   status;

    inFunction("CreateFile");

    /* Record template: field description in order */
    fieldSpecArray[0].fieldName = STOCK_NUM_FIELD;
    fieldSpecArray[0].fieldType = sfs_unsignedInt32;
    fieldSpecArray[0].collatingLanguage = NULL;

    fieldSpecArray[1].fieldName = QUANTITY_FIELD;
    fieldSpecArray[1].fieldType = sfs_signedInt32;
    fieldSpecArray[1].collatingLanguage = NULL;

.
.
.
}
```

*Figure 24. Sample Definition of an SFS Record Template*

Table 3 on page 99 shows the data types that can be used to create an SFS record template.

*Table 3. SFS Data Types for SFS Record Template Creation*

| Data Type | Description |
|---|---|
| sfs_unsignedInt16 | Unsigned 16-bit integer |

| Data Type | Description |
|---|---|
| sfs_signedInt16 | Signed 16-bit integer |
| sfs_unsignedInt32 | Unsigned 32-bit integer |
| sfs_signedInt32 | Signed 32-bit integer |
| sfs_unsignedInt64 | Unsigned 64-bit integer |
| sfs_signedInt64 | Signed 64-bit integer |
| sfs_decimal | A decimal number, representing a variable field from 1 to 18 bytes long |
| sfs_float | 32-bit floating point number |
| sfs_double | 64-bit floating point number |
| sfs_string | Fixed-length array of 8-bit characters. The string must be null-terminated. |
| sfs_nlsString | Fixed-length array of 8-bit bytes. The string must be null-terminated. The SFS default collating language, specified when the server was started, is used to collate all fields of type sfs_nlsString. |
| sfs_byteArray | Fixed-length array of unsigned 8-bit bytes |
| sfs_varLenByteArray | Variable length array of unsigned 8-bit bytes with a 4-byte long header |
| sfs_shortVarLenByteArray | Variable length array of unsigned 8-bit bytes with a 2-byte long header |
| sfs_timestamp | An 8-byte field consisting of two 4-byte unsigned integers |

After you have defined the record template structure, you must do the index specification. The index permits access to a record or range of records, based on the value of some field or fields from those records. The value of the fields on which an index is based provides the index key. The primary index of an SFS file defines the physical organization of the records in the file. Each SFS file has one and only one primary index. In addition, SFS files can also have any number of secondary indexes. These secondary indexes provide alternative access paths to the data by allowing different fields in the record to be used as index keys. All secondary indexes are implemented as B-trees. Each secondary index is stored in a separate area with its own storage allocation, and any number of secondary indexes can be dynamically created and deleted. Index names for a file must be unique.

Both primary and secondary index specifications contain an array of index field specifications, where each element of the array describes one of the fields used in the index. The index field specifications refer to the fields in the record specification for the file they will index. The order of the fields in the index specification determines the order in which key comparisons are made. Figure 25 on page 101 shows the structure that defines the index field.

```
typedef struct{
         char  *fieldName;
         sfs_indexFieldOrdering_t indexFieldOrdering;
         } sfs_indexFieldSpec_t;
```

*Figure 25.  Structure of the Index Field Specification*

In this structure, the fieldName field is the name of a field in the record that is to be included in the index. The indexFieldOrdering field can have one of two values, `sfs_ascending` or `sfs_descending`.

Before creating an SFS file, the primary index must be specified. The primary index specification includes the definition of a primary index name and a primary index structure. Figure 26 on page 101 shows this structure where the `numFields` field specifies the number of fields in the primary index and the `fieldSpecArrayP` field specifies an array of specifications describing each field. The `unique` field specifies whether key values in the index must be unique. The `unique` field is ignored for entry-sequenced and relative files.

```
typedef struct{
         unsigned int unique;
         unsigned int numFields;
         sfs_indexFieldSpec_t  *fieldSpecArrayP;}
sfs_primaryIndexSpec_t;
```

*Figure 26.  Primary Index Structure*

A secondary index defines an alternative sequence in which the records of the file can be accessed. A secondary index can be created after the file is created. An index specification for a secondary index, like that for the primary index, contains an array of index field specifications, where each element of the array describes one of the fields used in the index. The order of the fields in the index specification determines the order in which key comparisons are made. To define a secondary index, you must create a structure of type `sfs_secondaryIndexSpec_t` (Figure 27 on page 102) and then call the `sfs_AddSecondaryIndex` function. This structure contains the following information about a secondary index:

- The `active` field specifies whether the index is active (and is therefore updated as the indexed data changes).
- The `unique` field specifies whether duplicate key values are allowed in the index.
- The `alternateRecordSpecP` field specifies an alternate record specification from which to derive the index fields.
- The `excludedKeyP` field specifies a key value. Records with the specified key value are omitted from this index.
- The `numFields` and `fieldSpecArrayP` fields specify the fields on which the index is based.
- The `storageSpec` field specifies storage characteristics of the index area, providing information such as the volume on which this secondary index will be created and the size of the index area.

```
typedef struct{
        unsigned int active;
        unsigned int unique;
        sfs_recordSpec_t  *alternateRecordSpecP;
        sfs_key_t  *excludedKeyP;
        unsigned int numFields;
        sfs_indexFieldSpec_t  *fieldSpecArrayP;
        sfs_storageSpec_t storageSpec;} sfs_secondaryIndexSpec_t;
```

*Figure 27. Structure sfs_secondaryIndexSpec_t to Create Secondary Index*

Figure 28 on page 103 shows an example of how to define and create a relative SFS file, by using a primary index only. You can create an SFS file programmatically as well as administratively, using the sfsadmin tool. You can also use either method to perform other administrative operations, including copying files, creating new indexes, and getting status information about files.

```
/* CreateFile- Creates inventory file if it does not already exist. */

void CreateFile(sfsFileName, sfsVolumeName)
    char *sfsFileName, *sfsVolumeName;
{
    sfs_fileSpec_t        fileSpec;
    sfs_recordFieldSpec_t fieldSpecArray[2];
    sfs_indexFieldSpec_t  indexFieldSpecArray[1];
    sfs_status_t   status;

    inFunction("CreateFile");

    /* Record template: field description in order */
    fieldSpecArray[0].fieldName = STOCK_NUM_FIELD;
    fieldSpecArray[0].fieldType = sfs_unsignedInt32;
    fieldSpecArray[0].collatingLanguage = NULL;

    fieldSpecArray[1].fieldName = QUANTITY_FIELD;
    fieldSpecArray[1].fieldType = sfs_signedInt32;
    fieldSpecArray[1].collatingLanguage = NULL;

    /*  Index key information */
    indexFieldSpecArray[0].fieldName = STOCK_NUM_FIELD;

    /* File specification */
    fileSpec.fileOrganization = sfs_relative;
    fileSpec.recordSpec.numFields = 2;
    fileSpec.recordSpec.fieldSpecArrayP = fieldSpecArray;
    fileSpec.primaryIndexName = STOCK_NUM_INDEX;
    fileSpec.primaryIndexSpec.unique = TRUE;
    fileSpec.primaryIndexSpec.numFields = 1;
    fileSpec.primaryIndexSpec.fieldSpecArrayP = indexFieldSpecArray;
    fileSpec.storageSpec.volumeName = sfsVolumeName;
    fileSpec.storageSpec.allocated = INITIAL_FILE_SIZE;
    fileSpec.maxNumberRecords.high = 0;
    fileSpec.maxNumberRecords.low = MERCHANDISE_TBL_SIZE;

    status = sfs_CreateFile(sfsFileName, &fileSpec);
    if (status != SFS_FILE_NAME_EXISTS)
        CHECK_STATUS(status);
}
```

*Figure 28. Creating a Relative SFS File*

### 5.2.4 Opening an SFS File

Before any operation on the contents of an SFS file can be performed, you must open that file. To open an SFS file, follow these steps:

1. Prepare a complete description of the way in which you want to open the file.
2. Call the `sfs_OpenFile` function.

The attribute fields of an OFD specification describe the following (see Figure 29 on page 104):

- Access mode
- File access authority
- Consistency level
- Isolation level
- Maximum operation time
- File closing behavior
- Duplicate detection behavior

An OFD specification is a structure of type `sfs_ofdSpec_t`.

```
typedef struct{
         sfs_accessMode_t accessMode;
         unsigned long authority;
         sfs_consistency_t consistency;
         sfs_isolationLevel_t isolationLevel;
         unsigned long operationTimeout;
         unsigned int autoClose;
         sfs_duplicateDetection_t duplicates;
         boolean operationalForce;} sfs_ofdSpec_t
```

*Figure 29. Open File Descriptor Structure sfs_ofdSpec_t*

Below we briefly describe the attribute fields. See the *Encina SFS Programming Guide* for more detailed information.

- The accessMode field specifies whether the OFD requires exclusive access to the file (sfs_exclusiveAccess) or can share access to the file (sfs_sharedAccess).

- The authority field specifies the operations that can be performed on the OFD: some combination of read, insert, update, delete, inquire, and administer.
- The consistency and isloationLevel fields specify whether the OFD is to be used for transactional or nontransactional access and the extent to which access to records through this OFD is isolated from concurrent access by other users.
- The operationTimeout field specifies the timeout period, in seconds, for operations using the OFD.
- The autoClose field specifies whether the OFD should be automatically closed at the end of a transaction. If this field is set to TRUE, the file must be opened from within a transaction and is automatically closed when that transaction ends. If this field is set to FALSE, the OFD can be used by successive transactions. OFDs that can be reused by successive transactions are known as *reusable OFDs*.
- The duplicates field specifies the degree of duplicate detection provided by the OFD for active, nonunique indexes.
- The operationalForce field species whether all of the changes related to each operation made using this OFD are committed to disk before control is returned to the user.

You use the `sfs_OpenFile` function to open a file and obtain an OFD for it. Figure 30 on page 106 shows an example of creating an OFD specification and opening a file. Note that the file is being opened with transactional consistency and that autoClose is set to TRUE; thus, the file must be opened from within a transaction, and the OFD will be automatically closed when the transaction ends.

```
/*  Set up OFD specification and open file */
         ofdSpec.accessMode = sfs_sharedAccess;
         ofdSpec.authority = SFS_READ_FILE | SFS_INSERT_FILE;
         ofdSpec.consistency = sfs_transactional;
         ofdSpec.isolationLevel = sfs_serializability;
         ofdSpec.operationTimeout = OPERATION_TIMEOUT;
         ofdSpec.autoClose = TRUE;
         ofdSpec.duplicates = sfs_noDetection;

         transaction {
              sfs_OpenFile(fileName, &ofdSpec, OPEN_TIMEOUT, &ofd);
                ...
         } onAbort {
                fprintf(stderr, "Could not open file %s; %s\n",
                        fileName, abortReason());
                exit(1);
         }
```

*Figure 30. Example of an OFD Specification and File Open*

### 5.2.5 Performing I/O on an SFS File

Records in an SFS file can be accessed either randomly or sequentially. Both methods search an index using a specified key to select the record or range of records to process. For random access, an index is used to locate a single record that matches an index key value. Random access can be used only if the index is unique, which means that duplicate key values are not allowed in the index. Sequential access involves selecting multiple records by using key values and then sequentially stepping through those records. Selecting multiple records by using an index is known as selecting a range of records. A range of records is selected by specifying key values that bind the records you want to select, or by specifying an individual key value for which all matching records should be selected. An entire file can be selected for sequential processing if the boundary values supplied are the predefined constants for the beginning and ending records in a file.

SFS provides functions for read, write, update, modify, and insert operations in both sequential and random access modes.

#### 5.2.5.1 Sequential Access to Records

During sequential processing, SFS uses a logical record pointer called the *current record* pointer (CRP) to track which record in a selected range has just been processed. Before you select a range of records, the CRP is undefined. You define the CRP by using one of the SFS selection functions. These functions take parameters that specify which records to include in the range as well as a parameter that specifies whether the CRP initially will be positioned at the beginning or end of the selected range of records. You must use the `sfs_read` function to position the CRP. With this function you can also specify that the record next or previous to the CRP be read.

The CRP is moved by a successful sequential read when the selector parameter is set to sfs_next or sfs_previous. An unsuccessful read will not move the CRP unless one of the following two status codes is returned:

- **SFS_INSUFFICIENT_BUFFER** - If this status code is returned by a call to the `sfs_Read` function, it indicates that a buffer supplied to the sfs_Read call was not large enough to hold the entire record. In this case, the portion of the record that could be fit in is returned, and the CRP is moved such that sfs_current refers to the partially read record.
- **SFS_END_OF_KEY_RANGE** - When this status code is returned by a call to the `sfs_Read` function, the CRP is placed before or after the selected range, depending on whether you just tried to read before the first record or after the last record in the range.

Modifying the record does not change the CRP. Sequentially accessed records can be selected in one of three ways:

- The entire file can be selected and processing can begin at a specified place in the file.
- A record can be selected by a specified key value.
- A range of records between two specified key values can be selected.

If the OFD used in a call to any of the functions that position the CRP is a transactional OFD, the range must be selected from within the scope of a transaction. You can find more information on record access methods in the *Encina SFS Programming Guide*.

#### 5.2.5.2 Random Access to Records

You can read, update, or delete records anywhere in a file according to their key value. You must supply the name of a unique index and the value of a key in that index. The key value supplied to any random access function that selects a single record must be fully specified and must uniquely identify a single record. If you want to access records randomly, using a nonunique index, you must use a two-step process. First select a range (using the `sfs_SelectSingleKeyRange` function), then access the records in that range. Random access operations are independent of any sequential access operations; they do not affect the range or the CRP. For example, if you perform a sequential read, then a random read, the CRP still points at the location of the sequential read.

Randomly accessed records can be modified by either updating the record or changing a field in a record. With the `sfs_UpdateByKey` function, the record can be updated by identifying its key value. During this operation a write lock is automatically obtained on the record.

The `sfs_ModifyFieldByKey` function only modifies a single field; the rest of the record is left untouched. The key value used when calling `sfs_ModifyFieldByKey` must be fully specified and must uniquely identify a single record. By default, a write lock is automatically obtained on the record being modified. However, it is possible to set an increment lock instead of a write lock. Figure 31 on page 109 is an extract from the *Encina SFS Programming Guide* and shows how to use the `sfs_ModifyFieldByKey` function to decrement an inventory.

```
telshop_status_t merch_OrderItem(stockNum, quantity)
       long int stockNum;
       long int quantity;
     {
         sfs_key_t           key;
         long int           newQuantity;
         sfs_ofd_t           ofd = tranOfd;
         long int            orderQuantity = -quantity;
         telshop_status_t    returnStatus = TELSHOP_SUCCESS;
         sfs_status_t        status;

         inFunction("merch_OrderItem");

         /* Start a subtransaction so we can back it out */
         /* if user orders too many. */
         transaction {
            /* Set up packed key. */
           key.keyFormat = sfs_packed;
           key.keyRecord.format = sfs_contiguous;
           key.keyRecord.buffer = (sfs_pointer_t)&stockNum;
             key.keyRecord.bufferLength = IGNORED;
           key.keyRecord.dataLength = sizeof(stockNum);

           /* Decrement the inventory. */
             status = sfs_ModifyFieldByKey(ofd, "stockNumIndex", &key,
                           "quantity", sfs_modifyAdd, FALSE,
                         (sfs_pointer_t)&orderQuantity,
                         (sfs_pointer_t)&newQuantity);

          if (newQuantity < 0){
            returnStatus = TELSHOP_ILLEGAL_QTY;
            abort(TELSHOP_ABORT_MODIFY_FAILED);
         }
           /* Handle other status codes. */
           ....
          } onAbort {
             ...
         }
         return returnStatus;
     }
```

*Figure 31. Using sfs_ModifyFieldByKey to Decrement an Inventory*

#### 5.2.5.3 Adding Records to an SFS File

The `sfs_Insert` function inserts a new record into an SFS file. The insertion position of a record in a file is determined by the primary index, so the CRP is not used and is not affected. The `sfs_Insert` function never overwrites an existing record or modifies the CRP. SFS determines the location of the new record on the basis of the key value and the file organization:

- If the file is a clustered file, the record's primary key value determines its location in the file.
- If the file is an entry-sequenced file, the new record is inserted at the current end-of-file. The end-of-file is not transactional maintained, that is, it corresponds to the record after the highest current ESN. Because multiple transactions are allowed to concurrently insert records into an entry-sequenced file, records belonging to different transactions can be interleaved.
- If the file is a relative file, the primary key value (that is, the value of the field that contains the RSN) determines the position of the record in the file. For relative files, specifying the value `SFS_FIRST_AVAILABLE_SLOT` causes the first available slot in the file to be reused; thus, space consumption is optimized. Specifying `SFS_AFTER_LAST_OCCUPIED_SLOT` causes the record to be written at the end of the file.

For a description of other I/O functions that help you manage keys, see *Encina SFS Programming Guide*.

### 5.2.6 SFS File Access and Transactions

When you open a file, you specify whether you want a transactional OFD or a nontransactional OFD. Transactional OFDs must be used within the scope of a transaction. Any operations that use a nontransactional OFD do not participate in any user transaction. Locks obtained by transactional OFDs are held on behalf of a transaction. That is, if an application obtains two transactional OFDs (ofd1 and ofd2), and ofd1 write locks a record, ofd2 can still access that record. In general, locks held on behalf of a transaction are held until the transaction completes. A transactional OFD cannot be used by another transaction family until the transaction it is currently associated with commits or aborts or until it is explicitly dissociated from the transaction.

Locks obtained by nontransactional OFDs are held on behalf of that OFD. That is, if an application obtains two nontransactional OFDs (ofd1 and ofd2), and ofd1 write locks a record, ofd2 cannot access that record. Locks held on behalf of a nontransactional OFD are held for a duration based on the OFD's isolation level.

Transactional and nontransactional access to the same file is possible. The type of file access required by an application depends on the function and reliability requirements of the application. An application may, for example, want to perform processing applications transactionally while performing administrative operations nontransactional. This application could simultaneously have two OFDs open on the file: a transactional OFD to process the data in a file and a nontransactional OFD to perform administrative operations on that same file.

## 5.3 Encina RQS

The Encina RQS enables applications to transactionally enqueue and dequeue data. RQS supports flexibility in system configuration, reliability based on the Encina transaction processing environment, large capacity for enqueued data and number of elements, and concurrency for many users. An application can store data related to a task in a queue. This data can be subsequently processed by another program. This off-loading may be desirable when use of a resource incurs an unacceptable time penalty during peak usage hours, when one part of a transaction can take much longer than other parts, or when a resource is temporarily unavailable. For example, the confirmation of a sale can be completed in real time and the data associated with the sale can be stored in an RQS queue for later processing.

A queue is a linear data structure and holds elements that are added (enqueued) to the tail of a queue and removed (dequeued) from the head of the queue in an FIFO manner. Similar to SFS each queue is maintained by one RQS server. All interactions with that queue are handled by the server. An RQS server may contain multiple queues to be accessed by different functions or applications. Applications store data into a queue in the form of elements. An element has a record-oriented format defined by the application. The fields of an element store the related pieces of data. For example, a billing element might have fields for storing the customer name, customer account number, and current account balance.

Each element must have a type, which is specified when the element is added to a queue. 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 types can be queued and dequeued from the same queue. An element type also defines an element’s keys. An element key is a sequence of one or more element fields to be used as a basis for retrieving the element. An element type need not have any associated element keys.

Queues support multiple simultaneous requests to enqueue and dequeue elements, growing and shrinking in size according to the volume of requests. An RQS server tracks a variety of statistics on queue activity, such as processed and new element count, mean waiting time, and physical queue size, for a collection period and for the lifetime of the queue. It can also track the work of a queue. The work is the volume of business represented by a queue, for example, dollars transferred or tons produced. A queue can keep a running sum of the work as elements are enqueued and dequeued. Thus it is easy to oversee and control an application’s infrastructure.

An application may requeue an element to another queue for subsequent processing by another application. Requeuing is the process of moving an element from one queue to another. When an element is dequeued by an application, that application indicates its intent to requeue the element by identifying it as an orphan. An orphan is an element that has been dequeued but not yet requeued. This is usually a transitory state; elements do not remain orphans beyond the scope of a primary transaction.

Applications that select from several different queues when processing dequeue requests can use queue sets to simplify the selection process. A queue set is a collection of queues. A queue can belong to more than one queue set. A queue that belongs to a queue set can be accessed as part of that queue set or can be accessed individually. Each queue in a queue set is assigned to a priority class, which ranks the queue (or a group of queues) in importance relative to other queues in the same queue set: "priority 1" is higher than "priority 3" in the same queue set. Each priority class has an associated service level, which defines how to distribute the dequeuing service among the priority classes in the set. A service level is associated with each priority class in a queue set to establish a weighted prioritization of the member queues. The priority classes and service levels collectively define the selection process for the queue set.

### 5.3.1 Operations on Queues

RQS is built on top of the Encina Toolkit Server Core and the Encina Toolkit Executive. It provides higher level functions for administering and manipulating RQS information. The transactional services required by RQS are transparent, and you do not have to be familiar with most of the details of DCE or the Encina toolkit.

The main operation functions used in an RQS application are:

- Enqueuing
- Dequeuing

- Batch request (multiple queue requests)
- Requeuing
- Random access to elements through element IDs
- Scanning elements with cursors
- Retrieving elements with keys

#### 5.3.1.1 Enqueuing Elements

An application enqueues an element by identifying a target queue and providing element data. The application must supply the type of the element. This type must be defined before an element is enqueued. If the queue has work accumulation enabled, the client must also provide a work value. For each enqueue request, the RQS server creates an element to hold the data, enqueues the element, and returns the element's newly created unique identifier. Applications can enqueue elements in single requests or combine multiple enqueue requests in a single operation to minimize RPCs.

Figure 32 on page 113 shows the `rqs_Enqueue` function that is used to enqueue a single element.

```
rqs_status_t rqs_Enqueue (

         IN rqs_serverHandle_t server,

         IN char  *queue,

         IN char  *elementType,

         IN unsigned long valueLen,

         IN rqs_pointer_t value,

         IN rqs_unsignedInt64_t  *workP,

         OUT rqs_elementId_t  *eltIdP);
```

*Figure 32. rqs_Enqueue Function*

The `server` argument specifies the RQS server handle that has been specified at initialization. The `queue` string names the queue to enqueue the element. The `elementType` specifies the element type name. The `valueLen` argument specifies the length in bytes of the element data in the `value` parameter. The `value` parameter is a packed buffer that adheres to the specified element type. If work accumulation is enabled for this queue, `workP` points to the work quantity; if a null pointer is supplied, the work value is treated as zero. If work accumulation is not enabled, the `workP` argument is ignored. The `eltIdP` argument returns the new element's unique ID. The application can then use this ID to access or manipulate the data in that element. Element IDs are unique to an RQS server; an RQS server will never reissue an element ID, even if the element with which it was originally associated is destroyed. Once an application dequeues or deletes the element without requeuing it, the element's ID becomes invalid.

#### 5.3.1.2 Dequeuing Elements

An application can dequeue a single element from either a specific queue or a queue set. To dequeue an element from a queue, an application uses the `rqs_Dequeue` function. To dequeue an element from a queue set, it uses the `rqs_QSDequeue` function. Whenever an application dequeues an element from either a specific queue or a queue set, it must specify whether that element should be deleted from the server when it is dequeued. It must also specify what to do if no elements are available for dequeuing. Figure 33 on page 114 shows the `rqs_Dequeue` function.

```
rqs_status_t rqs_Dequeue(
         IN rqs_serverHandle_t server,
         IN char *queue,
         IN rqs_elementDeleteOption_t deleteOption,
         IN rqs_boolean_t blockOnEmpty,
         OUT rqs_elementDescriptor_t **elementPP)
```

*Figure 33. rqs_Dequeue Function*

The `deleteOption` argument specifies whether to delete the element from the RQS server. It can have one of the following values:

- If deleteOption is `rqs_deleteElement`, the element is deleted from the server. The element's lifetime expires, and its ID becomes invalid.
- If deleteOption is `rqs_orphanElement`, the element becomes an orphan when it is dequeued; that is, it remains at the server but is not part of any queue. An orphan element can subsequently be requeued by the dequeuing transaction, any nested transactions, or any other transactions in the transaction family that commit with respect to the dequeuing transaction. If the element remains an orphan until the entire transaction family commits, the element is deleted from the server.

The `blockOnEmpty` argument specifies what to do if there are no elements to dequeue from the queue or queue set. If this argument is true, the `rqs_emptyDequeueTimeout` time-out class applies; the function waits for the time-out period for an element to appear. If `blockOnEmpty` is false, the `rqs_operationAccessTimeout` time-out class applies. In this case, this function returns a status code indicating that no element is available, rather than waiting for elements to appear in the queue.

Figure 34 on page 116 shows sample code, extracted from the *Encina RQS Programming Guide,* which demonstrates the specific dequeue process from within a Tran-C transaction.

Figure 35 on page 117 shows sample code that calls the rqs_QSDequeue function. The sample code dequeues with the rqs_orphanElement delete option so that other code in the sample shipping application can requeue the element.

```
static void BillClient(rqs_serverHandle_t rqsHandle)
     {
         rqs_status_t status;
         rqs_elementDescriptor_t *elementP = NULL;

         /* Bill the customer */
         transaction {
             /* Test billing queue for client orders to be billed.
              * Dequeue element from the BillingQ; block if queue empty.
              */
             status = rqs_Dequeue(rqsHandle, BILLING_QUEUE,
                                  rqs_deleteElement,
                                  TRUE, /* Blocking dequeue */
                                  &elementP);
             if (status == RQS_TIMEOUT_EMPTY_DEQUEUE) {
                 /* No orders currently in billing queue. */
             } else {
                 /* Other errors fatal */
                 CHECK_STATUS(status);
             }
         } onCommit {
             merchandise_shipBillRecord_t *billRecordP;

             if (elementP) {
                 billRecordP =
                      (merchandise_shipBillRecord_t *)elementP->value;

                 printf("Billed customer %s for %d of item %d.\n",
                        billRecordP->customer, billRecordP->quantity,
                        billRecordP->stockNum);
             }
         } onAbort {
             printf("Billing transaction aborted: %s (%s)\n",
                    abortReason(), abortModuleName());
         }

         /* Clean up */
         if (elementP)
             rqs_Free(elementP);
     }
```

*Figure 34. Dequeuing an Element from a Specific Queue*

```
static rqs_elementDescriptor_t *ShipItem(rqs_serverHandle_t *rqsHandle)
     {
         rqs_elementDescriptor_t *elementP = NULL;
         rqs_status_t status;
         merchandise_shipBillRecord_t *billRecordP;

        /* Dequeue an item from a customer order; block if queue empty.
*/
         status = rqs_QSDequeue(rqsHandle, SHIPPING_QSET,
rqs_orphanElement,
                                TRUE, /* Blocking dequeue */
                                &elementP);

         switch (status) {
           case RQS_SUCCESS:
             /* If an item was dequeued (shipped), record the shipping
             /* time. */
             billRecordP=(merchandise_shipBillRecord_t \
                          *)elementP->value;
             /*
              * NOTE: the gettimeofday function is UNIX-specific.
              * NT applications can define a _timeb structure
              * and use the _ftime function.
              */
             gettimeofday((struct timeval *)&billRecordP->shipTime,
                        NULL);
             break;
           case RQS_TIMEOUT_EMPTY_DEQUEUE:
             /* No orders currently in shipping queue set. */
             break;
           default:
             /* Other errors are fatal. */
             CHECK_STATUS(status);
             break;
         }

         /* Return the dequeued (shipped) element, if any. */
         return elementP;
     }
```

*Figure 35. Dequeuing from a Queue Set*

#### 5.3.1.3 Requeuing Elements

If an application specifies a value of `rqs_orphanElement` for the `deleteOption` parameter when it dequeues an element, it can requeue that element into any queue within the same server in one of two ways:

- It can dequeue the element from one queue and requeue it to the same queue or to another queue, using the `rqs_Requeue` function.
- It can dequeue the element and then modify and requeue it, using the `rqs_RequeueAndModify` function.

**The requeuing transaction must commit with respect to the transaction that dequeued the element!**

Figure 36 on page 118 shows the `rqs_Requeue` function.

```
rqs_status_t rqs_Requeue (
         IN rqs_serverHandle_t server,
         IN char  *queue,
         IN rqs_elementId_t  *orphanP,
         IN rqs_unsignedInt64_t  *workP);
```

*Figure 36. rqs_Requeue Function*

The `queue` argument names the queue to which the RQS server requeues the element, and `orphanP` identifies the element. The `rqs_Requeue` function takes a work argument because an element's associated work value has a meaning based on the convention of the queue containing that element. The associated work can have different meanings in different queues. If work accumulation is enabled for this queue, the `workP` argument points to the work quantity. If a null pointer is supplied, the work value is treated as zero. If work accumulation is not enabled, the `workP` argument is ignored.

Figure 37 on page 119 shows the `rqs_RequeueAndModify` function used to requeue and modify an element.

```
rqs_status_t rqs_RequeueAndModify (
         IN rqs_serverHandle_t server,
         IN char  *queue,
         IN rqs_elementId_t  *orphanP,
         IN rqs_unsignedInt64_t  *workP,
         IN char  *elementType,
         IN unsigned long valueLen,
         IN rqs_pointer_t value);
```

*Figure 37. rqs_RequeueAndModify Function*

This function combines the functionality of the `rqs_Requeue` function and the `rqs_ElementModify` function into a single operation. The arguments of the `rqs_RequeueAndModify` function conform to the same constraints as the arguments of these two functions. An application can change the type of an element and the element's value.

#### 5.3.1.4 Random Access to Elements through Element IDs

When an application enqueues an element, the enqueuing function returns an element ID to the application. Whenever an application has a valid element ID, it can access and manipulate the element directly, as opposed to accessing the element through a key, cursor, or dequeue operation. With direct access to an element through its unique element ID, an application can read, modify, or delete the element. It can also drop locks held on the element.

**Note**

When accessing an element with an element ID, the application does not specify a queue. Element IDs are unique at the server, and thus the application can access elements through element IDs independent of a queue.

***Reading Elements***

To read an element by element ID without dequeuing that element, use the `rqs_ElementRead` function. This function returns a pointer to an `rqs_elementDescriptor_t` structure and allows you to specify a lock mode and the period of time for which that lock should be held. Figure 38 on page 120 shows the `rqs_ElementRead` function.

```
rqs_status_t rqs_ElementRead (
         IN rqs_serverHandle_t server,
         IN rqs_elementId_t  *eltIdP,
         IN rqs_elementLockMode_t eltLock,
         IN rqs_boolean_t retainLock,
         OUT rqs_elementDescriptor_t **elementPP);
```

*Figure 38. rqs_ElementRead Function*

The `eltIdP` argument specifies the element ID of the element to read. The `eltLock` argument specifies the lock mode required before reading the element; possible values are `rqs_noLock`, `rqs_readLock`, `rqs_upgradeLock`, and `rqs_writeLock`. The `retainLock` argument indicates whether the server should retain the lock beyond the duration of this function call, which means the lock is held for the duration of the calling transaction as long as the application does not explicitly drop the lock.

### Modifying Elements

The `rqs_ElementModify` function is used to modify an element's data or change its associated element type. Figure 39 on page 120 shows the function.

```
rqs_status_t rqs_ElementModify (
         IN rqs_serverHandle_t server,
         IN rqs_elementId_t  *eltIdP,
         IN char  *elementType,
         IN unsigned long valueLen,
         IN rqs_pointer_t value);
```

*Figure 39. rqs_ElementModify Function*

The `elementType` argument specifies the data element type. The `valueLen` argument is the length of that element data. An application can change the type of the data associated with the element by specifying a different element type name in the elementType field. However, the element's data must be compatible with the data in the new element type name.

### Deleting Elements

The `rqs_ElementDelete` function is used to delete an element and optionally to make it an orphan. Figure 40 on page 121 shows this function.

```
rqs_status_t rqs_ElementDelete (
        IN rqs_serverHandle_t server,
        IN rqs_elementId_t  *eltIdP,
        IN rqs_elementDeleteOption_t deleteOption,
        OUT rqs_elementDescriptor_t **elementPP);
```

*Figure 40. rqs_ElementDelete Function*

The deleteOption argument specifies whether to delete the element or to make it an orphan. The elementPP argument provides a location for the function to return a pointer to an element descriptor. The returned element descriptor reflects the state of the element immediately before the time of the call, and the containingQueue in this element descriptor is the name of the queue that held the element. If the element was an orphan at the time of this call, the containing queue field is NULL and the work field contains a zero work value.

### *Element Lock Droppings*

The rqs_ElementDropLock function is used to drop a lock held on an element. Figure 41 on page 121 shows this function.

```
rqs_status_t rqs_ElementDropLock (
        IN rqs_serverHandle_t server,
        IN rqs_elementLockMode_t eltLock,
        IN rqs_elementId_t  *eltIdP);
```

*Figure 41. rqs_ElementDropLock Function*

This function drops one instance of a lock of mode eltLock that is held on the specified element. The lock must have been acquired in a function call made within the scope of the calling transaction. It is impossible to drop a lock acquired by a parent, sibling, or child transaction. If the calling transaction does not hold the indicated lock on the element, this function returns the RQS_DROPLOCK_NONE status code. If the lock mode is rqs_writeLock, this function returns the RQS_DROPLOCK_WRITE status code and does not drop any locks. If the lock mode is rqs_noLock, this call has no effect.

The rqs_ElementListDropLocks function can be used to drop multiple locks simultaneously. It also can be used to drop all instances of a lock held in a particular mode on an element.

***Comparing and Converting Element IDs***
An element's ID is guaranteed to be unique within a server for all time. An RQS server can always determine whether an element ID it generated is currently valid. Client applications are responsible for ensuring that they always use element IDs with the server that generated them. Using an element ID with an RQS server other than the server that originally generated the element ID has undefined results. Figure 42 on page 122 shows the `rqs_ElementIdCmp` function being used to identify whether or not two element IDs point to the same element.

```
int rqs_ElementIdCmp(
         IN rqs_elementId_t  *eltId1P,
         IN rqs_elementId_t  *eltId2P);
```

*Figure 42. rqs_ElementIdCmp Function*

This function returns zero if the IDs identify the same element. When the elements are distinct, this function indicates which one was created first at the server. A return value of -1 indicates that eltId1 was created before eltId2, and a return value of 1 indicates that eltId2 was created before eltId1. This function does not check the validity of the element IDs.

With the `rqs_ElementIdToString` function, an element ID can be converted to text, which thereafter can be used to generate audit trails and program status reports. This function does not check the validity of the element ID.

#### 5.3.1.5 Scanning Elements with Cursors
A cursor is a logical, client-side object that is used to sequentially examine the elements in a queue. Cursors are owned by a specific transaction when in use, and only the owning transaction can use the cursor. Cursors have a lock mode and locking policy associated with them. As the cursor advances, the RQS server acquires element locks in the cursor's mode on behalf of the owning transaction. The locking policy for each cursor determines the duration for which the server holds the lock. The cursor locking policies supported by RQS enable locks to be held for three different periods of time:

- For the duration of the advancing operation
- Until the client advances the cursor again
- For the duration of the transaction that owns the cursor

Each policy provides a different guarantee about whether the element data returned to the client remains the same as the data for that element stored at the RQS server.

Cursors are client-side objects and can therefore be shared between transactions executing within the same application; they cannot be shared by applications. Cursors are *neither persistent nor recoverable*. If the application that creates a cursor terminates, that cursor ceases to exist. A transaction can simulate recoverable cursors by copying an existing cursor at the beginning of the transaction, using the new copy, and returning to the original cursor if it becomes necessary to recover to a previous location in the queue.

Only one transaction can own and use a cursor at one time. A transaction acquires cursor ownership by creating or copying a cursor within the scope of that transaction. If the cursor is created outside the scope of a transaction, the cursor is marked as "unowned," and a transaction can obtain ownership by specifying that cursor when calling functions that require a cursor. When an application creates a cursor, it specifies a lock mode and the locking policy for the cursor. When a client calls a function that advances a cursor, that operation moves through the specified queue, starting at the cursor’s current location, trying to acquire a lock in the cursor’s lock mode on each element. When an element can be locked in the specified mode, the function returns an `rqs_elementDescriptor_t` structure. The server holds the lock according to the locking policy specified for that cursor.

Cursors never refer to a particular element in a queue; a given cursor is located between elements, before the first element, or after the last element. When an application creates a cursor, it is located before the first element in the queue. When an application advances a cursor, the cursor moves past the first element on which it could acquire an appropriate lock and then returns a descriptor for that element. Cursors maintain their relative positions as applications dequeue, delete, and effectively insert surrounding elements (the latter when a dequeuing or deleting transaction aborts). A cursor never reads from a position in the queue past which it has already advanced unless the cursor is reset to the beginning of the queue.

For more detailes on using RQS cursors, see the *Encina RQS Programming Guide*.

#### 5.3.1.6  Retrieving Elements with Keys

RQS provides a mechanism for key-based retrieval of elements. When an element type is created, once or more keys can be declared for that element type. When retrieving an element using a key, the application does not specify a queue. Retrieving elements using keys is independent of the queue. Therefore, the element can be on any queue on the server. For more information about retrieving elements with keys, see the *Encina RQS Programming Guide*.

## 5.3.2 RQS Application Structure

RQS applications are clients to one or more RQS servers. These applications may also be servers themselves, publishing an interface and receiving RPCs from other applications. As RQS clients, RQS applications have the following basic life cycle:

1. Initialize Encina client components.
2. Look up an RQS server, using the DCE Directory Service, and obtain a binding handle to a server.
3. Perform whatever work or user interaction for which the application was designed and interact with RQS. The RQS server interaction involves enqueuing elements, dequeuing elements, administrative modifications to the server, performance querying, and other activities.
4. Terminate the application, closing and freeing the RQS server handle.

### 5.3.2.1 Initializing

RQS itself does not need to be initialized! However, to initialize an RQS application, all of the underlying Encina components must be properly initialized. If the application is running in the Monitor environment, underlying Encina components are initialized automatically. If not running in the Monitor environment, the application must initialize the underlying Encina components. For example, if the application is using Tran-C to manage transactions, it must initialize Tran-C.

### 5.3.2.2 Look up an RQS Server

When you develop general Encina clients using Tran-C, several steps are required to look up a server in the DCE Directory Service, obtain a valid binding handle, and connect to the server. RQS does this work for you, and applications need only call the `rqs_GetServerHandle` function. This function takes the RQS server name to look up in the DCE Directory Service and returns a handle to the specified server. Figure 43 on page 124 shows the `rqs_GetServerHandle` function.

```
rqs_status_t rqs_GetServerHandle(
        IN char *serverName
        OUT rqs_serverHandle_t *serverP);
```

*Figure 43. rqs_GetServerHandle Function*

If the RQS server is unavailable for whatever reason, this function returns the RQS_COMMUNICATION_ERROR status code. If this occurs, the client should delay and try again using the same handle.

**Note**

The Transaction Service aborts transactions involved in communication errors.

Figure 44 on page 125 shows the `Initialize` function from a sample application. This function initializes the application and returns an RQS server handle, which the client uses to communicate with the server.

```
static rqs_serverHandle_t Initialize(char *serverName)
     {
         rqs_serverHandle_t  rqsHandle;
         rqs_status_t status;

         /* Initialize Tran-C and TRPC */
         preInitTC();
         tc_InitTRPC();
         postInitTC();
         /* Get a handle to RQS server */
         status = rqs_GetServerHandle(serverName, &rqsHandle);
         CHECK_STATUS(status);
         return(rqsHandle);
     }
```

*Figure 44. Example of an RQS Client Initialization without Encina Monitor*

#### 5.3.2.3 Terminating an RQS Application

When an RQS application terminates with pending transactions that are unprepared, the termination may cause locks to be held on behalf of the pending transactions in the server. The server holds these locks until an administrator, using the tkadmin tool, manually aborts the transaction in the server.

### 5.3.3 Managing Queues

Queues are typically created by RQS administrators as part of configuring and maintaining an RQS server. Applications can also create and destroy queues. RQS also provides the "work accumulation facility," which is a mechanism for customizing some of the statistics associated with queues.

#### 5.3.3.1 Creating Queues

Applications can create queues with the `rqs_QCreate` function.

```
rqs_status_t rqs_QCreate (
        IN rqs_serverHandle_t server,
        IN char  *queueName,
        IN rqs_workAccumulation_t  *accumState);
```

*Figure 45. rqs_Qcreate Function*

The `queueName` argument specifies the name of the new queue; no existing queue within the server can have this name.

The `accumState` argument must be set to either `workAccumulationEnabled` or `workAccumulationDisabled` and will be used by the work accumulation facility of RQS. These arguments indicate whether or not a queue maintains a cumulative sum of the work represented by the enqueued elements. The work accumulation facility of RQS provides a mechanism for customizing the statistics associated with queues at an RQS server. It enables applications to measure the total volume of business (work) represented by the elements in the queues at a server.

For a more detailed discussion of the work accumulation facility, see the *Encina RQS Programming Guide*.

#### 5.3.3.2 Destroying Queues

Applications destroy queues with the `rqs_QDestroy` function. Only the server and the queue name need to be defined. After this function call, the queue no longer exists. The RQS server removes the queue from any queue sets of which it is a member. The elements in the queue are destroyed and their IDs become invalid. The elements can no longer be retrieved. Once a queue is destroyed, a new queue can be created by using the same name of the destroyed queue.

To remove all elements in a queue without destroying that queue, use the `rqs_DeleteAllElements` function. As with `rqs_QDestroy`, this function needs only the server and the queue name defined. `rqs_DeleteAllElements` does not modify any of the authentication, authorization, or statistical information associated with a queue.

### 5.3.4 Managing Queue Sets

RQS supports queue sets. The following aspects must be managed on queue sets:

- **Setting priority classes** - Priorities define the ordering among queues within a queue set. Whenever a queue is selected from a priority class within a queue set, queues in that particular priority class are selected in a round-robin fashion.
- **Setting service classes** - A service level defines how dequeuing requests are to be distributed among the priority classes; in other words, how many dequeuing requests have to be executed in a specific queuing class.
- **Creating queue sets** - Group queues into queue sets.
- **Adding queues to queue sets** - Add queues to an existing queue set.
- **Removing queues to queue sets** - Remove a queue from a queue set.
- **Destroying queue sets** - Destroy a queue set with all its members.
- **Maintaining queue statistics** - RQS maintains information about each queue associated with an RQS server, including general properties and usage statistics. This information can be queried for further usage.

## 5.4 Encina PPC

Encina PPC Services enable Encina transaction processing systems to interoperate with systems that have SNA LU 6.2 communication interfaces. PPC Services support both the X/Open CPI-C and the IBM SAA CPI-C. PPC Services also support SAA CPI-RR. These communication interfaces and the distributed transaction processing model operate within the Encina environment.

PPC Services offer the following features:

- Integration and migration between mainframe and Encina environments. Encina transaction processing systems can interoperate with systems using SNA. PPC Services enable bidirectional communication, so that both applications and data can be shared between mainframes and Encina, with either side initiating communications.
- LU 6.2 connectivity to the Encina Monitor. Thus an application running on a SNA network can allocate a conversation through the PPC SNA Gateway to an application in the Encina Monitor.
- Flexible administrative tools that can be used from any platform

- Concurrency. PPC Services provide thread-safe CPI-C and CPI-RR routines for execution in the OSF DCE

PPC Services include two products, the PPC Executive and the PPC SNA Gateway. The PPC Executive is a library that supports peer-to-peer communications and two-phase commit transactional semantics. The PPC SNA Gateway provides communication between Encina applications in a DCE cell and LU 6.2 applications in a SNA network. This cross-network communication is transparent to both peer applications involved.

Figure 46 on page 128 illustrates the PPC Services model for communicating through a gateway server.

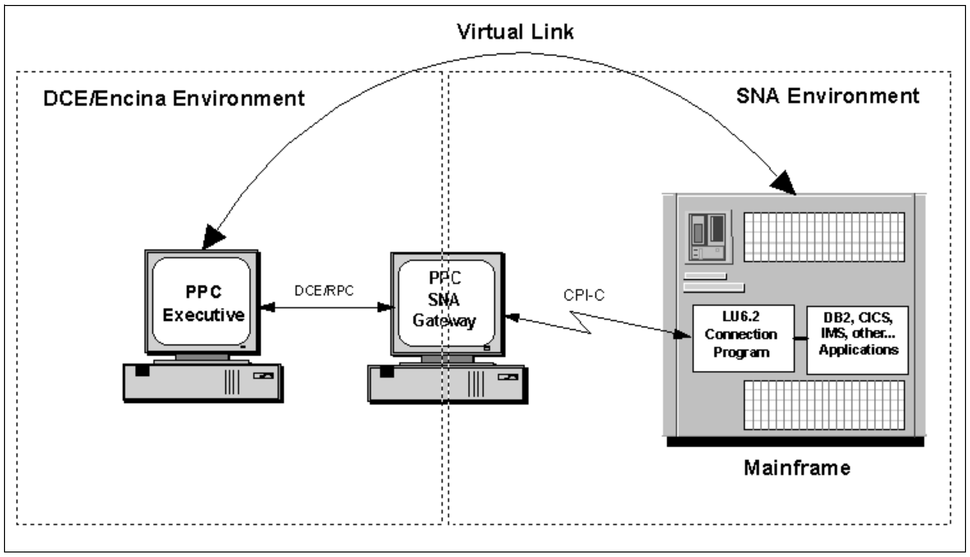


*Figure 46. PPC Services Model*

The gateway server runs on a machine that is part of a DCE cell and a SNA network. The PPC SNA Gateway establishes a virtual link between a SNA LU 6.2 application on a mainframe and a PPC Executive application on a DCE node.

You use the functionality provided by the PPC Executive to create programs (PPC Executive applications) that communicate with other PPC applications, such as those running on mainframes. In general, you need only be aware of the PPC Executive library. The PPC SNA Gateway is transparent to application programs. Only those programs performing certain administrative tasks need be aware of the gateway. Gateway configuration is typically managed by the system administrator.

PPC Executive applications are fully integrated into the Encina DCE environment; that is, an application can communicate with a mainframe, using SNA, while using Encina and DCE to communicate with other applications in the Encina DCE environment. An Encina-to-Encina conversation is between two PPC Executive applications. Encina-to-Encina conversations do not use the gateway server.

The peer-to-peer communication model differs from the client/server model used by the rest of Encina. In the Encina client/server model, a client initiates an RPC to a server and waits for a response. The server receives and processes the RPC, 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 a client, through the server function, and then returns 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. While there is still an originator of a conversation, akin to the client that originates an RPC, once the conversation is established there is no distinction between the roles of the two partners.

PPC applications are written using CPI-C, as specified by IBM SAA and X/Open. CPI-C provides a number of services, including:

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

In addition to CPI-C, IBM SAA specifies the CPI-RR for transaction demarcation. X/Open specifies the TX interface for the same purpose.

### 5.4.1 LU 6.2 Conversations and Synchronization

The logical unit (LU) serves as a port into the network and acts as an intermediary between the end user and the network. The LU is engaged in session establishment with one or more partner LUs and manages the exchange of data with partner LUs. LU types define the sets of functions in an LU that support end-user communication. LU 6.2 is the most flexible LU type and is also known as APPC. The two-way communication of two application programs over an LU 6.2 session is called a conversation. The two application TPNs are partners in a conversation and exchange information. Each LU 6.2 session can carry one conversation at a time. Figure 47 on page 130 shows the basics of the SNA peer-to-peer communication model.

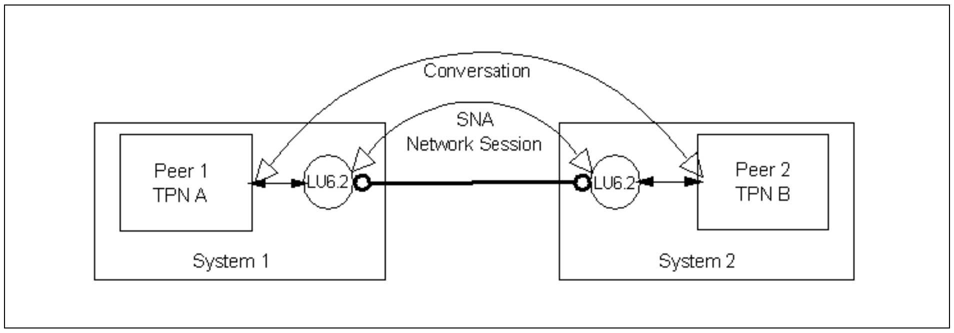


*Figure 47. SNA Peer-to-Peer Communication Model*

To establish a conversation, one program allocates it; that is, it specifies the LU, mode, and TPN with which it wants to communicate. The program that allocates the conversation is called the allocator. The acceptor is the recipient of an allocator's conversation request. It accepts the conversation allocated by another peer. To end a conversation, one peer deallocates it and its peer receives notification of the deallocation. Between allocation and deallocation, the peers can exchange data and do work on each other's behalf.

For a program to allocate a conversation to a peer, it requires certain initialization information, such as the peer's TPN and the name of the partner LU. This information is called *side information*. It is generally supplied and maintained by the system administrator in a side information file, which a program can read in before allocating a conversation. The information is accessed by a symbolic destination name, which is independent of the actual SNA parameters.

There are two types of conversation. Mapped conversations allow programs to exchange arbitrary data records in formats agreed on by the programmers. Basic conversations allow programs to exchange data in a standard format; that is, a 2-byte length field followed by user data.

The synchronization level (synclevel) specifies the degree of synchronization that occurs between peers in a conversation. The LU 6.2 protocol supports three levels of synchronization:

- **synclevel syncpoint** (SL2): Synclevel syncpoint is the highest level of synchronization. It provides transactional conversations, which use the two-phase commit protocol, and rollback and resynchronization capabilities.

- **synclevel confirm** (SL1): Synclevel confirm provides simple synchronization that involves a single message exchange rather than full two-phase commit processing. This confirmation synchronization is provided through a pair of functions that enable applications to explicitly request confirmation and acknowledge the request. Synclevel confirm is used for nontransactional conversations.
- **synclevel none** (SL0): Synclevel none provides no confirmation or syncpoint processing. For example, if a program sends data, it does not receive automatic acknowledgment that its peer received that data. A program cannot issue any calls that require higher synchronization levels. Synclevel none is used for nontransactional conversations.

A synclevel syncpoint conversation works on behalf of a logical unit of work (LUW). An LUW is a transaction, a set of operations that must be executed together. No operations are performed if any one of them is not performed. An LUW is identified by a logical unit of work identifier (LUWID), which is a SNA global transaction identifier. LUWs can be chained; when one ends, another starts automatically. There is always a syncpoint or a backout between two LUWs. LUWs can be completed in one of two ways: they can be committed or aborted. To commit an LUW (that is, to make all changes since the last commit permanent), either peer calls for a syncpoint. A syncpoint is a reference point to which resources can be returned if a failure occurs. To abort an LUW (that is, to undo all changes since the last syncpoint), either peer can back out the LUW.

This model of LUWs differs from the transaction model used in the rest of Encina in which new transactions are not generally started when one ends and in which work can occur outside the scope of the transaction. For synclevel syncpoint conversations, no work occurs outside an LUW. Because of this difference in models, PPC Executive applications using synclevel syncpoint conversations must follow several rules to work correctly in the Encina environment:

- Start a transaction before allocating a conversation.
- End the final transaction after the conversation has been deallocated.

Table 4 on page 132 summarizes how to start an LUW from an Encina transaction.

*Table 4. Tasks to Start Communication from Encina Transactions to LUWs*

| | To Start First Transaction | To Start Last Transaction | Chaining Allowed? |
|---|---|---|---|
| PPC Executive allocators | Use Tran-C | Do nothing (ended by Tran-C) | No |
| | Use TRAN/threadTid or TX | Use TRAN/threadTid or TX | Yes |
| PPC Executive acceptors | Do nothing (transaction is started by the allocator) | Use TRAN/threadTid or TX | Yes |
| PPC Executive allocators inside TRPC server function | Do nothing (transaction is started by the client) | Do nothing | No |

Tran-C differs from TRAN/threadTid and TX in that it cannot chain transactions. The transaction construct delimits a single transaction, which begins at the opening brace and ends at the closing brace. Thus, Tran-C can be used only in applications that have one transaction per synclevel syncpoint conversation. Furthermore, because Tran-C starts the transaction it is working on, applications cannot accept (allocate only) synclevel syncpoint conversations (and thus work on behalf of a transaction started by a peer) in a transaction construct.

Figure 48 on page 133 illustrates the use of PPC Services with Tran-C. The calls to initialize the conversation and set the synchronization level can also be performed outside the transaction construct.

```
transaction  {
          /* Initialize the conv. and set requested synclevel */
         Initialize_Conversation(conversationId, symDestName,
                                 &returnCode);
         PPC_STATUS_CHECK(returnCode);
         Set_Sync_Level(&convId, CM_SYNC_POINT, &returnCode);
         PPC_STATUS_CHECK(returnCode);
         /* Allocate the conversation */
        Allocate(convId, &returnCode);
         PPC_STATUS_CHECK(returnCode);

          /* Exchange data with the other side */
          ...

         /* All synclevel syncpoint conversations in
          * deallocate state. See text. */

     }   /* commit processing takes place here */
```

*Figure 48. Using Tran-C to Allocate a Synclevel Syncpoint Transaction*

To abort the transaction, the application can call the Tran-C abort function. This function also deallocates the conversation. The transaction can also be aborted by simply deallocating the conversation abnormally.

## 5.4.2 Programming Interfaces

PPC applications are written using CPI-C, as specified by IBM SAA and X/Open. These CPI-C standards specify the programming interface used for PPC communications. The PPC Services support both the X/Open and the IBM SAA versions of the standard. The IBM SAA and X/Open interfaces are nearly identical. The functions provide the same functionality, use the same arguments, and generally return the same status codes. CPI-C provides a number of services, including:

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

An application can accept one conversation and allocate another and so forth. Any peer in such an allocation tree can call for a syncpoint provided that all of its conversations are in a send state. The commit is then received by all of its peers, which in turn call Commit to propagate the commit to their peers.

Most Encina PPC Executive applications use the interface as defined in the cpic.h header file, which also defines Encina-specific functions. There are a few minor differences between the interface defined in this file and the interface defined in the latest specifications. The primary differences are:

- In the specifications, all functions return status by using an OUT parameter to the function. As defined in cpic.h, functions also return status in this way but in addition return the same status as a return value.
- The data types are slightly different.
- Other header files are provided for applications that need to be strictly compliant. PPC Services only support using the CPI-C and CPI-RR functions using the C programming language.

In addition to using any of the CPI-C and CPI-RR functions, an application can use any of the Encina or DCE components.

Encina provides three components for managing transactions (that is, for delimiting, starting, committing, and aborting transactions):

- Tran-C
- The lower level Encina Toolkit components (specifically, TRAN and threadTid)
- TX

These components are used by applications using synclevel syncpoint (SL2) conversations.

Tran-C is the recommended mechanism for creating the transactions within which transactional operations must be executed. Tran-C is discussed in Section 5.1.2.1, "Transactional-C Programming Language" on page 80. The transaction construct defines a scope within a program; all functions called within that scope become part of that transaction. When the transaction construct is encountered in a program, a transaction is automatically created by Tran-C. When the end of the transaction construct is successfully reached, Tran-C automatically attempts to commit the transaction. If the transaction is aborted during the execution of the computations bound by the transaction construct, Tran-C automatically transfers control to the end of that scope.

### 5.4.3 Distributed Program Link

Encina PPC distributed program link (DPL) provides a mechanism for Encina applications to communicate with Customer Information Control System (CICS) applications that is conceptually similar to a TRPC. DPL allows Encina applications to interact with CICS DPL applications (that is, CICS applications that use the EXEC CICS LINK command) and other Encina DPL applications. DPL always sets a syncpoint on its return.

As shown in Figure 49 on page 135, the PPC gateway server is the bridge between the Encina and CICS applications, seamless linking the DPL client to the DPL server. Encina PPC and DCE specify the logical and physical connections between the Encina application and the PPC gateway server. SNA LU 6.2 specifies the logical and physical connections between the PPC gateway server and the mainframe. The PPC library handles the details, including syncpoint processing, conversation deallocation, and security.

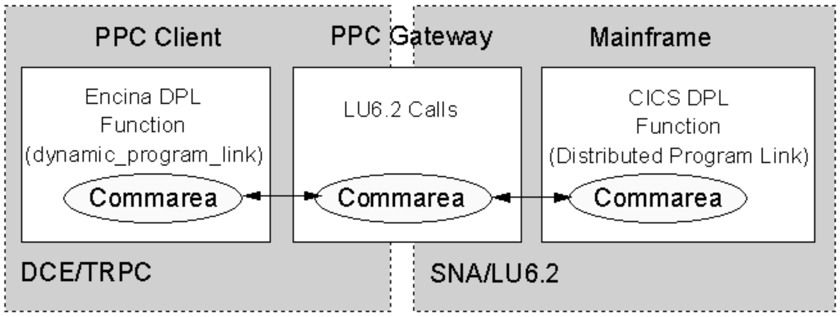

PPC Client
PPC Gateway
Mainframe
Encina DPL Function (dynamic_program_link)
LU6.2 Calls
CICS DPL Function (Distributed Program Link)
Commarea
Commarea
Commarea
DCE/TRPC
SNA/LU6.2

*Figure 49. Encina PPC Client Calling a CICS Transaction through DPL*

In PPC, DPL allows Encina applications to act as either the linking or the linked-to program. Conceptually, linking to a remote program is like making a TRPC to the remote program. Therefore, from an Encina point of view, the linking program is a client, and the linked-to program is a server. If the server function returns without error, DPL automatically sets a "syncpoint on return."

Data is passed between the client and the server through a single opaque buffer called the communications area (COMMAREA). The client and server must agree on the format and size of this buffer. The client specifies the format of binary or string data that is passed back and forth in the

COMMAREA. Figure 50 on page 136 shows the `Dynamic_Program_Link` function linking to a CICS program.

```
CM_RETCODE Dynamic_Program_Link (
         IN CONVERSATION_ID conversationId,
         IN int isLastTrpcP,
         IN char *progName,
         IN char *invokingProgram,
         INOUT char *commArea,
         IN int commAreaLength,
         INOUT int *dataLengthP,
         OUT CM_RETCODE *returnCodeP);
```

*Figure 50. Dynamic_Program_Link Function*

You do not have to include any special logic in the application to initialize multiple conversations within the same transaction. DPL can reuse conversations within a transaction. A PPC conversation is allocated for the first `Dynamic_Program_Link` call during a transaction. The conversation is reused for subsequent calls that have the same PPC allocation parameters. It is automatically deallocated when the transaction commits.

#### 5.4.3.1 Writing DPL Client Code

Writing a DPL client application is similar in many ways to writing a PPC Executive application. Both applications must initialize PPC Services, configure information about the remote application, and initialize a conversation. DPL clients also use the same side information files as PPC Executive applications. However, a DPL client application does not need to follow the logic of the remote application to the same degree as a standard PPC application. DPL's transactional RPC-like interface allows more flexibility in client application design. You also do not need to explicitly deallocate and reallocate conversations between Dynamic_Program_Link (CMDPLINK) calls, because DPL automatically reuses conversations within a transaction whenever possible.

An Encina program acting as a DPL client must perform the following steps:

1. Initialize PPC by calling the `cpic_Init` function.
2. Configure information about the remote application to which the client is to link. Typically, this information is stored in a side information file and read into the program by calling the `cpic_ReadSideInfo` function.
3. Initialize the conversation by calling the `Initialize_Conversation` function.

4. Set other conversation characteristics such as the synclevel. DPL requires synclevel syncpoint (SL2) conversations.
5. Place any data to be sent to the remote application in the COMMAREA.
6. Optionally, specify the byte order or code page for the COMMAREA by calling the `Set_DPL_Locale` (`CMSDPLOC`) function.
7. Start a transaction by using the Tran-C transaction construct.
8. Make the RPC by calling the `Dynamic_Program_Link` function.
9. Check for server errors by calling the `Extract_DPL_Error` (`CMEDPLER`) function.
10. Process any data returned in the COMMAREA.

Steps 1 through 4 are required for all PPC applications that allocate conversations. Steps 5 through 10 apply to DPL clients.

Figure 51 on page 138 shows a complete example of a DPL client. The example, extracted from the *Encina PPC Programming Guide,* invokes a function named QUERY in the remote program.

```
#define COMM_AREA_SIZE 1024
     void Link_To_Remote_Program (char *symDestName,
                                  char *tpn)
     {
          CONVERSATION_ID  conversationId;
          char             commArea[COMM_AREA_SIZE];
          CM_RETCODE       returnCode;
          /* Initialize byte order and code page to local defaults */
          CM_DPLBYTEORDER  byteOrder = CM_DPLBYTEORDER_LOCAL;
          CM_DPLCODEP  codePage = CM_DPLCODEP_LOCAL
          /* Send entire COMMAREA */
          int dataLength = sizeof(commArea);
          /* Initialize PPC Services and read configuration information
           * from sideinfo file -- see Figure 2.3  */
          ...
          /* Initialize a PPC conversation, set the synchronization
           * level to synclevel 2, and allocate a conversation --
           * see Figure 2.7  */
          ...
          /* Initialize the COMMAREA and read input data into the
           * commArea buffer. */
          memset(commArea, 0, dataLength);
          sprintf(commArea, "Hello, CICS!");
          /* Optionally, set the byte order and code page of the server */
          Set_DPL_Locale(conversationId, byteOrder, codePage,
                      &returnCode);
          /* Begin the transaction */
          transaction

          {
               /* Make the TRPC */
               Dynamic_Program_Link(conversationId,
                      TRUE,
                      "QUERY",
                      "MY_PROGM",
                      commArea,
                      COMM_AREA_SIZE,
                      &dataLength,
                      &returnCode)
               /* Check the return code to detect any server errors */
               Extract_DPL_Error  (conversationId,
                      abendcode,
                      *condCodeP,
                      *returncode);
               /* Process return data */
               ...
          } onCommit{
              /* Do commit-specific processing here */
          } onAbort{
              /* Do abort-specific processing here */
          }
     }
```

*Figure 51. Example of a DPL Client*

# Chapter 6. Using Encina++

In this chapter we describe the basic steps for building applications that use the Encina++ programming model. Encina++ is an extension to Encina that is used to build object-oriented distributed computing systems. For a more general description of Encina++, see Section 2.4, “Encina++” on page 29. The Encina++ programming model is described in Section 6.2, “The Encina++ Programming Model” on page 141.

## 6.1 Overview of Encina++ Application Development

An Encina++ application is written in an object-oriented language such as C++. The objects can be accessed locally or remotely through an IDL provided by DCE or CORBA (Orbix from IONA Technologies, Ltd.). Encina++ itself contains several APIs, some of which are common to both DCE and CORBA, and some are designed for only one or the other. The interface you use depends on the requirements of your particular application.

### 6.1.1 Encina++ Interfaces

Common interfaces:

- The Encina C++ interface defines C++ classes and member functions that enable the creation and management of client/server applications and provide support for the underlying environment.
- The Tran-C++ interface defines C++ constructs and macros as well as classes and member functions for distributed transaction processing. This interface provides an object-oriented alternative to the Encina Tran-C interface.
- The OMG OTS interface also defines C++ classes and member functions for distributed transactional processing. This interface implements the OTS specification as documented in OMG document 94.8.4.

DCE-only interfaces:

- The RQS C++ interface (RQS++) defines C++ classes and functions for enqueuing and dequeuing data transactional.
- The SFS C++ interface (SFS++) defines C++ classes and functions for manipulating data stored in record-oriented files while maintaining transactional integrity.

CORBA-only interfaces:

- The Object Concurrency Control Service (OCCS) interface defines C++ classes and functions that enable multiple clients to coordinate access to shared resources. This interface implements the OMG Concurrency Control Service Proposal as documented in OMG document 94.5.8.
- The Java OTS client interface defines Java classes and functions that enable Java client applications to begin and control distributed transactions. This interface implements the OMG OTS specification as documented in OMG document 94.8.4.

The Encina++ interfaces are designed to support functionality exported by the Encina Monitor and can be used to create Monitor application servers and clients in C++. Monitor application servers and clients can use DCE or CORBA or both. See Section 5.1, “Encina Monitor” on page 77 for more details on the Monitor.

The Encina++ interfaces also support the development of C++ client and server applications that do not run under the control of the Encina Monitor. Encina applications that do not use the Monitor are sometimes generally referred to as Toolkit clients or servers. Toolkit applications can use DCE or CORBA or both. Applications that use CORBA rely on the Orbix ORB from IONA Technologies, Ltd.

RQS++ and SFS++ applications can be Monitor application servers or clients or they can be Toolkit servers or clients. All RQS++ and SFS++ applications require DCE, so they can use either DCE only or both DCE and CORBA.

### 6.1.2 Developing a Distributed Encina++ Application

The following steps are required to develop a distributed application in Encina++:

1. Design the application and determine the local and remote objects and procedures that are required.
2. Use an IDL to define the remote objects and procedures.
3. Compile the IDL files to generate client and server stub classes.
4. Write the application code for the client and server.
5. Compile and link the client and server applications.

Step 1, application design, is covered in Part 3, “Case Study” on page 177. Steps 2, 3, and 5 depend on the environment, DCE or CORBA, for which you are developing your application. The environment determines which IDL, IDL compiler, and libraries to use to build your application.

Step 4 involves some tasks that are environment-specific and some that are not. The tasks that are common to both environments, such as initialization and termination of Encina++ clients and servers, are covered in this chapter.

## 6.2 The Encina++ Programming Model

The Encina++ classes support a client/object programming model in which clients access objects instead of servers. Servers export one or more interfaces (classes) and one or more instances of each class (objects). The client application can access objects exported by servers without you knowing how the objects available in the system map to servers.

Clients can bind to objects exported by servers. They can bind to individual objects when the objects are known, or they can bind to a class when the objects are not known or when all objects of a specific class provide the same capabilities. Typically, you specify a name for an object. Although each object created has a UUID, naming an object allows clients to bind to the object by name instead of by UUID.

In Encina++, an IDL is used to specify the interfaces to objects in the form of remote procedures. The remote procedures are used for communication between the client and server applications. The interface compiler generates files that include client stub and server stub classes for each interface. These stub classes give the client and server a slightly different view of the same interface.

Before RPCs can be made between a client and server, the server must be available to receive requests from clients. Creating an instance of the server stub class within a running server causes the object to be exported to the namespace so that a client can locate and bind to it. The instance is referred to as a *server object*.

A client creates an instance of the corresponding client stub class; the instance is referred to as a *client proxy object*. The client application uses the client proxy object to bind the client to the server object. After the client proxy object is bound to the server object, each member function call made on the client proxy object invokes an RPC to the server object, which executes the procedure and returns results to the client proxy object. The client communicates with the server object through the client proxy object.

Normally, server objects must be created before the server begins listening for RPCs so that clients can locate and bind to the server objects. Server objects can also be created dynamically; you can use a factory to create server objects while the server is listening for RPCs. A factory is an object designed to create other objects that are managed by the server.

Before a server begins listening, it can create a factory object, exporting the factory object to the namespace and enabling a client to bind to it. When the factory object creates a server object, the factory object must return a unique identifier, called an *object reference*, for the server object. Clients use the object reference to create a client proxy object, so that the client proxy object will bind directly to the server object. The client proxy object is created automatically in CORBA applications, whereas in DCE applications, the client must create the client proxy object explicitly.

Clients and servers are implemented as objects in Encina++ applications. The Encina C++ interface supplies a client class and a server class that can be used to initialize clients and servers; the Java OTS client interface supplies a client class that can be used to initialize clients. Operations available on an initialized server instance can be used to register XA resources, make server objects available to clients, and listen for incoming RPCs.

In a DCE environment, you can administer Encina++ application servers by using the Enconsole administrative tool. In a CORBA environment, you can administer Encina++ application servers by using the administrative tools provided by your ORB.

Applications use transaction processing to ensure that data remains correct, consistent, and secure. Transaction processing in an object-oriented distributed environment enables distributed objects to meet the same requirements. Encina++ supplies two different C++ interfaces for object-oriented transaction processing: Tran-C++ and the OMG OTS. These interfaces can be used either separately or together in an Encina++ application. Encina++ also supplies a Java language version of the OTS interface for Java clients.

Tran-C++ provides constructs, macros, and classes that integrate transactional semantics into the C++ programming language. In Tran-C++, a transaction class is used to implement transactions as objects. The constructs and macros use functionality defined for transaction objects to simplify the creation and management of transactions in Encina++ applications.

Encina’s OMG OTS interface provides CORBA-compliant classes for transaction processing. The OTS interface defines classes for two transaction demarcation models. In the implicit model, the client implicitly passes the transaction context that defines a transaction to an object by associating the context with the calling thread. In the explicit model, the transaction context must be passed explicitly to an object as a parameter in a function call.

The Data Definition Language (DDL) provides a means for defining the data objects that RQS++ and SFS++ use to represent elements, records, and keys. Each type of record or element is specified as an interface in a DDL file. Keys can be specified for the data types.

To use DDL, you define the data objects you need for your RQS++ and SFS++ programs in a DDL file. The DDL file is then processed by the ddl command, which generates header and source files containing C++ classes based on the data objects specified in the DDL file.

SFS++ applications use objects of the generated classes as records for file input and output. The same classes can also be used for creating SFS files. Objects of the key classes generated by DDL can be used to access records in SFS files and to add secondary indexes to SFS files.

RQS++ applications use objects of the classes generated by DDL as the elements that are enqueued, dequeued, and requeued. The generated classes can also be used to define element types at the RQS server.

## 6.3 Writing Encina++ Server Applications

To initialize an Encina++ server application, follow these steps:

1. Create one server class instance to manage the server.
2. Register any resources required by the server (optional).
3. Initialize underlying Encina services (optional).
4. Create one or more server objects.
5. Listen for incoming RPCs.
6. Terminate the server.

```
int main(int argc, char *argv[])
     {
       // process command-line arguments...
// Step 1:
       // Create and initialize the server
       Encina::Server server;

// Step 2:
// Perform database initialization and get the XA switch and
        // the open and close strings for the resource
        dbInit(&xaSwitch, &open, &close);
        // Register a resource with the server
        server.RegisterResource(xaSwitch, open, close, 0);

// Step 3:
// Initialize the server
       server.Initialize();
// Initialize other Encina components...

// Step 4:
// Create server objects...

       try {
// Step 5:
          // Listen for incoming RPCs

server.Listen(Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS);
       }
       catch (...) {
// Step 6:
          cerr << "An exception was raised." << endl;
          server.Exit(1);
       }

// Step 6:
       server.Exit(0);
       return 0;
     }
```

*Figure 52. Example of Initializing an Encina++ Server Application*

The first step is to create an instance of the `Encina::Server` class to represent the application server. The class constructor takes no arguments.

After the server instance is created, you must register any resources that your server requires. The `Encina::Server::RegisterResource` function registers XA-compliant resources and makes the server recoverable.

After you have created the server instance and registered any required resources, you have to initialize the underlying Encina components and services. Initialization can be done explicitly by calling the `Encina::Server::Initialize` function. Calling this function is optional in certain cases because the `Encina::Server::Listen` function (called as the final step in your server application) initializes the underlying components and services if they are not already initialized. You must call the `Encina::Server::Initialize` function if you want to do application-specific initialization that relies on these underlying components before the server begins listening for RPCs. If you need more information about initializing Encina servers, see the *Encina Object-Oriented Programming Guide* at http://www.transarc.com.

After the server is initialized, you must create one or more server objects to handle incoming requests. Named server objects (as well as factory objects) must be created before the server starts listening for RPCs.

After you have created the server objects, you must start the server listening for incoming RPCs. Calling the `Encina::Server::Listen` function causes the server to start accepting RPCs and sets the concurrency mode for the server. The value passed as the function parameter sets the concurrency mode, which determines whether transactions and incoming RPCs are serialized at the server. This setting controls the type of access that the client has to the server. For example, if you specify no serialization, the server starts a new thread automatically for each transaction and RPC. See the programming reference manual for the `Encina::Server::ConcurrencyMode` type for descriptions of the available modes. The example in Figure 52 on page 144 shows the function being used to listen for RPCs. The `Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS` concurrency mode specifies that all TRPCs and transactions are serialized at the server. If the server accesses a resource, such as a database, that does not have thread-safe libraries, you must specify that the server serialize TRPCs and transactions.

Terminating a server stops the server from listening for incoming RPCs and stops the underlying Encina services. Normally, a server is shut down administratively or through a system failure.

If you need to forcibly terminate the server application programmatically at any time, you can use the `Encina::Server::Exit` function. The function never returns; it takes one argument, which is an integer status value that is returned to the calling environment. The example in Figure 52 on page 144 shows the function being used to terminate a server application when an exception is thrown.

If the server is stopped in an orderly manner (not forcibly), the `Encina::Server::Listen` function returns. Thus your server application can do any necessary cleanup before the application exits.

## 6.4 Writing Encina++ Client Applications

Follow these steps to initialize an Encina++ client application (see Figure 53 on page 146):

1. Initialize underlying Encina services
2. Bind to a remote object
3. Terminate the client

```
int main(int argc, char *argv[])
     {
        // process command-line arguments

// Step 1:
        // Initialize the client
        Encina::Client::Initialize();

// Step 2:
        try {
           // Bind to a remote object
        }

// Step 3:
        catch (...) {
           cerr << "An exception was raised." << endl;
           Encina::Client::Exit(1);
        }

        // Perform work...

        return 0;
     }
```

*Figure 53. Example of Initializing an Encina++ Client Application*

The `Encina::Client::Initialize` function initializes a client application and all of the necessary underlying Encina components and services.

**Note**

The functions of the `Encina::Client` class are static functions; it is not necessary to create an instance of the class.

The client application uses the client stub class generated from the interface definition to locate and bind to remote objects or servers that export the requested interface. After the client application is initialized, you use an instance of the client stub class, a client proxy object, to bind to a remote object. The call you make to bind the client to a remote object depends on whether your client is an Encina++/DCE or Encina++/CORBA application. The client stub class includes member functions for the operations defined in the interface. Once a proxy object is bound to a remote object, calling a member function on the proxy object initiates an RPC, invoking the corresponding method on the remote object.

Normally, a client application terminates at the end of the main routine (or whenever the `Encina::Client` object is destroyed or goes out of scope). If you need to terminate the client at any other time, you can use the `Encina::Client::Exit` function. The function takes one argument, which is an integer status value that is returned to the calling environment. Terminating the client application also terminates all underlying Encina services for the client application. All of the transactions in progress are completed (either committed or aborted) before the application exits.

*If the client application is interrupted by the user, it exits automatically, aborting any transactions in progress.*

## 6.5 Terminology

As discussed in Section 6.1.1, “Encina++ Interfaces” on page 139, Encina++ has the Encina++/DCE, the Encina++/CORBA, and the Encina++ common interfaces. Although the underlying functionality is the same, the Encina++/DCE and Encina++/CORBA implementations address the differences in DCE and CORBA. Important differences include client and server stub generation and binding methods.

To write Encina++/DCE applications, you must use Encina's TIDL compiler to generate stub files for communications between Encina++ clients and servers, adding transactional semantics to remote procedures; using TIDL, you define which functions in the interface are transactional.

To write Encina++/CORBA applications, however, you use a CORBA IDL compiler specific to the ORB being used; using the CORBA IDL, you define entire interfaces, rather than individual functions, as transactional.

Encina++/DCE clients bind to exported server objects by using constructors defined in the client stubs generated by the TIDL compiler. In Encina++/CORBA applications, however, clients use a binding method that is specific to the ORB being used (or clients can use a CORBA Object Naming Service, if available). Applications that use CORBA rely on the Orbix ORB from IONA Technologies, Ltd.

### 6.5.1 Encina++/DCE Programming

Encina++ /DCE supports the development of transactional, object-oriented applications for DCE. Encina++ /DCE applications use DCE's RPC mechanism for communication between clients and servers. This dependency on DCE RPC affects interface definition, binding, and exception handling.

In the DCE environment, the TIDL must be used to specify object interfaces in the form of remote procedures. The TIDL compiler generates C++ stubs that include client stub and server stub classes for each interface. These stub classes give the client and server a slightly different view of the same interface.

On the server side, two server stub classes are generated. The abstract server stub class contains virtual functions that map to the remote procedures defined in the interface. The concrete server stub class is derived from the abstract server stub class. The server application developer typically implements the remote procedures for the interface as member functions of the concrete server stub class. The server application can instantiate objects of the concrete class; these objects are then available to clients.

On the client side, a client stub class is generated. The client stub class includes several constructors that hide the details of binding. These constructors enable an instance of the class to represent a remote server object; a client stub class instance acts as a proxy for the object to which it is bound at the server. The client stub class defines member functions that map to the remote procedures defined in the interface. Calls to the proxy object's member functions result in RPCs to the object that the proxy is bound to; the

RPCs invoke the member functions that are defined in the concrete server stub class.

TIDL is also used to define factory objects. The TIDL interface defining a factory object must include remote procedures for creating and deleting server objects. Typically, create and delete functions are defined for the factory object.

Depending on the requirements of the client application, you can use one of several methods to bind the client to the server. How you create an instance (client proxy object) of the client stub class determines the binding method used. The client proxy object can be bound to one of the following:

- Any compatible server object at any server
- A server object with a specific name
- A specific server object created dynamically at a specific server
- A specific server that exports the required interface

The first time a member function call is made on the client proxy object, the proxy object is bound to a server object (or server) and the call is passed to that server object. The client then communicates with the server object through the client proxy object.

The Encina++ programming interfaces that are supported only for DCE provide object-oriented access to RQS++ and SFS++. In addition, you can use Encina's DDL compiler to generate stub files for RQS++ and SFS++ applications.

### 6.5.2 Encina++/CORBA Programming

Encina++ /CORBA supports the development of transactional, object-oriented applications for the CORBA environment. Encina++ /CORBA applications rely on an ORB for communication between clients and servers. This dependency on an ORB affects interface definition, binding, and exception handling.

In the CORBA environment, the CORBA IDL must be used to specify the interfaces to objects. The operations defined by an object's interface are used for communication between the client and server applications. The CORBA IDL compiler generates stub files that include client stub and server stub classes for each interface.

Depending on the requirements of the client application, to bind the client to the server you can use either an Object Naming Service or a binding method specific to the ORB.

Encina++ /CORBA relies on the Orbix implementation of the IDL-to-C++ mapping for the definition of interfaces and on the Orbix IDL compiler for generating stub files. You define interfaces for objects in Encina++ /CORBA applications in the same you define them in Orbix applications.

Objects that participate in transactions or make transactional requests on other objects are called *transactional objects*. You make an object transactional by specifying that its interface is derived from the `CosTransactions::TransactionalObject` class in the IDL file. Interfaces to objects must specify that the object is transactional if your application uses Tran-C++ or the Current class to manage transactions implicitly.

An ORB-specific binding method can be used to bind the client to the server. For Orbix, the IDL compiler generates a client stub class that corresponds to the interface definition. The generated class contains a static member function named *_bind*; calling the _bind function creates an instance (client proxy object) that is bound to an object at the server.

When the binding function call is made on the client proxy object, the proxy object is bound to a corresponding remote object, the server object. The client then communicates with the server object through the client proxy object.

To implement the server interface, you must define a C++ class and class methods corresponding to the interface definition in the IDL file. The class name must be different from the class name used by the client application. Encina++ /CORBA server applications can use either of the Orbix approaches to implementing the IDL interface: the Basic Object Adapter (BOA) approach or the TIE approach. If you are using the BOA approach, your implementation class must inherit from the BOA class defined in the header file generated by the IDL compiler.

To create a server object in an Encina++ /CORBA server application, you simply create an instance of the implementation class you defined for the server. When a client application calls the binding method defined for the client proxy object, the proxy object binds to the server object.

You must create one or more server objects before the server application starts listening for incoming requests. You can use either the `Encina::Server::Listen` function or the `CORBA::Orbix::impl_is_ready` function that Orbix provides to start the server listening. The difference between the two functions is that `Encina::Server::Listen` automatically creates a pool of threads that handle concurrent requests in servers that are thread aware, unless you specify otherwise.

Before you run an Encina++ /CORBA server, you must specify a name for the server. If the server is a shared Orbix server (started dynamically through the Orbix daemon), you specify the server name by registering the server with the Orbix daemon. If the server is a persistent Orbix server (started manually), you specify the server name through the ENCINA_OTS_TK_SERVER_ARGS environment variable or the -encina command-line switch.

For all Encina++ /CORBA servers that are recoverable servers, you must also specify a name for the server's restart files and the name of a log volume. Restart files and log volumes can also be specified through the environment variable or command-line switch.

### 6.5.3 Encina SFS++

The SFS++ interface consists of a set of C++ classes for creating Encina SFS applications. Together with other parts of Encina++, such as Tran-C++, SFS++ enables you to develop object-oriented Encina applications. The SFS++ classes contain functions that invoke the most commonly used features of SFS. Using these functions, SFS applications can perform the following operations:

- Insert, read, update, and delete records in SFS files
- Create and delete SFS files
- Add secondary indexes to existing SFS files
- Set idle time-outs for SFS file objects and operation time-outs for operations on SFS file objects
- Get open file descriptors (OFDs) for file objects to allow use of SFS C functions within SFS++ applications

The SFS++ interface does not support the following operations; however, SFS++ allows you to call any of the SFS C functions to perform them:

- Creation of entry-sequenced and relative files
- Partial record reads and updates
- Batch operations

SFS++ encapsulates SFS features into a set of classes.

The `Sfs::Server` class is an abstraction of an SFS server. It provides methods for creating, opening, and deleting SFS files and for adding secondary indexes to SFS files.

Before calling any SFS++ functions, an application must create an `Sfs::Server` object by using the class constructor. The application must specify the fully qualified name of a running SFS server when it creates the `Sfs::Server` object.

The `Sfs::File` class is an abstraction of an SFS file. It provides functions for selecting ranges of records within files and for inserting, reading, updating, or deleting records from files.

An object of this type is returned when an application calls the `Sfs::Server::OpenFile` function. There are no public constructors for the `Sfs::File` class.

The `Sfs::Volume` class identifies an SFS data volume and a specific amount of space on that volume to be used to store the contents of an SFS file or an SFS file's secondary index. Once defined, an `Sfs::Volume` object can be used by the `Sfs::Server::CreateFile` and `Sfs::Server::AddSecondaryIndex` functions to allocate storage space for files and secondary indexes.

The `Sfs::Exceptions` class defines the SFS++ exception classes. All SFS++ exception classes are derived from the `OtsExceptions::Any` class, allowing a single C++ catch clause to be used to catch all SFS++ exceptions.

Each SFS file can store records of only one record type. The record type defines the data type for each field of the record. SFS++ applications use DDL to define SFS record types. From these definitions, the ddl command then generates a record class, derived from the `Pos::Object` class. The record class represents the record type and contains constructors to create and initialize record objects of that class.

Each SFS file must have a primary index and can have one or more secondary indexes. These indexes are used to access the records in the file. SFS++ applications use DDL to define SFS indexes. From these definitions, the ddl command then generates key classes, derived from the `Pos::Key` class. The key class represents the key type and contains constructors to create and initialize key objects of that class.

### 6.5.4 Encina RQS++

The RQS++ interface consists of a set of C++ classes for creating Encina RQS applications. Together with other parts of Encina++, such as Tran-C++, it enables you to develop object-oriented RQS applications.

Using RQS++, an application can perform the following operations:

- Enqueue, dequeue, and requeue elements to queues
- Dequeue elements from queue sets
- Use cursors to read elements
- Create queues and queue sets
- Add queues to queue sets and set service levels for queue sets

RQS++ does not support the following operations; however, RQS++ allows you to call any of the RQS C functions to perform them:

- Using element identifiers and keys to access elements
- Callbacks
- Batch operations
- Administrative operations, such as gathering information about queues and queue sets

RQS++ encapsulates RQS features into a set of classes. All of them are contained within the `Encina::Rqs` class.

The `Rqs::Server` class is an abstraction of an RQS server. It provides methods for creating and deleting element types, queues, and queue sets. Before performing any other RQS++ operations, an application must create an `Rqs::Server` object by using the class constructor. The application must specify the name of an actual running RQS server when it creates the `Rqs::Server` object.

After creating the `Rqs::Server` object, the application can then use `Rqs::Server` class member functions to create `Rqs::Queue` and `Rqs::QueueSet` objects.

The `Rqs::Queue` class is an abstraction of an RQS queue. It provides member functions for enqueuing and dequeuing elements, controlling access to a queue, and getting cursors for sequentially scanning elements in a queue. Before an application can enqueue, dequeue, or requeue elements to a queue, it must create an object of this class.

The `Rqs::Queue` class has no public constructor. Instead, an object of this type is returned when an application calls the `Rqs::Server::GetQueue` or `Rqs::Server::CreateQueue` function.

The `Rqs::QueueSet` class represents a queue set. A queue set serves as a dequeueing structure that regulates how elements are dequeued from its member queues. An application simply dequeues from the queue set, and the individual queue is chosen by the selection process defined for the set. The `Rqs::QueueSet` class contains member functions for adding and removing queues from a queue set, service levels, and dequeuing elements from a queue set. Before an application can dequeue elements from a queue set, it must create an object of this class.

The `Rqs::QueueSet` class has no public constructor. Instead, an object of this type is returned to an application as a result of calling the `Rqs::Server::GetQueueSet` or `Rqs::Server::CreateQueueSet` function.

The `Rqs::Cursor` class represents a cursor. A cursor is a logical, client-side object that is used to sequentially examine the objects in a queue. The `Rqs::Cursor` class has member functions that allow applications to use cursors to read through the elements in a queue. It also provides a function to allow the application to obtain a cursor handle, which can be used to access the C functions that provide additional features for using cursors.

The `Rqs::Cursor` class has no public constructor. Instead, cursors are created by using the `Rqs::Queue::GetCursor` function. Cursors are owned by a specific transaction, and only the owning transaction can use the cursor. Cursors have a lock mode and locking policy associated with them. As the cursor advances, the RQS server acquires element locks in the cursor's mode on behalf of the owning transaction. The locking policy for each cursor determines the duration for which the server holds the lock.

The `Rqs::Object` class represents RQS elements. An RQS element is record-oriented data that can be stored in a queue. Each element must have a type, which defines the data type and size for each field of the element. A queue can store RQS elements of different types. When an application dequeues an element from a queue that can store multiple types, the type of the element is not known prior to the dequeue operation. In this case, the dequeue operation returns an object of type `Rqs::Object`. The application can then query the object to determine the element type (using the `Rqs::Object::GetType` or `Rqs::Object::IsOfType` function) and convert the object to an instance of the appropriate class.

DDL must be used to define RQS elements used in RQS++ applications. From these definitions, the ddl command then generates classes, derived from the `Pos::Object` class, to represent element types.

Objects of the `Rqs::Object` type cannot be created by an application. They are returned by RQS++ functions and must be converted to an instance of the correct type.

The `Rqs::Exceptions` class defines the RQS++ exception classes. All system exception classes are derived from the `OtsExceptions::Any` class, allowing a single C++ catch clause to be used to catch all system exceptions.

To represent the elements it enqueues and dequeues, an application uses classes defined by DDL and generated by the ddl command. The class generated by DDL contains variables corresponding to the fields in the element. It provides several constructors for the class and member functions that automatically marshall and unmarshall the data to and from the fields when elements are enqueued and dequeued.

# Chapter 7. Internet Access for Java Clients

In this chapter we describe how you can build Java applications that access Encina servers through the DE-Light Java client API. A high level overview of the DE-Light product suite is presented in Section 3.3, “Encina DE-Light Web Components” on page 51.

The structure of a DE-Light Java client application is similar to that of other Java client applications. The key difference between a DE-Light application and other applications is that the DE-Light client can invoke Encina TRPCs to perform transactional work on behalf of the client. The DE-Light client application communicates with a DE-Light gateway, using a simplified RPC protocol called the DE-Light Dynamic Remote Procedure Call (DRPC). The DRPC protocol provides the tools for packaging up a TRPC to be invoked by a DE-Light gateway. A DE-Light client application makes a TRPC by creating a connection to a DE-Light gateway and then sending that gateway the following information:

- The name of the server to call
- The name of the interface that exports the desired remote procedure
- The name of the remote procedure to be executed
- The arguments to the remote procedure

The DRPC protocol provides means for selecting a DE-Light gateway, using a data dictionary for passing TRPC parameters between the client and the server, using SSL security between the client and the gateway, and handling exceptions.

## 7.1 Access to DE-Light Gateways

Before issuing TRPCs, a client must first establish a connection with a DE-Light gateway. On completion of the RPC, the connection must be closed.

### 7.1.1 Establishing a Connection

You must create a `DrpcConnection` object to establish a connection with a DE-Light gateway:

```
drpc = new DrpcConnection(gwy_name);
```

Here, the new operator creates a `DrpcConnection` object and returns a reference to it, called *drpc*. The `DrpcConnection(gwy_name)` method is a constructor, called by the new operator, that initializes the newly created

DrpcConnection object. The gwy_name argument is a string that specifies the DE-Light gateway. This string must be constructed as follows:

```
protocol:machine-name[port-number]
```

where *protocol* refers to the transport protocol being used, *machine-name* is the name of the machine on which the DE-Light gateway is running, and *port-number* is the port number (or port numbers) where the DE-Light gateway is listening. The DE-Light gateway administrator decides which port number should be used.

Currently, TCP, HTTP, and HTTPS are the supported transport protocols. For example, the gwy_name argument for a DE-Light gateway running on machine prod_one could be one of the following strings:

```
tcp:prod_one[1234]
http:prod_one[5678]
http:prod_two[1234,5678]
```

Notice that when you use HTTPS you use the same specification as you would use for HTTP, but you also specify a second port number. The first port is the nonsecure port and the second is the secure port, which you need for your HTTPS connection.

Figure 54 on page 158 shows the code used to create a `DrpcConnection` object that makes a TCP connection with a DE-Light gateway using port 13013 on a machine called prod_one:

```
DrpcConnection drpc;
try {
    drpc = new DrpcConnection("tcp:prod_one[13013]");
} catch (DrpcException e) {
    drpc = null;
    System.out.println("Failed to obtain DrpcConnection: " +
        e.toString() );
}
```

*Figure 54. Example of a TCP Connection with a DE-Light Gateway*

**IMPORTANT:** Before establishing a connection with a DE-Light gateway, be sure that the gateway has been loaded with the Encina server’s IDL or TIDL file (see Section 7.6, “Loading Gateways with IDL and TIDL Files” on page 172).

### 7.1.2 Closing a Connection

When your client application has finished making RPCs, close the connection to the DE-Light gateway. Closing the connection frees resources at the gateway. These resources can later be used to service other clients. Use the `close()` method:

```
drpc.close();
```

where *drpc* is a reference to the `DrpcConnection` object created in the client application when a connection was established.

When you close a connection that is using transactions, the active transaction is aborted.

## 7.2 Data Dictionaries

DE-Light clients use data dictionaries to pass information to the Encina servers through the DE-Light gateway. DE-Light considers a data dictionary to be a mapping between variable names and their values. A data dictionary entry includes the variable name and its value. The names of data dictionary variables are strings. The values of the variables are to be passed to or returned by remote procedures. DE-Light supplies a default data dictionary with each gateway connection.

Once a connection has been established with the Encina server, the DE-Light Java client can use the data dictionary for the connection to access the Encina server, by following these steps:

1. Get a handle to the data dictionary to be used.
2. Use the data dictionary to set the values of any input parameters.
3. Make the RPC or TRPC.
4. Read the values of any output parameters from the data dictionary.

You can obtain the handle of your default data dictionary by using the `dictionary()` method:

```
DrpcDictionary dict = drpc.dictionary();
```

where *dict* is a reference to the DrpcDictionary object being found, and *drpc* is a reference to the `DrpcConnection` object defined earlier in the client application.

### 7.2.1 Loading Data Dictionary Variables

Before making an RPC, you need to set the values of any input parameters for the RPC in the data dictionary. The `DrpcDictionary` class provides the `put()` method, which stores an object reference in the data dictionary. The `put()` method takes two arguments: the name of the variable in the data dictionary, and a reference to the object that holds the variable's value.

To store scalar values, the `DrpcDictionary` class provides several convenience methods, which convert scalar types to object references and then call the `put()` method internally. The `DrpcDictionary` class provides the following convenience methods:

```
putBoolean(String, boolean)
putByte(String, byte)
putChar(String, char)
putDouble(String, double)
putFloat(String, float)
putInt(String, int)
putLong(String, long)
putShort(String, short)
putUnsignedByte(String, byte)
putUnsignedShort(String, short)
```

For example, suppose a stock_OrderItem remote procedure has two input arguments, one an integer, the other a string. The server's IDL file exports the remote procedure as follows:

```
void stock_OrderItem([in] long stockNum, [in] long name);
```

The integer parameter is set by calling the `putInt()` method:

```
dict.putInt("itemNum", 42);
```

where *dict* is a reference to a DrpcDictionary class defined earlier in the client application and *itemNum* is the input parameter being defined in the data dictionary. The value of itemNum is set to 42.

The string parameter is set by calling the put() method:

```
dict.put("itemName", "printer");
```

where *"itemName"* is the input parameter being defined in the data dictionary. The value of itemName is set to the string "*printer*".

The names of the variables in the data dictionary do not have to be the same as the names of the parameters in the IDL file. In the previous example, the data dictionary itemNum variable corresponds to the *stockNum* parameter in the stock_OrderItem remote procedure. Similarly, the data dictionary itemName variable corresponds to the *name* parameter.

### 7.2.2 Retrieving Data Dictionary Variables

After an RPC returns, you can retrieve the values of output parameters from the data dictionary. The `DrpcDictionary` class provides the `get()` method, which retrieves an object reference from the data dictionary. The `get()` method takes one argument: the name of the variable in the data dictionary.

To retrieve scalar values, the `DrpcDictionary` class provides several convenient methods, which call `get()` internally and then convert the returned object reference to a scalar value:

```
getBoolean(String)
getByte(String)
getChar(String)
getDouble(String)
getFloat(String)
getInt(String)
getLong(String)
getShort(String)
```

The following example retrieves one output parameter, an integer, from the data dictionary through the getInt() method:

```
return dict.getInt("price");
```

where *dict* is a reference to the `DrpcDictionary` object created earlier in a client application, and *price* is the output parameter being extracted from the data dictionary.

## 7.3 Access to Encina Servers

You can use DE-Light Java clients to access both DCE and Encina servers by using RPCs and TRPCs. The DCE IDL provides a number of attributes and data types that you can use when defining an interface. At load time, the DE-Light gateway rejects any RPC function declarations that use unsupported elements. Table 5 on page 162 lists all TIDL elements that are not supported by DE-Light. You should consult your release notes and the DE-Light documentation to find out whether any of the supported elements are interpreted differently in DE-Light from the way they are interpreted in Encina.

*Table 5. TIDL Elements Unsupported by DE-Light*

| DE-Light Constant | Description |
|---|---|
| Attributes | endpoint<br>exceptions<br>ignore<br>local<br>reflect_deletions |
| Type declarations | pipe<br>union<br>ISO_MULTI_LINGUAL<br>ISO_UCS |

### 7.3.1 Making Remote Procedure Calls

After setting the values of the input parameters in the data dictionary, you can make an RPC by using the `callRpc()` method. This method takes one argument, a string that describes the RPC to be invoked at the server. The string has two parts: an RPC description and a parameter description.

The RPC description contains the following information:

- The name of the server. This name must be specified for DCE and Encina Toolkit servers. It should not be specified for Encina Monitor application servers.
- The name of the interface. This name is required if more than one remote procedure of a given name is available at the gateway.
- The name of the remote procedure. This name is required and must always be specified.

Each part of the RPC description is specified by a list of keyword=value pairs, separated by white space. The keywords are *server*, *interface*, and *rpc*. The values are tokens, without white space, or constants. For example, an RPC description could be specified as:

```
"server=/.:/servers/merchandise interface=merchandise rpc=OrderItem"
```

The parameter description is made up of the names of data dictionary variables or constants. All function parameters must appear in the order given in the prototype for the remote procedure as defined in the IDL or TIDL file. If the remote procedure returns a value, the name of a dictionary variable in which the function is to place the return value must appear first in the parameter list.

Each parameter in the parameter list can optionally be preceded by a tag that shows its direction: [in], [out], [in,out] (or [inout]), or [return]. Constants in parameter descriptions can be preceded only by the [in] tag. Constants in the parameter description must be one of the following types:

***String Constant***
A (possibly empty) sequence of characters enclosed in double quote ( “ “ ). If the string constant is contained within another string, for example, if a string constant is used as a parameter in the string argument to a `callRpc()` method call, the string constant's double quotes must be preceded by a backslash ( \ ).

***Integer Constant***
A signed, 64-bit, long integer. A leading zero (0) indicates that the integer is specified in octal; a leading 0x or 0X indicates that the integer is specified in hexadecimal.

***Floating Point Constant***
A signed double that is written with a decimal point and an optional exponent, which is converted as if by the `valueOf` method in the `Double` class.

***Array Constant***
Arrays are delimited by braces ( { } ); the items in the array are separated by commas. Arrays can contain strings, long integers, or doubles. For example, an RPC call to an Encina monitor application server supporting the hello interface looks like this:

```
// Note that the server name is not specified for
// Encina Monitor applications.
drpc.callRpc("interface=hello rpc=helloEncina" +
             " [in] client_greeting [out] server_reply");
```

## 7.3.2 Making TRPC Calls

Basic TRPCs are similar to nontransactional RPCs except that they are prefaced by a `txBegin()` method call (marking the beginning of the transaction), and they end with one of the following method calls:

- `txCommit()`, which commits all changes made by RPCs within the transaction
- `txRollback()`, which notifies the participants to roll back any changes made by RPCs within the transaction

The basic steps in managing a transaction are:

1. Start the transaction by calling the `txBegin()` method.
2. Perform any actions that are part of the transaction (such as making RPCs).
3. End the transaction by calling either the `txCommit()` method or the `txRollback()` method.

Under TX, only the participant who started the transaction can initiate commit processing. This occurs in an application when you call the `txCommit()` method. However, calling this method does not mean that the transaction has committed. It means only that you want to commit the transaction. Other participants in the transaction must agree to commit.

Any participant in the transaction (the server, for example) can explicitly abort the transaction if it encounters an error. If the transaction cannot be committed, the `txCommit()` method throws an exception specifying that the transaction has been aborted. The client can explicitly abort the transaction by calling the `txRollback()` method.

The Java code sample in Figure 55 on page 165 shows the structure of a basic transaction using the following methods:

```
txBegin()
txCommit()
txRollback()
txSetRollbackString()
txGetRollbackString()
```

Code not essential to the example in Figure 55 on page 165 has been omitted (such as setting up parameters). Note that the sample block is enclosed by try and catch statements to catch and handle exceptions. For more information about exception handling, refer to Section 7.4, “Exceptions” on page 166.

```
Transaction t = drpc.transaction();
try {
    t.txBegin();
    // Set up parameters (code omitted)...
    // Make the TRPC call
    drpc.callRpc("server=/.:/foo rpc=withdraw " +
    "[in] acct [in] amount [out] balance");
    // ... get values of out parameters (code omitted)
    // now commit or rollback the transaction.
    if (balance > 0) {
        // Attempt to commit. If another participant has rolled
        // back, an exception will be thrown.
        t.txCommit();
    }
    else {
        // Not enough funds, so roll the transaction back
        t.txSetRollbackString("Insufficient funds");
        t.txRollback();
    }
} catch (Exception e) {
    // Control arrives here if either the RPC failed, the
    // transaction commit failed or there was some other
    // runtime exception.
    if (t.txState() == t.TX_ACTIVE) {
        t.txSetRollbackString("Exception during transaction:" +
                              e.toString());
    }
    // Roll back the transaction and report if the rollback fails.
    try {
        t.txRollback();
    } catch (DrpcTxException tExc) {
        throw new Exception("Rollback failed:" +
                          tExc.toString());
    }
    // Report the reason for the rollback.
    throw new Exception("Transaction aborted:" +
                      t.txGetRollbackString());
}
```

*Figure 55. Sample Java Code for Basic Transaction Programming (TRPC)*

When you complete a transaction by using the `txCommit()` method, there is some delay between the time the gateway receives the last transactional RPC and the notification that the client has called the `txCommit()` method. You can reduce the number of messages between the client and the gateway by calling the `declareLastCall()` method before calling the final RPC in the transaction. This tells the gateway to begin commit processing as soon as the next transactional RPC completes successfully:

```
// ... code preceding the last RPC within a transaction
// ... transaction was started, more RPC were made
drpc.declareLastCall();
drpc.callRpc("server=/.:/foo rpc=withdraw " +
    "[in] acct [in] amount [out] balance");
// ... get values of out parameters (code omitted)
if (balance > 0) {
    t.txCommit();
else {
    // ... roll-back thetransaction in the usual way
```

## 7.4 Exceptions

Exceptions are used in Java to handle errors. A Java exception is an object that describes an error that has occurred. When an error occurs, an Exception object is created and thrown in the method that caused the exception.

The `try/catch` block is used to manage exception handling. Basically, you try to execute a block of code. If an error occurs, the system throws an exception. You can choose to catch the exception based on its type. When handling an exception, you can choose to throw the exception again to enable others to catch and handle it as well.

The following example shows the basic form of an exception-handling block:

```
try {
    // block of code
} catch (Exception e) {
    // code to handle exceptions of type Exception
    throw(e);    // re-throw the exception
}
```

DE-Light has defined two public exception classes. The `DrpcException` class includes exceptions that can occur during DE-Light operations. The `DrpcTxException` class includes exceptions that can occur during DE-Light transactional operations.

These classes encapsulate DCE, Encina, and DE-Light error codes as Java exceptions. Each exception contains an integer code number (drawn from the DCE, Encina, and DE-Light error spaces) and a detail message.

The detail message normally contains a translation of the error code, except when the exception was raised during a transaction. In that case, the detail message contains the transaction rollback string. (Note that rollback strings are not always available to every participant in a transaction.)

## 7.5 Java Client Security

Creating a DE-Light application involves communications between a client, a gateway, and a server. DE-Light security, which uses the SSL in Web browser applets, is used for communication between the client and the gateway. DCE security is used for communication between the gateway and the server.

### 7.5.1 Setting the DE-Light Security Level

DE-Light security is used in communications between the client and the DE-Light gateway when there is an HTTPS end point (for instance, http:prod_one[1234,5678]). DE-Light security uses the SSL in Web browsers to provide various levels of security.

When you use the supported browsers to run a DE-Light client as a Web browser applet, and that applet was loaded from a Web server, Java connections can be made only to the same host from which the applet was loaded. As a result, the Web server and the DE-Light gateway must be running on the same host.

If your DE-Light client applet requires security, the following requirements must also be met:

- The gateway must have been configured to use security. It must have been started with a secure end point and with a signed key ring file and password combination. Optionally, the gateway's security range can also be restricted by using the -S option with the drpcgwy command.
- The browser must have SSLV2 enabled. See the Release Notes for a list of the SSL-capable browsers that have been certified to work with DE-Light.
- The DE-Light client applet must request a security level that is compatible with the set of ciphers that are available in both the browser and the gateway. The set of available ciphers can be limited by export restrictions or by the configuration of the gateway.

You can set the DE-Light security level by using the `setSecurity()` method. The method takes the constants listed in Table 6 on page 168 as its values.

*Table 6. DE-Light Client Security Levels*

| **DE-Light Constant** | **Description** |
|---|---|
| SEC_NONE | Use no DE-Light security. |
| SEC_BEST | Use the highest available DE-Light security level even if that means no security. |
| SEC_INHERIT | Continuously inherit the DE-Light security level based on the DCE security level in use between the gateway and the server. |
| SEC_CIRCUIT_AUTH | Use SSL security for privacy and integrity protection only on the initial request (while establishing the connection). |
| SEC_PACKET_INTEGRITY | Use SSL security for privacy and integrity protection on the initial request and for integrity protection on all other requests. |
| SEC_PACKET_PRIVACY | Use SSL security with a cipher key that is 40 bits or greater. |
| SEC_ENCRYPT | Identical to SEC_PACKET_PRIVACY. |
| SEC_PACKET_PRIVACY_WORLD | Use SSL security using only US-exportable ciphers. |
| SEC_PACKET_PRIVACY_US | Use SSL security with a cipher key that is greater than 40 bits. |
| SEC_PACKET_PRIVACY_US_128 | Use SSL security with a cipher key that is 128 bits or greater. |
| SEC_MAX_VALUE | Use the highest SSL security level that is defined (currently 128 bits). |

If the DE-Light security level is set to SEC_INHERIT, the DCE security level in use between the gateway and the server is inherited for use between the client and the gateway.

To set the DE-Light security level, call the `setSecurity()` method:

```
drpc.setSecurity(value);
```

where *drpc* is a reference to a *DrpcConnection* object defined earlier in the client application, and *value* is one of the constants listed Table 6 on page 168. For example, to set the DE-Light security level to SEC_NONE, use the following code:

```
drpc.setSecurity(DrpcConnection.SEC_NONE);
```

If you are using a TCP or HTTP connection between the DE-Light Java client and the DE-Light gateway, the only possible value for `setSecurity()` is SEC_NONE.

### 7.5.2 Setting the DCE Security Level for the Gateway

The standard DCE security levels are available for use in communications between the DE-Light gateway and the Encina servers.

You can set the DCE security level by using the `setDceSecurityLevel()` method. The method takes the constants listed in Table 7 on page 169 as its values.

*Table 7. DCE Security Levels for DE-Light Gateways*

| DE-Light Constant | DCE Value | Description |
|---|---|---|
| DRPC_DCE_PROTECT_DEFAULT | 0 - Default | Use the rpc_c_protect_level_default DCE security level for communications between the gateway and the server. |
| DRPC_DCE_PROTECT_NONE | 1 - None | Use the rpc_c_protect_level_none DCE security level for communication between the gateway and the server. This DCE security level specifies that no authentication is performed, no tickets are exchanged, and transmissions are in the clear. |
| DRPC_DCE_PROTECT_CONNECT | 2 - Connect | Use the rpc_c_protect_level_connect DCE security level for communication between the gateway and the server. This DCE security level specifies that protection is performed only when the client establishes a relationship with the server. |

| DE-Light Constant | DCE Value | Description |
|---|---|---|
| DRPC_DCE_PROTECT_CALL | 3 - Call | Use the rpc_c_protect_level_call DCE security level for communication between the gateway and the server. This DCE security level specifies that protection is performed only at the beginning of each RPC when the server receives the request. |
| DRPC_DCE_PROTECT_PACKET | 4 - Packet | Use the rpc_c_protect_level_packet DCE security level for communication between the gateway and the server. This DCE security level ensures that all RPCs for a given secure connection are from the same DCE principal on the gateway. |
| DRPC_DCE_PROTECT_PKT_INTE G | 5 - Packet Integrity | Use the rpc_c_protect_level_pkt_integ DCE security level for communication between the gateway and the server. This DCE security level ensures and verifies that none of the data transferred between the gateway and the server has been modified. |
| DRPC_DCE_PROTECT_PKT_PRIV ACY | 6 - Packet Privacy | Use the rpc_c_protect_level_pkt_privacy DCE security level for communication between the gateway and the server. This DCE security level specifies that protection is performed as specified by all of the other levels and that each RPC argument value is also encrypted. |
| DRPC_DCE_PROTECT_MAX_VAL UE | 6 - Packet Privacy | Use the highest DCE security level that is defined for communication between the gateway and the server. Currently, the maximum is packet privacy. |

Set the DCE security level using the `setDceSecurityLevel()` method:

```
drpc.setDceSecurityLevel(value);
```

where *drpc* is a reference to a `DrpcConnection` object defined earlier in the client application, and value is one of the constants listed Table 7 on page 169. For example, to set the DCE security level to DRPC_DCE_PROTECT_NONE, use the following code:

```
drpc.setDceSecurityLevel(DrpcConnection.DRPC_DCE_PROTECT_NONE);
```

Notice that the DCE security level cannot be changed in the middle of a transaction.

**IMPORTANT:** The DCE security levels set through this method are the minimum DCE security levels that will be used. The server can request a higher security level, and the gateway will comply.

To obtain the current DCE security level, call the `getDceSecurityLevel()` method:

```
int level = drpc.getDceSecurityLevel();
```

### 7.5.3 Creating a Login Context

If your Encina server requires authenticated RPCs, your DE-Light Java client has to authenticate to DCE before attempting any communication with the Encina application. Because the Java client does not necessarily run within the same DCE cell, it uses the DE-Light gateway to perform the authentication to DCE. Therefore you have to create a DCE user for each client that accesses the Encina application.

A DE-Light Java client can authenticate to DCE through the gateway by using the dceLogin() method:

```
drpc.dceLogin(principal, password);
```

where *drpc* is a reference to a `DrpcConnection` object defined earlier in the client application. The values of *principal* and *password* have already been set, possibly by prompting the user using the Java applet to type them in. The arguments, principal and password, are strings that hold the name of the DCE principal to which you want to authenticate and the password for that DCE principal.

The following code demonstrates how to authenticate to DCE at the DE-Light gateway:

```
try {
    DrpcConnection drpc = new DrpcConnection(gwy_name);
    drpc.dceLogin(principal, password);
} catch (DrpcException e) {
    System.out.println("Login failed: " +
        e.toString() );
}
```

**IMPORTANT:** Set the DE-Light and DCE security levels before creating a login context to ensure that the login information is protected.

## 7.6 Loading Gateways with IDL and TIDL Files

In order for a DE-Light client to communicate with an Encina server, the DE-Light gateway has to be configured to understand the interface exported by the Encina server. This configuration is done by loading the TIDL file describing the TRPC interface into the gateway. As soon as the interface has been loaded the gateway is ready to receive requests from the client, interpret them, and forward them to the corresponding Encina server.

The DE-Light gateway can dynamically load TIDL files. It can also be started with a specified list of TIDL files that are loaded right away. You can specify which TIDL files you need to load by using multiple -L and -X options to drpcgwy.

Once the gateway is up and running, you can load additional TIDL files through the gateway administrative utility, `drpcadmin`:

```
drpcadmin list interfaces -gateway /.:/orders/enc_prod/server/drpc-gateway
drpcadmin load YOUR.TIDL -gateway /.:/orders/enc_prod/server/drpc-gateway
```

The first command shows you which TIDL files are currently loaded in the gateway /.:/orders/enc_prod/drpc-gateway. The second command loads the TRPC interface described in the TIDL file, YOUR.TIDL, into the same DE-Light gateway.

## 7.7 Short Example

Figure 56 on page 173 through Figure 58 on page 175 provide a short example of how to write a simple DE-Light Java client. In the example the client sends a single transactional request to an Encina Monitor server.

```
/**
 * This Java DE-Light example is intended to demonstrate the use of the
 * basic methods that are involved in creating an Encina DE-Light
Monitor
 * client.
 *
 * Note that in addition to the TIDL file Monitor clients require a
 * TACF file, which describes the server binding method, among other
things.
 *
 * HelloEncina is designed as a client for a client/server application
 * that supports a single TRPC with the following the TIDL signature:
 *
 *      [transactional] void helloEncina (
 *                        [in,string]     char client_greeting[],
 *                        [out,string]    char server_reply[100]
 *       );
 *
 * This example client is intended for illustrative purposes only and
 * no corresponding server is provided.  See the ListRpc, Greet, and
 * Telshop examples for a more detailed look at writing Java
 * DE-Light applications.
 */
import COM.Transarc.Delight.*;
import java.io.*;
public class HelloEncina {
    /* DE-Light connection information */
    private static DrpcConnection drpc;
    /**
     * The main method is called to run this standalone applet.
     *
     * @param arg  Specifies the DE-Light gateway name
     *             to which the client is to bind.
     */
    public static void main(String arg[]) throws Exception {
        // Check the argument.
        if(arg.length < 1) {
            System.out.println("Usage: java HelloEncina" +
                        " gateway-name");
            System.exit(1);
        }
```

*Figure 56. (Part 1 of 3) Example of a Simple DE-Light Java Client*

```
// Obtain a gateway connection, specifying the gateway
        // location.
        try {
            // The gateway argument should have the form:
            // proto:machine[proto-specific], e.g., tcp:gecko[1234]
            drpc = new DrpcConnection(arg[0]);

drpc.setDceSecurityLevel(DrpcConnection.DRPC_DCE_PROTECT_DEFAULT);
        } catch (DrpcException e) {
            // Control arrives here if the gateway connection fails.
            drpc = null;
            System.out.println("Failed to obtain DrpcConnection: " +
                        e.toString());
            e.printStackTrace();
            System.exit(1);
        }
        // Get a reference to the default data dictionary
        // and the transaction context for the connection.
        DrpcDictionary dict = drpc.dictionary();
        Transaction t = drpc.transaction();
        // Make the TRPC.
        try {
            // Begin the transaction.
            t.txBegin();
            // Store the [in] parameter in the data dictionary.
            dict.put("client_greeting", "Hello server.");
            // Issue the call to the gateway/server.
            // Note that the server name is not specified for
            // Encina Monitor applications.
            drpc.callRpc("interface=hello rpc=helloEncina" +
                     " [in] client_greeting [out] server_reply");
            // Retrieve the [out] parameter from the data dictionary.
            String reply = (String) dict.get("server_reply");
            System.out.println("The server said: " + reply);
            // Attempt to commit the transaction.
            t.txCommit();
```

*Figure 57. (Part 2 of 3) Example of a Simple DE-Light Java Client*

```
} catch (Exception e) {
            // Control arrives here if either the RPC failed, the
            // transaction commit failed or there was some other
            // runtime exception.
            if (t.txState() == t.TX_ACTIVE) {
                t.txSetRollbackString("Exception during transaction:" +
                                  e.toString());
            }
            // Roll back the transaction and report if the rollback
fails.
            try {
                t.txRollback();
            } catch (DrpcTxException tExc) {
                throw new Exception("Rollback failed in helloEncina:" +
                                       tExc.toString());
            }
            // Report the reason for the rollback.
            throw new Exception("Transaction aborted in helloEncina:" +
                                    t.txGetRollbackString());
        }
    }
};
```

*Figure 58. (Part 3 of 3) Example of a Simple DE-Light Java Client*

# Part 3. Case Study

How can I design my application using Encina technology? How do I start my project? What are the infrastructure considerations I have to think about? Where do I start coding? How do I code all the different interfaces? How can I integrate an existing database into my new, multitier, distributed transaction system architecture? How can I enable my distributed transaction system architecture to the Web?

In Part 3 we discuss all of these questions and many more. We show you through a case study how to architect, design, Web-enable, and code an Encina multitier distributed transaction application. Although the application we develop in this part is not a finished fully functional sample application, you can download most of the modules we discuss and describe from http://www.redbooks.ibm.com. There you must go to "additional material" and click on book number SG245241.

# Chapter 8. Analysis and Architecture Phase

In this chapter we focus on issues related to understanding the business problem at hand and establishing the high-level architectures for both the software and the infrastructure. This phase begins with an examination of the business processes at work in the environment and of the existing computing infrastructure. Once these are well understood, we proceed with the development of an architectural model for the application and for the infrastructure in which the application will run.

## 8.1 Business Problem Analysis

The first and most critical step in the process of developing the case study application is to fully understand the needs that exist and the objectives of the effort. This requires that we first step back from the technical issues and examine the business itself. When we look at the situation at this level we find that a set of business processes drives the business in its current form. There is also a new set of business processes, envisioned by the leaders of the business, that represent an evolution from the current business. The new set of business processes requires that current information systems be modified or new information systems be developed.

In addition to the business processes, there is also a set of factors that are important to the business or to the leaders of the business. Such factors might include a desire to develop a competitive edge based on innovative technology or a desire to minimize risk by implementing only "industry standard" technology. Misunderstanding the drivers behind the business can easily result in inappropriate decisions during the application development process.

### 8.1.1 Case Study Business Problem

Our case study customer, ACME Widgets, produces and sells a range of products. Its current business is based on selling its products through retail outlets, not directly to the general public. However, in an effort to expand its business, ACME wants to begin marketing its products directly. It does not intend to open retail stores, however. Instead it will focus on marketing activities that will promote sales through the Internet and by telephone. ACME recognizes that new information systems have to be developed to support this new type of business but wants to leverage its exiting systems as much as possible. Clearly ACME sees that an innovative application of technology such as direct sales through the Internet can be a competitive advantage. However, the leaders of the business want to preserve their existing investments in hardware, software, and application development as much as possible. They also want to base the new development on proven technologies that will allow for robust, scalable systems.

On the basis of conversations with critical business leaders and reviews of company documents, we produced a model of the new business processes required to implement this new line of business. For the purposes of our case study, we have taken a very limited scope on the problem so that we can show the entire process in a reasonable space. However, our experience has been that the techniques we are using here will continue to work well with far larger problems. It is just as important to use methodologies that scale up for larger problems as it is to build software systems that scale up for higher processing volumes.

### 8.1.2 The Use Case Model

For this case study we chose to use a business process analysis technique known as *use case analysis*. This technique originated as part of Jacobson's work with object-oriented software development but has become widely accepted as a business process engineering tool. For complete descriptions of the technique see Jacobson's *Object-Oriented Software Engineering: A Use Case Approach*, and *The Object Advantage*, both published by the Addison-Wesley Publishing Company.

The intent of the use case analysis is to produce a model of the business in terms of actors and use cases. Actors are the people and things that are affected in some way by a business process. Use cases are small-grained, individually meaningful business processes. A use case model contains of all the actors that are important within the scope of the problem being considered. The model also contains all of the use cases which, taken together, represent the complete set of processes that are implemented by the business - again within the scope being considered. The relationships between the actors and use cases are shown on a diagram known as a *use case diagram*. In addition, the model contains complete narratives for each of the use cases. The narratives describe, in business terms, the activities that are performed as part of the use cases.

Using a tool specifically designed for use case modeling facilitates construction and maintenance of the model. We use a tool called *Rational Rose* from Rational Corporation to develop the case study model.

### 8.1.2.1 Actors

The actors in the use case model are: External Customer, Order Operator, Order Processor, and Product Database. These are roles that are played by people and things considered to be "users" of the new system.

**External Customer** - The External Customer actor represents a person accessing our system to build and submit an order for one or more of our products. There is no assumption that the customer has ever placed an order before.

**Order Operator** - The Order Operator actor represents an employee of our company who is responsible for creating orders on behalf of a customer. The Order Operator may work with the customer through any communications medium, such as phone, fax, or e-mail.

**Order Processor** - The Order Processor actor represents an employee of the company who is responsible for fulfilling the orders that are placed by, or on behalf of, the External Customer.

**Product Database** - The Product Database actor represents the existing product database in use by the company. This database is shown as an actor because it is considered to be a fixed component that is not subject to change in the new system.

### 8.1.2.2 Use Cases

The use cases that describe the granular business processes that are part of the new business area are:

**Place an Order** - The Place an Order use case is responsible for the sequence of activities that are required to be executed as part of creating, constructing, and submitting an order. Here is the sequence of activities: The requester uses an interface (Web-based, GUI, or command-line) to provide one or more product selections. The product selection includes the product identifier and a requested quantity. The requester then provides information about shipping and payment, such as the ship-to name and address and credit card information. All of the information submitted is included in the "order data store." Once all of the information is provided, the order is marked as completed. Finally, the order is verified through the Verify Order use case.

**Maintain Product Info** - The Maintain Product Info use case is responsible for providing access to the product info data store and for updating the data store. The access requirements are:

1. Produce a complete list of products to be produced that includes the product ID, description, and price for each product, and the current inventory level for each product.
2. Update the inventory level for a given product by a specified amount.

**Verify Order -** The Verify Order use case is responsible for examining the payment information provided as part of an order, specifically the credit card information, and verifying its accuracy. When the accuracy of the information is verified, the order is marked as "ready to process" in the order info data store. If the order cannot be approved because of incomplete or inaccurate information, it is marked for review. The Mark Order for Review use case is responsible for the activities involved in marking the order for review.

**Review Order** - The Review Order use case is responsible for providing access to the information about orders that have previously been placed. The use case is initiated by either the External Customer actor or the Order Operator actor. The use case requires that an order number be provided. The use case accesses the stored information about the specified order and makes it available to the requester.

**List Orders** - The List Orders use case is responsible for accessing the order data store and producing a list of the basic order information about each order, including information about the status of the order, the recipient of the order, and the payment method for the order.

**Mark Order for Review -** The Mark Order for Review use case extends the Verify Order use case when a decision made within the Verify Order use case calls for a manual review of the order. Review types can be either a "warning" review or a "failure" review, depending on the severity of the problems with the order information.

**Print Review Report** - The Print Review Report use case is responsible for producing a formatted report of all orders that have been flagged for review. Information that is critical for a manual review of the order is included in the report. Additional information can be retrieved through the Review Order use case. The system will produce the report whenever a user requests it.

#### 8.1.2.3 Use Case Diagram

The relationships between actors and use cases are captured in the use cases, but a more meaningful and concise representation can be shown in a use case diagram. Figure 59 on page 183 shows the use case diagram that summarizes the information for our model.

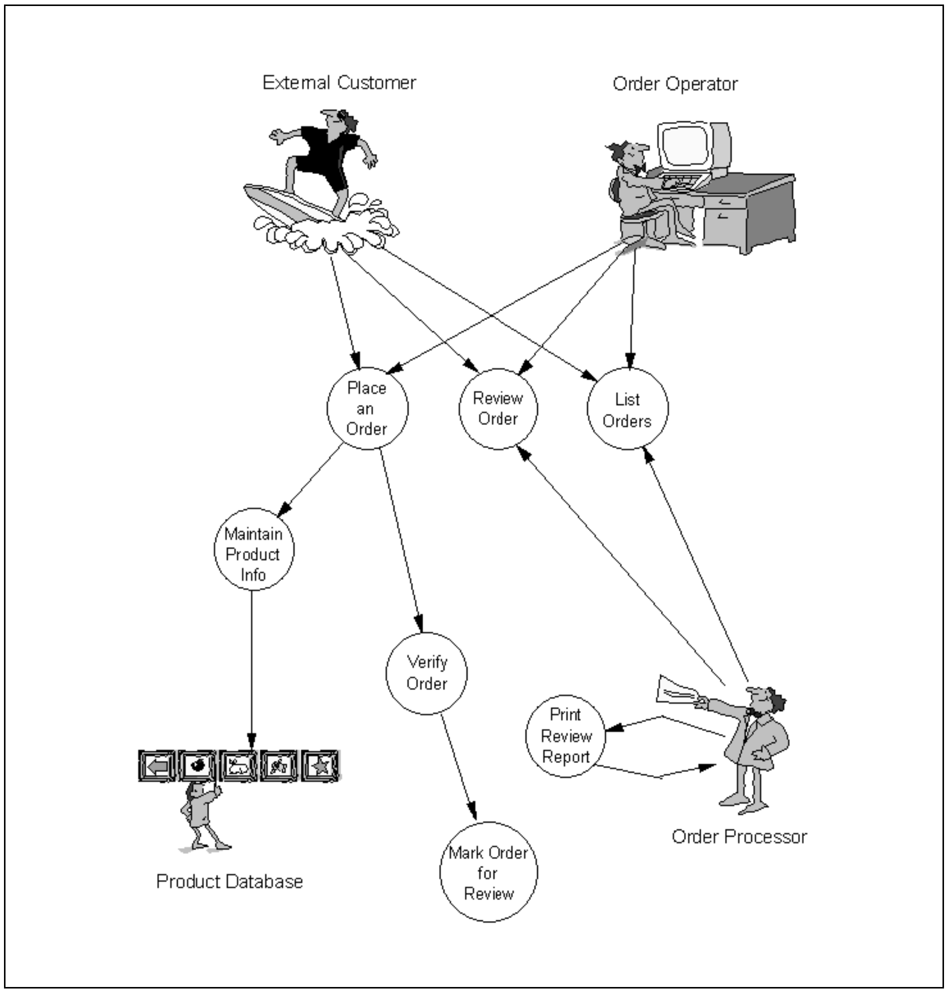


*Figure 59. Use Case Diagram*

## 8.2 Existing Infrastructure Analysis

To develop an appropriate strategy for building a new information system, the critical aspects of the existing infrastructure must be identified and researched. Among the areas that must be considered are the existing systems or data stores that will interface with the new systems, the hardware environment, the network environment, and the development standards for languages and tools.

### 8.2.1 Systems and Data

Our initial analysis has identified that there is an existing product database with associated programs for accessing the database. The database and the associated program logic will be preserved. Our new system will use the existing programs to access the product database. Our research reveals that the product database system is based on VSAM and is accessed through CICS programs. The CICS programs are layered into presentation programs and access programs that communicate through APPC (LU 6.2).

No other existing systems are considered as part of the limited scope of our case study. However, in more realistic scenarios it is common for a number of existing systems, possibly on disparate platforms, to be actors relative to a new system.

### 8.2.2 Hardware Environment

The hardware environment into which we will place our new system consists of a host environment running MVS/ESA, RS/6000 servers running AIX, and microcomputers running Windows NT.

### 8.2.3 Network Environment

The network environment in place consists of both a SNA network and TCP/IP network. One of the AIX servers participates in the SNA network with the host. All of the AIX servers and Windows NT computers participate in the TCP/IP network. There is no existing DCE cell in place.

### 8.2.4 Languages and Tools

We have determined that the application development group has expertise in COBOL, CICS, and C programming. Within the AIX and Windows NT environments, the tools in use are the C-Set product suite and the Visual Studio product suite, respectively.

## 8.3 Application Architecture

The next step is to develop a high-level, architectural description of the new system. All of the information gathered previously feeds into the considerations we now go through in developing the application architecture. Typically this is an iterative process where the designers propose ideas for portions of the architecture and then test these ideas against the requirements. As portions of the architecture emerge from this process, the new parts of the architecture must be tested for integration with the other architectural pieces as well as against the requirements.

## 8.3.1 Architectural Decisions

On the basis of the business requirements and drivers identified, we have determined that the new application will be built as a distributed processing system. We are going to be working with multiple interface points into our application (in-house and from the Internet) and accessing data from multiple sources (existing product data and new order data). A multitier distributed transaction system architecture is well suited to this type of situation.

### 8.3.1.1 Distribution of Responsibility across Tiers

The need to support multiple types of interfaces, particularly a Web-based interface, leads us to focus on a "light client" approach to the architecture. The "light client" concept is based on the idea that the component of the system that executes on the end user’s computer should focus on gathering input from the user and presenting the results. All processing will be done by "application server" programs with which the client programs communicate. This approach allows the business logic to be coded only once (as part of the application server program), rather than recoded for each different client implementation.

### 8.3.1.2 Transactional Requirements

The need to deal with multiple data sources and to ensure that updates are always synchronized between data sources leads us to conclude that a transaction processing monitor environment is an appropriate architectural choice. The use of a transaction processing infrastructure will enable us to use the existing product database and to implement a new order database without introducing data integrity problems between the data sources.

### 8.3.1.3 Synchronous and Asynchronous Processing Requirements

A review of the use cases shows a number of interactions between actors and use cases that are clearly synchronous in nature. For example, the Review Order use case is used by the External Customer actor to see the contents of the current order. This must be implemented synchronously. However, we will implement the Verify Order use case as an asynchronous process to avoid potential processing bottlenecks. We will implement a queue-based communication between the process that implements the Place an Order use case and the process that implements the Verify Order use case.

### 8.3.1.4 Encina Capabilities Supporting Architecture Needs

All of the critical architectural issues to be dealt with here are handled by the Encina product suite. We can use the Encina Monitor environment to implement application servers that perform the logic required by the application. We can use the standard Encina and DCE client libraries to implement a Windows-based client program for in-house users. We can use the DE-Light client and DE-Light Gateway to implement a Java applet client for use by external customers over the Web. We can use the standard RPC mechanisms for synchronous processing and an RQS server for asynchronous processing. We will use the PPC Gateway component of Encina to provide transactional access to the product database on the host. We will use the XA resource management capabilities to implement our new order database in a relational database in the distributed environment.

#### 8.3.1.5 Mapping Use Cases to the Application Architecture

The use cases can be divided roughly into those that are directly accessed by the External Customer or by the Order Operator as the External Customer's proxy, and those that are accessed by the Order Processor. Because we have opted for a thin client approach in our architecture, the mapping of use case responsibilities will be to application servers, exclusively. In this case we will choose to have two separate application server types defined in our architecture, with the responsibilities of each application server divided along the use case boundaries identified in the previous sections.

The application server which supports the External-Customer-related use cases will be called OrderProcServer and will implement the following use cases:

- Place an Order
- Review Order
- List Orders
- Maintain Product Info

The application server that supports the Order Processor use cases will be called the VerificationServer and will implement these use cases:

- Verify Order
- Mark Order for Review
- Print Review Report

Notice that we place the Verify Order use case in the application server that supports the Order Process use cases even though it is indirectly related to the Place an Order use case, which is in the application server that supports the External Customer use cases. Our reasoning here is twofold: first, the requirements have already identified that there is a need to make the verification processing asynchronous, and second the distribution of responsibility would be skewed toward the OrderProcServer if it handled the Verify Order use case in addition to the others already assigned to it. We are applying a heuristic consideration at this point that is intended to avoid potential performance problems down the road. Our experience has shown that a division of responsibility along the lines of related use cases and an even distribution of responsibilities across application servers are good starting points for an application server design.

### 8.3.2 Application Architecture Diagram

The architectural decisions described above are captured in a high-level application architecture diagram (see Figure 60 on page 188). The major components of the application are shown along with the communication mechanisms that are used between the various components. Note the focus here is on the application components and not on the hardware. One of the strengths of Encina is that it provides great flexibility in how a system is actually implemented. For example, our application architecture calls for two Encina monitor application servers to be built. When these are implemented, we may choose to run multiple instances of each server, possibly on different computers.

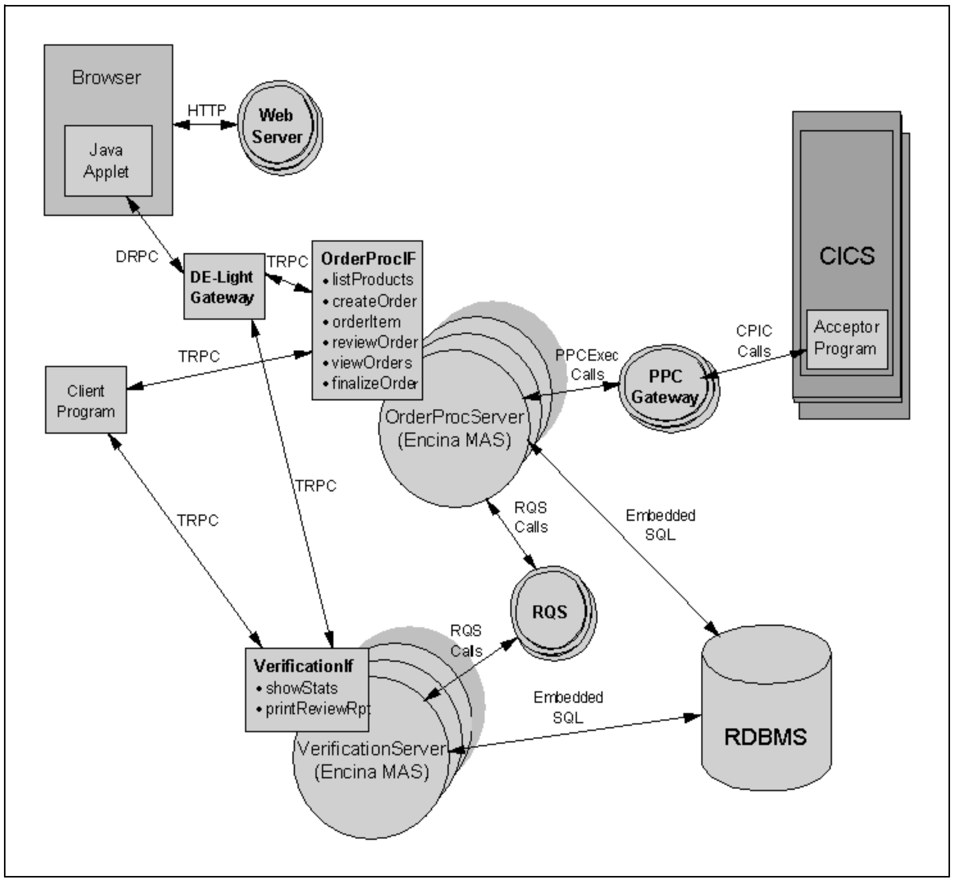


Figure 60. Application Architecture Diagram

## 8.4 Infrastructure Architecture

After considering the existing hardware and networking infrastructure and the application architecture, we have to develop an infrastructure architecture. In an Encina/DCE environment the infrastructure architecture is largely concerned with the DCE cell structure, the Encina cell structure, and the management strategies that affect the assignment of application components to physical computers.

The infrastructure architecture referred to here is for the production deployment of the application. There will, of course, also need to be an infrastructure for the development environment.

### 8.4.1 DCE Cell Structure

DCE provides the fundamental facilities on which Encina and the application depend. A number of considerations led us to conclude that the best choice for a production implementation is to implement the entire application within a single DCE cell. Security services and naming services are cell-specific. An Encina cell cannot span DCE cells. Although it is possible to implement an application that spans multiple DCE cells, the mechanisms that must be coded to make this work are not trivial. Thus, unless you find extraordinary circumstances in the geographic distribution of computing devices or have special security considerations, we recommend that you implement a single DCE cell for the production application. Typically, many applications will share the same DCE cell.

### 8.4.2 Encina Cell Structure

An Encina cell differs from a DCE cell in that only the computers that will run application servers or Encina services are part of the Encina cell, not the computers that simply run client programs. However, all of the computers that are part of the application, including the clients, must be part of the DCE cell.

Thus the decision about Encina cell structure consists of deciding how to divide the computers that are part of the DCE cell into one or more Encina cells. The considerations that drive this decision are primarily related to fault tolerance and fail-over. Because there is a single, nonredundant Encina Cell Manager process, the architecture must make provisions for handling a loss of this critical component.

One fail-over strategy is to implement multiple Encina cells, each of which contains a complete set of the server components for the application. The client can then respond to a loss of communication with the Encina cell by rebinding to another cell. Because the other cell has identical components, you do not have to change the client code to use the alternative set of servers. In fact, if this strategy is implemented, there would typically be active clients using each of the Encina cells. If a cell becomes unavailable, the clients using that cell would join the other clients that were already attached to the other cell. With this strategy computing resources are well utilized, but there could be a performance degradation if all clients are forced to use the same Encina cell.

Another fail-over strategy is to implement a single Encina cell for the application but provide for quick recovery of the Encina Cell Manager in the event of a problem. This can be done through a "hot swappable" hardware implementation, or by having multiple Encina Cell Managers defined on different computers in the cell (with only one actually running at any point in time).

The choice of fail-over strategies hinges on the critically of the application and the expense involved in providing the level of fault tolerance. For our case study application we will implement a single Encina cell.

# Chapter 9. Design Phase

During the design phase of the development life cycle we focus on the internal structure of the various components of the application and of the infrastructure. We make decisions regarding the layering of the software, determine the structure of the new data, define the interfaces between the application components, make decisions about error handling, and determine the placement of the infrastructure components on the available computers.

## 9.1 Application Design

The design phase consists of a number of activities that are carried out somewhat in parallel with iterations that successively refine the design of the system. The activities are detailed analysis of each component, design of the application logic, design of the application data sources, design of the server interfaces, and design of the transactions within the system.

One of the discoveries we made during the business problem analysis was that the programming expertise of the application development group was largely in C. For this reason, we decided to develop the application in C, using the standard Encina API as opposed to using C++ with the Encina++ API. Making this decision early on was significant because of the differences in the capabilities between Encina and Encina++. Specifically, the standard Encina API does not support dynamic server objects, which we might have considered using for internal state management.

Despite the decision to implement in a nonobject language, there is still benefit in doing the analysis and design of the application logic in terms of business objects. Object-oriented analysis and design has proven to be more effective in most cases than traditional techniques. The steps of this phase will be documented in Rational Rose, building on the results of the use case analysis described previously.

## 9.2 Object Modeling

The object modeling process is concerned primarily with identifying the business objects and implementation objects whose data and behavior jointly implement the overall functioning of the system. The information about the objects that is developed in this activity is captured in a model, sometimes referred to as the *object* model. The object analysis process contributes the information about the business objects and their structural relationships that is captured in a class structure diagram. The object model is refined by the object design process, which focuses primarily on the behavior of the objects and the functions required to provide the behavior.

An examination of the use case model reveals some, if not all, of the business objects in the system. Most of the structural relationships between the business objects are revealed as well. Reviewing an initial object model with the key business people allows us to fill in any missing information. The class structure diagram in Figure 61 on page 192 shows the results of this process for our case study.

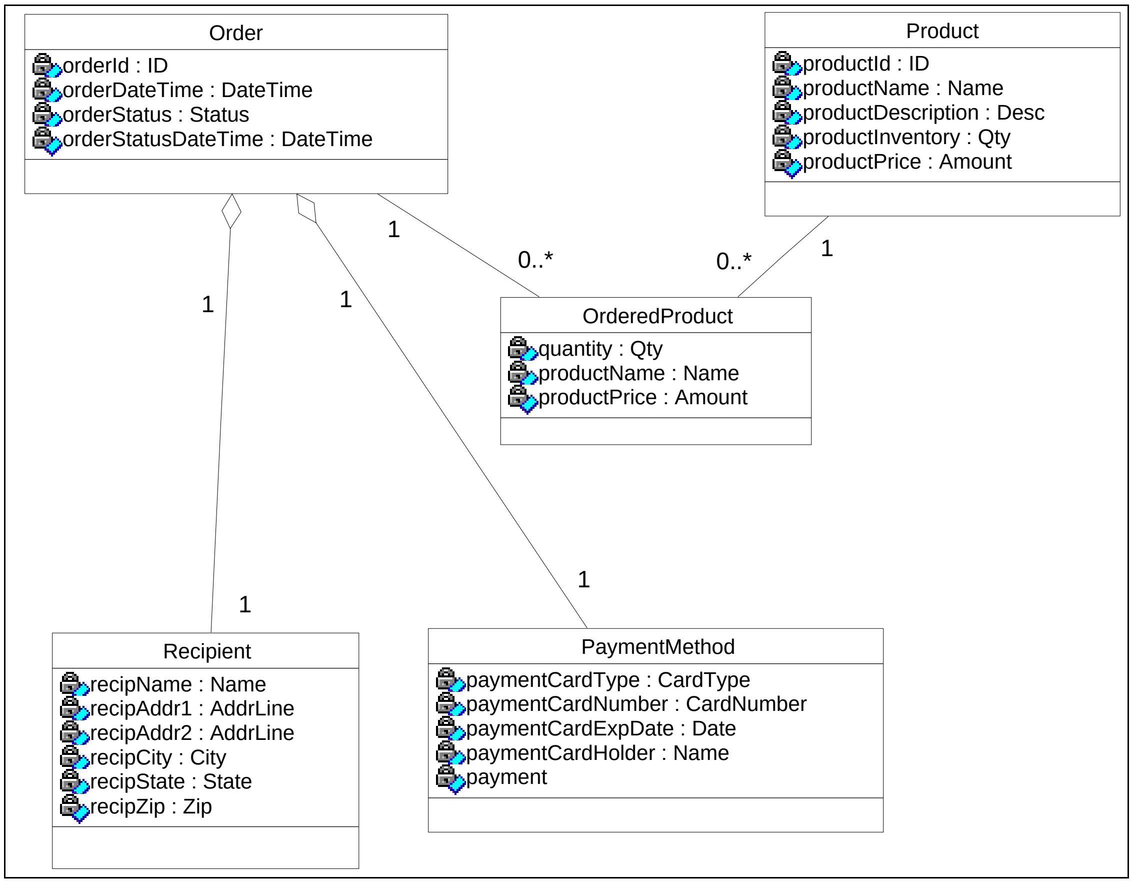


*Figure 61. Business Objects Class Structure Diagram*

The class structure diagram at this point shows the business classes and the data associated with each class. The object design will identify the classes needed to implement the working system and fill in the functions associated with each object. The data design process will use the information in the class structure diagram to determine a database design for the new data.

## 9.3 Object Design

The object design process results in a refined object model. The object model is expanded to include objects that are introduced for implementation (rather than business) reasons and the functions that implement the responsibilities of each of the objects. In our case study, the implementation objects that we will introduce are related to the multitier character of the application.

The functions that are required to implement the behavior of the system are typically defined through a process called *event trace modeling,* which maps the sequence of function calls that perform a larger-scale process. We will use event trace modeling here.

### 9.3.1 Adding Implementation Classes

The nature of our application is that it will consist of a number of correlating components, several of them implemented as Encina Monitor application servers.

To represent each of the Encina Monitor application servers in our system, we will introduce a class. The OrderProcServer will directly respond to client requests, in the form of RPCs, so we will also introduce a class to represent this interface, called OrderProcServer. Each of our data sources will be represented by a class as well. These classes are OrderDB, ProductDB, and VerifyQueue, representing, respectively, the new database containing the order information, the existing product database on the host, and the RQS queue used for asynchronous messaging. Figure 62 on page 194 shows the relationships among the implementation classes.

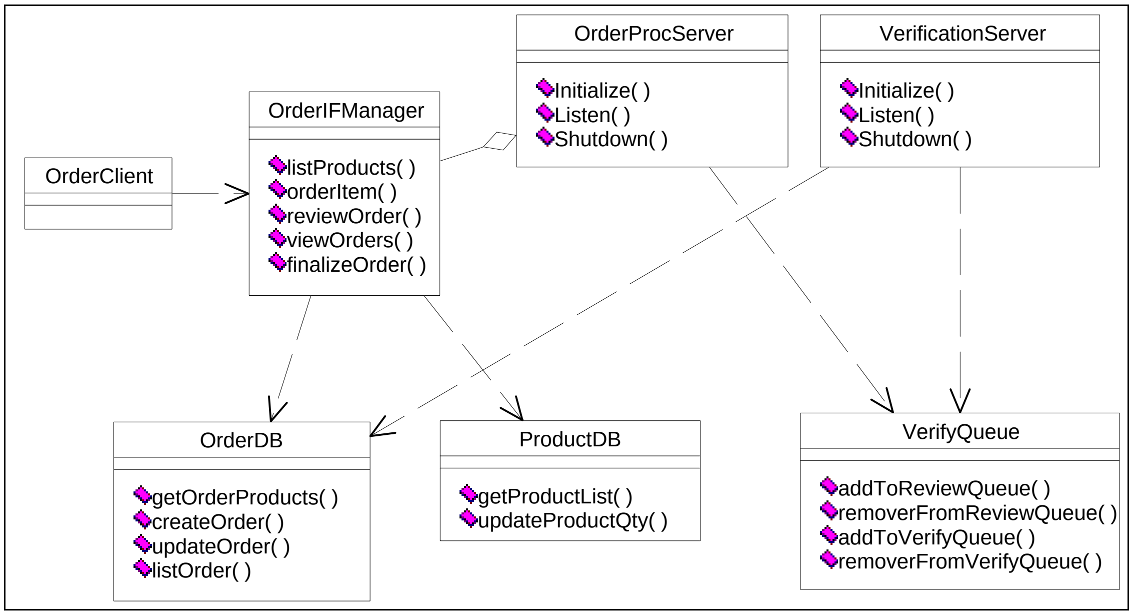


*Figure 62. Implementation Class Diagram*

We opted to use separate classes to represent the order database, the product database, and the verification messaging queue to gain some flexibility in making implementation decisions. We have identified the need to use a relational database for some of our data storage needs. We have not made any decisions as to the particular database product that we will use. By isolating the responsibility for accessing the relational database to one unit of code, we can much more easily convert between implementations based on different products. Similar considerations will cause us to isolate the processing of the queues and of the host access. We might want to be able to convert our application to use MQSeries as the queue manager instead of RQS, or to use some mechanism other than PPC to communicate with the host.

From an object modeling perspective, we can consider the implementations of different versions of the same functionality to be subclasses derived from an abstract base class. The extensions to the implementation class diagram in Figure 63 on page 195 shows some of the possible subclasses for different implementation decisions.

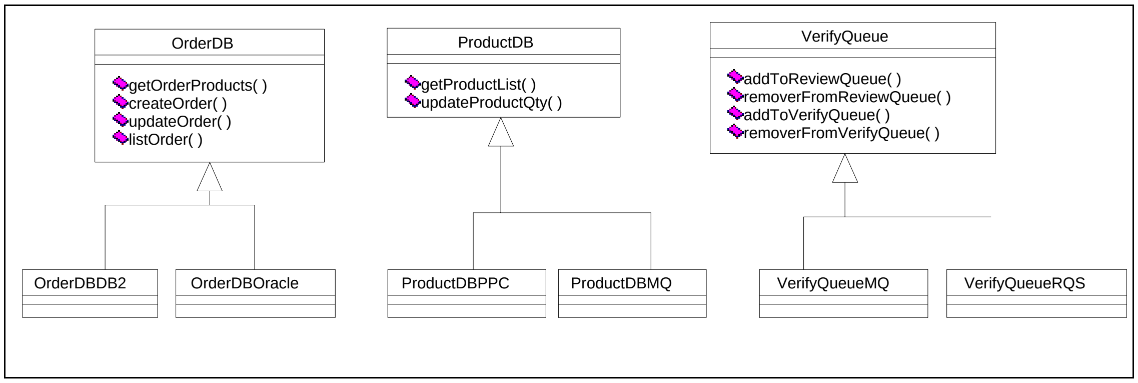


*Figure 63. Extensions to the Implementation Class Diagram*

## 9.3.2 Mapping the Functions

Each of the operations that our OrderProcServer will process is represented by a function of the class that represents the server's interface, OrderProcIFManager. Note that the VerificationServer does not have an interface that will be called from a client, and thus there is no corresponding interface object for this server.

Each of the functions in the interface will be examined in more detail during this phase of the lifecycle, and a diagram showing the sequence of function calls required to implement the initial client request will be produced. These diagrams are referred to as interaction diagrams, or event trace diagrams. In the interest of space, not all of the diagrams will be shown. The diagram for the `finalizeOrder` function is shown in Figure 64 on page 196. This diagram shows one of the most complex functions we will need to implement, and serves as a good example of how the transition will be made from the model to the actual implementation of the software.

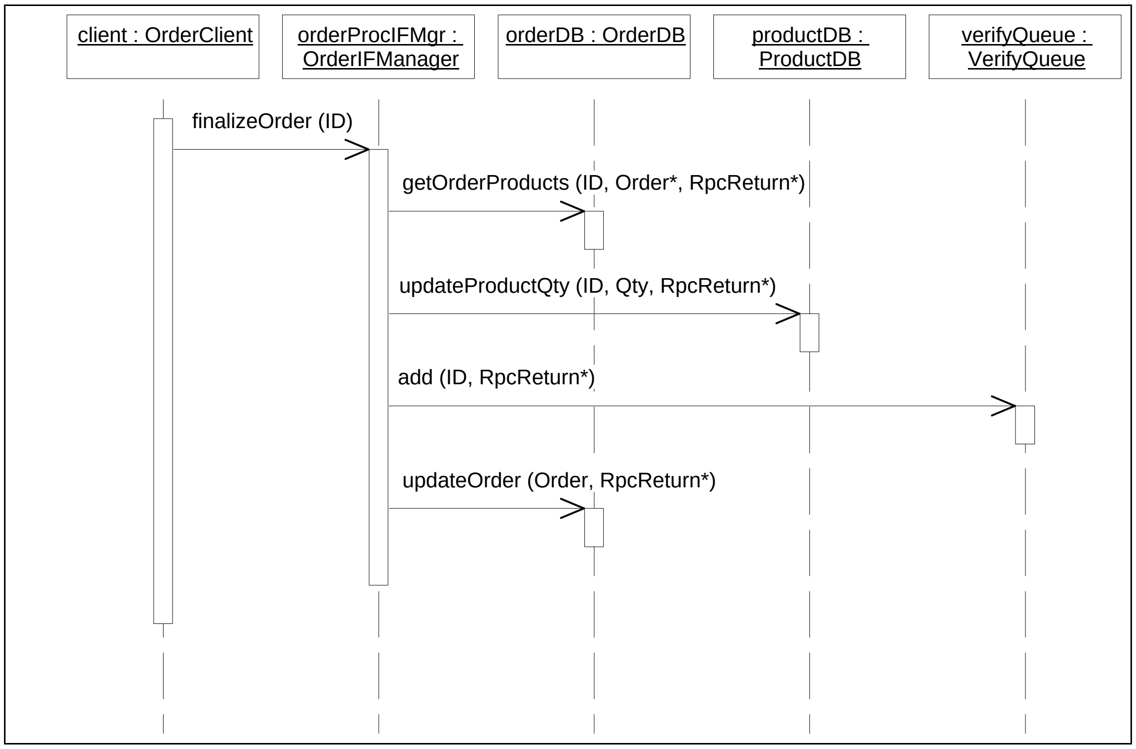

client : OrderClient
orderProcIFMgr : OrderIFManager
orderDB : OrderDB
productDB : ProductDB
verifyQueue : VerifyQueue
finalizeOrder (ID)
getOrderProducts (ID, Order*, RpcReturn*)
updateProductQty (ID, Qty, RpcReturn*)
add (ID, RpcReturn*)
updateOrder (Order, RpcReturn*)

*Figure 64. Interaction Diagram for the finalizeOrder Function*

Figure 64 on page 196 shows that the initial client request, finalizeOrder, is made by passing the ID of the order as a parameter. The implementation of this request proceeds to get the identified order from the order database, update the product inventory in the product database for each of the products on the order, add an entry on the verification queue for the asynchronous request to the verification server, and update the order (to change the order status). This sequence is shown as arrows between the lines representing the objects. Each line is labeled with the function on the receiving object that is being executed.

These diagrams work at a lower level of detail than those in the analysis phase but are still at a fairly high level. Within each of the functions invoked on the objects, for example, the getOrderProducts function on the OrderDB object, there is still significant complexity of implementation. It is up to the designer to determine the right level of detail to document at this point. In general, the interactions between components should always be captured, but the internals of the component's implementation will be detailed only if it is deemed to be complex enough to warrant the effort at this point. In our finalizeOrder diagram, the implementation of the `getOrderProducts` function of the OrderDB will involve issuing database commands to a relational database. The `updateProductQty` function of the ProductDB will involve a PPC conversation with the host. The `add` function of the `VerifyQueue` will involve issuing RQS commands to an RQS server. We cover these issues in detail in Chapter 10, “Development Phase” on page 205.

## 9.4 Common Application Components and Standards

The identification of common application components that can be standardized and shared throughout the development team is frequently left for the implementation phase of a project but really should be addressed at this point. On a project of any size, there will be some division of labor (a topic addressed in some detail in Chapter 10, “Development Phase” on page 205) among developers. If we can identify aspects of the development that can be standardized and reused instead of being “built from scratch” by each team of developers, we can save development time and produce a neatly structured application. Given that our application architecture places most of the complexity of the system into the application servers, we must focus on the application servers to identify commonalities. Among the candidates that can be built as shared code modules or adopted as standards in the coding are:

- A standard error message structure and conventions for RPCs
- A module to perform logging of messages to the server’s output log
- A standard set of functions for server startup and processing
- A module to generically process the results of Encina function calls
- Standardized routines to access databases and other data stores

On a typical project, additional opportunities for developing common modules will be identified during the development. We expect this to be the case and will not consider it to be a failure of our foresight, but rather a test of our resolve. It is easy to lose your focus on the “big picture” during the detailed phases of implementation. This loss of focus causes you to miss or ignore cases where you really should go back and create a new shared module. When you decide to add a new common module during the implementation phase, you will have to modify some existing code to take advantage of the new module. In our experience, it is has always been worth the effort to do this.

## 9.5 Data Design

The design of the new database for order information can proceed once the class structure diagram of the business objects is complete. As we noted earlier, many of the tasks in the design phase are conducted iteratively and somewhat in parallel. The initial design of the database would probably be done once the class structure diagram is complete, and before the interaction diagrams are developed. Then, issues that arise during the design of the interactions may cause the class structure and thus the database design to evolve.

Our model shows that we have a collection of data concerning the order itself and associated collections of recipient data and payment data. The relationships between the classes representing this data are basically “one-to-one," but allowing for the recipient and payment collections to be optional. In a situation like this, the default translation between the objects and the database design is to combine the collections into a single database table. We will take this decision here and call for a single table, CustOrder, which will have columns for each of the data items from the Order, Recipient, and Payment objects.

The Order object is related to Product information through an intermediate object, OrderProduct. This situation is interesting due to the fact that the Product information will not be part of our new data because it already exists in the host database. We do need to make provisions for the OrderProduct information because it does not exist as part of the host product database. Therefore we have chosen to include a table named OrderedProduct in our new database design and to relate this table to the CustOrder in a “one-to-many” fashion that allows the relationship to be optional. Typically, all that would be done as part of this transition would be to include an orderId column in OrderedProduct as a foreign key that establishes the “one-to-many” relationship to the Order table, and a productId column that does the same thing for the relationship to the Product table. However, since the Product side of the original relationship between Orders and Products is not part of our database, we have chosen to replicate some of the Product information in our new database, specifically the productName and productPrice data items. In database terms this is a *denormalization* of the design. Denormalizing a database design is essentially a calculated risk. Duplicated data has the potential to get out of synchronization and introduce data integrity problems. However, having data readily available for processing instead of incurring additional time to fetch it from the exclusive source can yield significant performance improvements. Ours is clearly a case where denormalization is warranted, because the alternative is to incur additional calls to the host system for Product data as part of the `reviewOrder` function and elsewhere. In any event, because the productId is required to be part of the OrderedProduct table but the RDBMS cannot treat it as a normal foreign key, we have already introduced a denormalization of sorts. So, based on the considerations above, we have two database tables, CustOrder and OrderedProduct.

During our examination of the interactions that implement the interface of the OrderProcServer, we were faced with the problem of how new order identifiers are generated. The `createOrder` function requires that an identifier be generated, but it is not clear where or how this should be done. In a simplistic sense, we have three choices for where the ID will be generated: by the client, by the application server, or by the database. The problem that must be solved here is to ensure that the identifier generated is unique within the system. If we attempt to generate the identifier within the client program, we have to figure out how to keep one instance of the client program from generating the same identifier as another instance of the client program in use by another user. Although algorithms could be devised to guarantee a unique identifier in this situation, they are difficult to implement. Considering the application server as a possible generator of the order identifier raises the same issue that we discussed for the client. Remember that there will usually be multiple instances of the application server program running concurrently. So we are left with the database as the most likely candidate for being the generator of the identifier. As a rule of thumb, it is good practice to allow the RDBMS to perform activities that are directly related to data integrity, such as key generation and referential integrity checking.

There are several strategies for implementing key generation within a RDBMS. Many database engines provide a built-in data type that can be used as a primary key. These built-in data types are known by various names, such as identity (Sybase) or sequence (Oracle). To use this facility, you simply define the column to be of the appropriate type, and the generation of keys is automatic. A problem with this approach is that it is somewhat proprietary, or vendor-specific. And once the RDBMS generates the key, you typically have to take additional steps to figure our just what key the RDBMS generated for you. Another strategy that is widely used and is not proprietary to any DBMS is the use of a special “next key” table to keep track of the keys that have been generated. The next key table is the single source for identifiers and thus can assure uniqueness. In an effort to avoid being database specific in our case study application, we will implement the next key table approach to key generation.

The final transition issue to be addressed is the determination of the data types to use for the database columns. In the object model, the types were left purposely vague to avoid dealing with too many details in the early stages of the process. Now is the time to make the decisions about data types and sizes. We have opted to keep the case study application simple by choosing a limited number of different data types. We will use an integer data type for identifiers. We will use char data types of various lengths for most of the other columns, with the exception of the quantity column, which will be integer; the price column, which will be a decimal type with a precision of 9 and a scale of 2; and the date/time columns, where we will use the database's built-in datetime data type. These data type mappings use ANSI standard SQL data types and thus should be transportable across RDBMS products with little effort.

Figure 65 on page 200 shows the final Order database design.

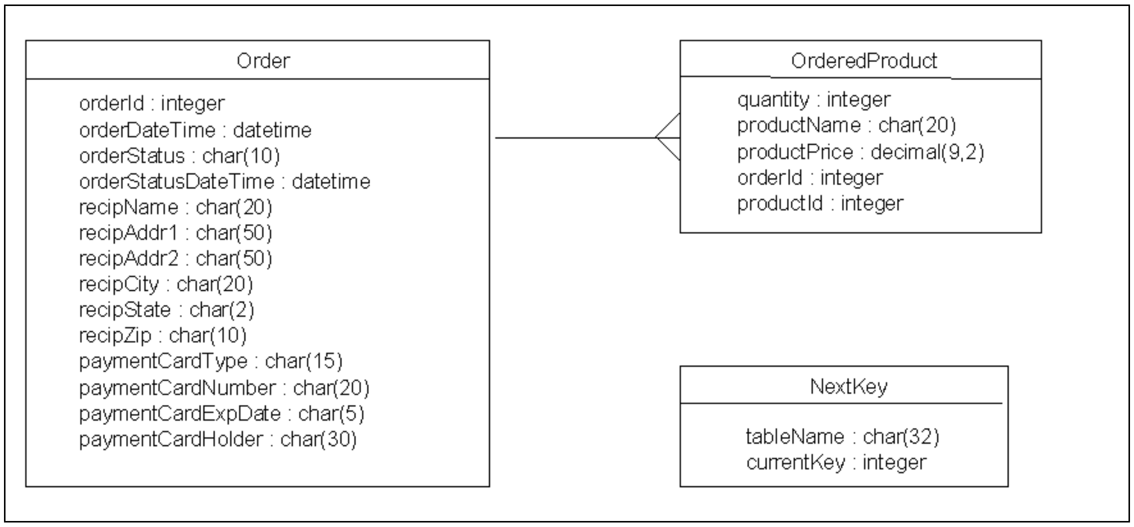


*Figure 65. Order Database Design Diagram*

## 9.6 Transaction Design

Critical to the design of distributed transaction processing systems is the handling of transactions within the system. There is a strong interaction between the architecture of the system and the details of the interactions when it comes to designing the transactions.

The first step is to identify the processing where transactions will be required. Where updates are made to more than one data source in the interaction diagrams we have to introduce a transaction to ensure the integrity of the data sources. We also have to examine the logical sequence of calls that the clients will make to the servers to see whether there are sets of client-to-server calls that must be completed as a unit of work.

An examination of the interactions in our system reveals that several of the implementations within the application servers will have to be transactional. A good example is the finalizeOrder implementation, which updates the order database, the product database, and the verification queue. You must consider all of the interactions carefully to ensure that every instance of multiple data source update is taken into account. An example of a transactional requirement that would be easy to miss is the processing of the VerificationSever. This process will need to dequeue an entry from the verification queue, read an order from the order database, and update the order's status. Remembering that the dequeue operation is an update of the verification queue, we see that this scenario must be transactional.

In our case study design none of the client requests to the servers will have to be done within a unit of work. The semantics of each request are such that each represents a complete operation that is independent of the other calls. (Note that this does not imply that there are no sequencing requirements for the calls; clearly there are such requirements in this case.) In fact, this is no accident or coincidence. It is generally considered to be good practice to limit the number of instances where a transaction has to be started within the client process. Having the client as a participant in the coordinated commit processing adds to the overhead of the commit and affects performance. In the most extreme case, poor design could cause a transaction to be held while a user views data on the screen, potentially causing a severe concurrency problem in the system. In addition client-side transactions place limits on the implementation of the client programs. For example, extending an application by introducing remote client programs that communicate with severs over slow or unreliable communication channels might be impractical if the clients have to initiate transactions. The design process we have used here has purposely avoided the need for client-side transactions. However, if you find yourself in a situation where they are required, remember that the coding techniques for client-side transactions are identical to the techniques we will use within the servers to start and end transactions.

We know of no diagramming notation specifically for recording transaction requirements, so we will simply list the instances where we have concluded that transactions will be needed. They are:

- OrderProcServer
  - finalizeOrder()
    - updates order database

        - updates product database
        - update verification queue
- VerificationSever
    - processQueueEntry()
        - updates verification queue
        - updates order database

Now we need to validate our architectural decisions in light of these transactional requirements. Taking the case study application, we find that the components affected are the RDBMS in which we implement the order database; the Product database on the host, which we are accessing through the PPC gateway; and the verification queue, which is implemented with RQS. The latter two components are certainly no problem, because PPC and RQS can participate in the transactions through the native Encina transaction mechanisms. We do have to issue one caution about PPC, however: PPC servers to bridge between the Encina transactional environment and the CICS transactional environment. The CICS system must be accessible from the computer running the PPC gateway through a SNA connection that supports the synclevel 2 protocol. Not all operating systems support a SNA connection at synchlevel 2, so you must be sure that you are using an operating system on the PPC gateway computer which does (for example, AIX). The RDBMS cannot participate in native Encina transactions. Instead it participates in Encina transactions as an external resource. Encina uses the XA resource management protocol to manage transactions with external resources. Most of the common RDBMS products support this standard, but not all of them. Our case study application will actually be developed to run against several different RDBMS products.

## 9.7 Naming Conventions

One of the things that we should always try to do when building systems is to make them as easy to understand and as maintainable as possible. A simple practice that pays big dividends over the life of a system is adherence to naming conventions. A Consistent and convenient naming of the parts of the system will make building and maintaining the system easy for the application team. It will also make the job of learning the system easy for those who come into the process later on.

The key to naming lies not in a particular standard, but in consistency and common sense. Good naming conventions should give clear, meaningful names to components of the system. The names should make clear what role the component plays in the system. But be careful not to make the names so specific that they reflect decisions that might change over time. An example from our case study might be the name we choose to give to the order database. It is likely that we will have a particular RDBMS product in mind when we do the design of the system. Perhaps we know that we will be using DB2, and so we decide to name this component of the system DB2OrderDatabase, thinking that this makes things very clear. Unfortunately, that name will become not only unclear but inappropriate should we decide at the last minute to use an Oracle RDBMS to implement the order database. The name we have been using, OrderDB, makes the component's role clear without implying implementation decisions. Notice also that we use *DB* instead of spelling out *Database*. In general, we favor using abbreviations when they are part of the common parlance, and when they are applied consistently. In the interest of consistency, we refer to our order database as OrderDB and our product database as ProductDB (note the disparity in implementation that this covers).

As a style in generating names, we are using a convention that concatenates the words that form the name with no spaces between the words. Every word after the first is capitalized. The first word may also be capitalized, depending on that type of thing it is. This is a case-sensitive naming style, which is appropriate for the environment in which we will be working.

We will use the following naming conventions:

- Encina Monitor application servers: a name reflecting the business processing that the server supports, followed by the word *Server*. The first word is capitalized, for example, OrderProcServer.
- Application server interfaces: a name reflecting the business processing that the functions in the interface implement, followed by the abbreviation *IF*, for *Interface*. The first word is capitalized, for example, OrderProcIF.
- Function names: a name describing the actions to be performed, with the first word not capitalized. Typically a phrase in the form of "verb" and "object" makes a good name, for example, `createOrder()`.
- Data item names: a name describing the data, with the first word not capitalized. If the data item is logically part of a larger structure, including the structure name or an abbreviation as the first part of the name is a good idea, for example, orderId.
- Data structure names: a name describing the group of data items being represented. The first word is capitalized, for example, Order.

## 9.8 Final Note Concerning the Design Approach

In this chapter we show how object modeling techniques can be used as part of the design process for our nonobject, multitiered case study application. Note, however, that the object modeling style we are using is somewhat different from what we would have used had we decided on an object-oriented implementation.

The most significant divergence from standard object modeling practice is the division we made between the business objects and the server-related objects. Our decision was to associate the functions exclusively with the server-related objects, and keep the business objects as simple data-only objects. This is contrary to the normal practice of distributing the functions and data more uniformly across the objects in the system. By instituting this division we are facilitating the translation of the server and interface objects into Encina interface definitions (which have only functions, not data), and the translation of the business objects into C structures (which have only data, not functions). Object purists may object to this approach, but we found it useful.

# Chapter 10. Development Phase

In this chapter we address the issues surrounding the actual coding of the application. We begin by covering topics related to the environment and project management. We proceed through the development of the application code itself.

## 10.1 Development Environment

The first step in development is to implement a special environment where the application can be developed without affecting production systems or other systems under development. To accomplish this we have to install separate instances of some or all of the infrastructure components on which our application builds.

### 10.1.1 Source Code and Version Control

It is important to establish a strategy for code management and version control from the beginning of the project. Encina development projects tend to be complex from the standpoint of managing the code modules. Some of the issues that complicate the management of code are:

- Some code is generated by the TIDL and IDL compilers each time they are run.
- Multiple external libraries must be included to support DCE, Encina, the RDBMS, PPC, RQS, and any other existing components that are used.
- Clients and servers must share certain files, for example, the header file describing the interface generated by the IDL compiler.
- Frequently common code modules are shared among servers.

Managing the dependencies among all of these modules requires discipline and planning on the part of the development team, and a strong source code and version control product. Many such products are available, but it is beyond the scope of this book to discuss them in detail. Each product has its own strengths and weaknesses, but the decision to implement rigorous code management and the determination to adhere to the practices and procedures are much more important than the particular product that is chosen.

Creating a directory structure that makes sense for your project is one of the first things to consider. In general, we recommend that you have separate directories for each server, and a separate directory structure for the common code modules. Each client program should have a separate directory as well. One approach that has proven useful is to maintain the interfaces separately from the servers, in a directory specifically for the interfaces and the files generated from them. You will also want to make provisions for including such things as useful scripts and documentation in your source code structure. Figure 66 on page 206 shows a sample code directory structure.

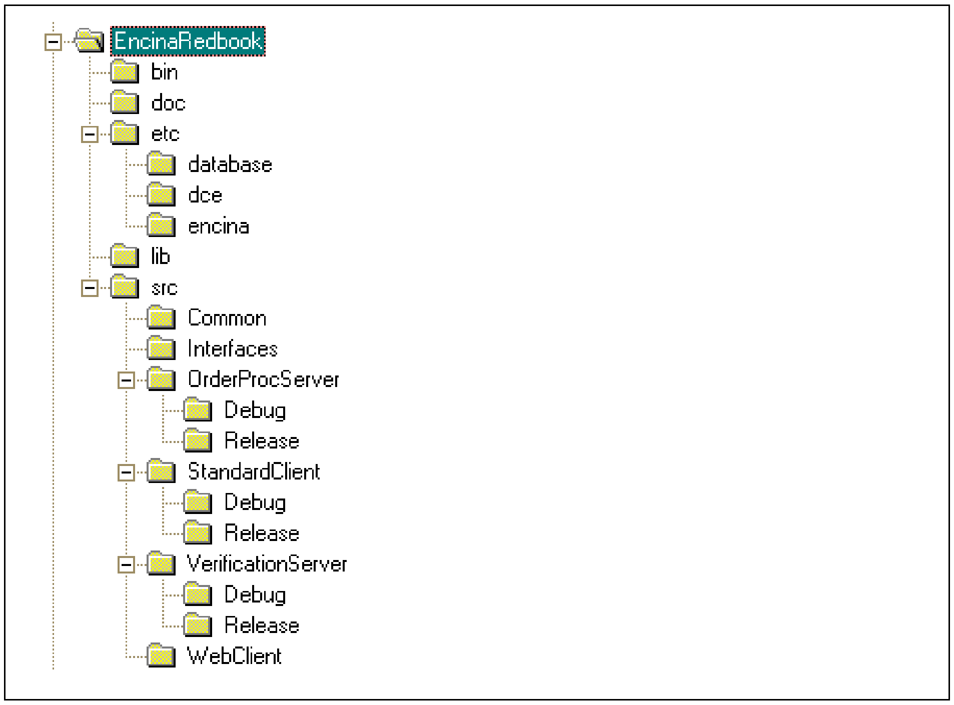


*Figure 66. Sample Source Code Directory Structure*

#### 10.1.1.1 Multiplatform Source Code Issues

Our case study environment consists of a mixture of hardware, raising the question of whether we should maintain separate source code structures on each platform or try to have a single, uniform code base across all the platforms. There are source code control products which can help synchronize source across platforms. However, our experience has been that this is difficult in practice. For the case study project we will choose to maintain a single source code base that is buildable on either of our platforms (AIX and NT). There will be some issues here, but not major ones. As we proceed through the code development we will point them out as they arise.

### 10.1.2 Build Management

On complex projects with many different components, keeping the executable components of the system synchronized as development proceeds can be difficult. It is important to adopt, and adhere to, a disciplined strategy for building the executable components of the system. Looking ahead to the coding of our fairly simple case study application, we can see that we will have to build a library of interface-related files and a library of common code modules. Both of our application servers will be dependent on these libraries, and the clients will be dependent on the interface library (at least). We can handle these dependencies in several different ways. One approach is to adopt a standard practice that whenever either of the libraries changes, we will rebuild all of the executable programs (clients and servers). This approach will force us to immediately resolve any conflicts that might have been introduced by a change in the interfaces or in the common code modules. Another approach is to use a versioning scheme as part of our code management, to maintain different versions of the libraries with different "labels." A group of developers working on an application server (for example) could defer dealing with changes to a common module by continuing to work with an earlier Version of the library. The former approach is probably satisfactory for a small- or medium-sized project, but the latter will probably be needed on large projects.

### 10.1.3 Code Partitioning

Part of the project management aspect of building distributed processing systems is the division of labor for the various components that must be developed. On any project involving more than one developer, there will need to be some rules established as to who does what. Fortunately (or unfortunately depending on your perspective), the type of system we are considering can be subdivided in many different ways.

#### 10.1.3.1 Client versus Server Division

One of the most natural dividing lines between system components is between the client tier and the application server tier. It makes a lot of sense to divide a development team along these lines. The skills required are quite different, for one thing. On the client side the focus is on presentation and ergonomics. On the server side the focus is on business logic and processing efficiency. Another motivation for a division of this sort is that the programming languages in use for the clients and for the servers often differ.

The challenges associated with this approach mostly revolve around the design of the interfaces between the clients and the servers. If the exact structure of the interface between the clients and the servers is not designed up front, there will be constant changes on both sides as new issues in the interface design surface and are resolved. It is possible to spend much more time in meetings about interface issues than in writing code if you are not careful.

#### 10.1.3.2 Application Server Division

Another very natural way to divide a team's efforts is along the boundaries between application servers. If the recommendations we have made previously have been followed, you will have designed application servers that divide along reasonable functional lines based on your business analysis. Forming separate teams to do the coding of each application server allows the developers to become expert in a particular facet of the application and of the business it supports. This is much easier than becoming expert in the entire range of the system's functionality. This dividing concept can be used by itself or in conjunction with a client versus server division. If it is used by itself, the team would be responsible for the development of the server and of the client programs that access that server. If there is also a division along the client and server lines, the team would focus exclusively on the application server.

The challenges here are related to the management of the common code components. Our case study problem exemplifies this well. Both of our application servers will access the order database, and so it makes sense for the database access routines to be part of the common code library. However, if one group discovers a shortcoming in the database access routines and makes a change, it is very likely that this will disrupt the development efforts of the team working on the other server. For this type of division to work, the design of the common code components should be considered in some detail. There will also need to be mechanisms in place to facilitate communication between the teams to handle the discovery of new common module candidates. The degree of formality in these procedures depends on the size of the project.

#### 10.1.3.3 Common Code versus Server-Specific Code Division

The final dividing line we consider is along the lines of common code modules as opposed to code that is specific to a given server. One team can focus on providing all of the common code that is required across the system, and the other teams can focus on the server-specific coding for each server. This is a less obvious dividing line than the others because the distinction between common code and server-specific code is often blurred and subject to change during the project. However, having a group focused on the reusable code does resolve some of the issues concerning communication between separate groups independently trying to create reusable components.

One of the challenges of this approach is to manage the dependencies among the development teams. The server-specific teams will need to use the common code modules as part of their development, so ideally these modules would already exist when these teams start their work. In reality it is likely that the common code module team and the server-specific teams will be doing their work in parallel. A good strategy for dealing with this is to think of it as a sort of “client/server” situation and concentrate first on detailing the interfaces to the common code modules. Once these interfaces are determined, the common code module team first provides “stub” versions of the common code that implements the interface but in a trivial fashion, giving the server-specific teams a library to use that supports the function calls in their code. As the common code module team fills in the implementations of the modules, new releases of the common library are built and made available to the server-specific teams.

### 10.1.4 Encina Infrastructure

In the development environment, we will need to implement a small, isolated Encina cell that we can control as needed for our project. Sharing a cell with other projects is certainly possible, but life will be easier for everyone if each project can have full control of its environment. We will use a simple, one-machine Encina cell for our development. The development environment will be Windows NT. For the production implementation we are targeting both NT and AIX, but the Encina interfaces are the same on both platforms, so we do not expect any issues there.

Our development Encina cell will be named /.:/encina/tricell1. The single node in the cell is on a computer named Triserv2, and thus it will be named /.:/encina/tricell1/node/Triserv2.

### 10.1.5 DCE Infrastructure

Organizations do not usually set up self-contained DCE cells for each development project. Typically there will be a development DCE cell that is shared across projects. In our experience, this usually works fine. For our application development we will make the Triserv2 machine a client in an existing DCE development cell.

### 10.1.6 Database Infrastructure

The ideal situation is to have available in development a dedicated database server of the same type that you will have in production. Sharing a database server with other development projects is usually satisfactory, as long as you can have a separate database for your project’s tables and other database entities. You will also want to have administrative authority over the database objects that your project uses.

If you do not have access to the same type of database server you will use in production, you can still make progress with development. Most of the SQL that is used to access RDBMSs is standardized. The commands used to create the database structures may be significantly different, but this does not affect the application code. Beginning development with one type of database and then shifting to the actual target database product can be done without too much trouble. You do need to allow for time in the project to thoroughly retest all of the database code against the new RDBMS after the switch is made. One of the advantages of embedded SQL is to provide some insulation from the RDBMS to facilitate product changes.

Our development effort for the case study application began before a copy of DB2 was available for our use. We initially coded the database portions of the project against a Sybase SQL Anywhere database running on Windows NT and then migrated to the DB2 database near the end of the development phase. The SQL Anywhere product does not support XA connections, so this was a less than ideal situation. We coded each of the database access routines in embedded SQL with a connect and disconnect command within each function. When we made the switch to DB2, we removed the connect and disconnect commands, added the Encina Monitor calls to initialize the database as an XA resource, switched the build process to use the DB2 preprocessor instead of Sybase’s, and completed the conversion, with little pain.

## 10.2 Application Development

In this section we proceed through the development of the application code. The sequence of topics matches the considerations above concerning code division and structure. Specifically, we use a combination of all three of the code division strategies described above. We treat the development of the client programs and server programs separately, develop each server independently, and focus on the common code modules we have identified.

### 10.2.1 Interface Coding

Regardless of the decisions you have made about how to divide the development effort, it is good practice to start your development effort by creating the interfaces your servers will provide. Our application architecture has two interfaces defined, OrderProcIF and VerificationIF. We determined which responsibilities each server would have in terms of use cases and devised a set of functions required in the interface to support those responsibilities. This set of functions was the result of the function mapping with interaction diagrams described in Chapter 9, “Design Phase” on page 191.

Before we can actually code the interface definition as TIDL, we need to examine the details of the interface and the data that is passing between the clients and the servers. We will look for opportunities to define data structures that will facilitate manipulation of the function parameters and return values easier and create as many reusable definitions as possible.

#### 10.2.1.1 Code File Structure

The code structure we will use for our interface definition code will consist of a common file containing the structure definitions and separate files defining each of the interfaces. The common file will be in IDL syntax and will be called OrderProcCommon.idl. The interface files will, of course, be in TIDL syntax and will be called OrderProcIF.tidl and VerificationIF.tidl.

#### 10.2.1.2 RPC Return Structure

Our first decision will be to describe the return value from the functions in our interfaces. It is usually a good idea to have a standardized return from RPCs so that client programs can have generic processing routines for handling the return values from the server. The strategy we will use here is to define a data structure that contains result codes and error messages and have this structure be the return value from all of our interface functions. Because we have various components in the back end of the system that might generate errors, we want to be able to return both a general return that signifies the success or failure of the requested operation and a more specific return that can contain component-specific error information. Figure 67 on page 211 shows the portion of the OrderProcCommon.idl file that defines this structure.

```
/* standard return structure */
typedef struct {
long retCode;
[string] char errorSystem[11];
[string] char errorCode[81];
} RpcReturn;

typedef [ptr] RpcReturn* RpcReturnPtr
```

*Figure 67. IDL Definition of the RpcReturn Structure*

Notice that the data fields that are character arrays are intended to be treated as null-terminated strings, and DCE is informed of this fact with the [*string*] qualifier. An additional byte is allocated to allow room for the null character at the end.

#### 10.2.1.3 Product Data Structures

The results of our analysis provide us with a set of functions and the data groupings that the functions use as parameters. We will need to define a structure in the OrderProcCommon.idl file for each of the structures that is used in a function within either of the interfaces. In addition we will include a type definition for a pointer to each of the structures.

One group of data used in several ways as a parameter is the product data. We have the need to pass a single instance of product data and to pass a variable size list of product structures. On the basis of the information gathered about this data, we develop the IDL definitions and include them in the OrderProcCommon.idl file (see Figure 68 on page 212). Notice the form of the definition for the variable length list, ProductList. The structure consists of a long integer field to hold the number of occurrences and a special IDL definition for an array of ProductInfo structures.

```
/* product information from the product database */
typedef struct {
long productId;
[string] char productName[21];
[string] char productDesc[201];
float productPrice;
long productInventory;
} ProductInfo;
typedef struct {
long numProds;
[size_is(numProds)] ProductInfo products[*];
} ProductList;
```

*Figure 68. IDL Definitions for the Product Data Structures*

#### 10.2.1.4 Order Data Structures

The order data group is a bit more complicated than the product data group. The order consists of a set of data about the order itself and a list of ordered products associated with the order. We approach the definition of the structures by first creating a structure called BasicOrder, which describes the data items concerning the order as a whole, basically the data items in the CustOrder database table defined in the data modeling phase. We then define a structure for a single instance of an ordered product, basically the data from the OrderedProduct table without the foreign key to the CustOrder table, orderId. In addition we define a structure, OrderInfo, which consists of the fields from the BasicOrder structure and a variable length list of OrderedProduct structures. The form of the variable length list definition here is the same as when it was used by itself with the ProductList structure. Another formulation of the OrderInfo structure would be to use the BasicOrder structure as a component of the OrderInfo structure, instead of duplicating the field definitions. In fact, this would be the preferred approach except for the fact that the current version of the DE-Light Gateway does not handle nested structures as RPC parameters. Because we will access this interface from a standard client and from a Web client, using the DE-Light Gateway, we will use a formulation of the OrderItem that is compatible with the Gateway. Finally we define pointer types for each of the structures. Figure 69 on page 214 to Figure 70 on page 215 shows the IDL syntax of the OrderProcCommon.idl file for the order data structures.

```
/* basic order information from the order database */
typedef struct {
long orderId;
[string] char orderDateTime[27];
[string] char orderStatus[11];
[string] char orderStatusDateTime[27];
[string] char recipName[21];
[string] char recipAddr1[51];
[string] char recipAddr2[51];
[string] char recipCity[16];
[string] char recipState[3];
[string] char recipZip[11];
[string] char paymentCardType[21];
[string] char paymentCardNumber[21];
[string] char paymentCardExpDate[11];
[string] char paymentCardHolder[21];
} BasicOrder;

/* ordered product data from the order database */
typedef struct {
long productId;
[string] char productName[21];
float productPrice;
long orderedQty;
} OrderedProduct;

/* variable length list structure of ordered products */
typedef struct {
long orderId;
[string] char orderDateTime[27];
[string] char orderStatus[11];
[string] char orderStatusDateTime[27];
[string] char recipName[21];
[string] char recipAddr1[51];
[string] char recipAddr2[51];
[string] char recipCity[16];
[string] char recipState[3];
[string] char recipZip[11];
[string] char paymentCardType[21];
[string] char paymentCardNumber[21];
```

*Figure 69. (Part 1 of 2) IDL Definitions for the Order Data Structures*

```
/* variable length list structure of basic orders */
typedef struct {
long numOrders;
[size_is(numOrders)] BasicOrder orders[*];
} OrderList;

typedef [ptr] OrderedProduct* OrderedProductPtr;
typedef [ptr] BasicOrder* BasicOrderPtr;
typedef [ptr] OrderInfo* OrderInfoPtr;
```

*Figure 70. (Part 2 of 2) IDL Definitions for the Order Data Structures*

#### 10.2.1.5 Verification Statistics Data Structure

The structure of the verification statistics is provided from the verification server's interface. It is a simple structure with no lists, and so the definition is straightforward. Figure 71 on page 215 shows the IDL definitions for this structure.

```
/* verification statistics data */
typedef struct {
long numOrdersReviewed;
long numOrdersApproved;
long numOrdersFailWarning;
long numOrdersFailFatal;
} VerifyStats;
```

*Figure 71. IDL Definitions for the Verification Statistics Data Structure*

#### 10.2.1.6 Complete OrderProcCommon.idl File

All of the data structure definitions are coded within the scope of an interface definition called *Common*. This name is arbitrary, because it is not ever referenced directly. However, we are defining an interface, so we are required to provide a UUID and version information as with any interface definition. We used the uuidgen utility to produce the UUID for inclusion in the file. Figure 72 on page 216 to Figure 73 on page 217 shows the complete OrderProcCommon.idl file.

```
[ uuid(00387520-78f9-14cf-ab42-00a024c008a6), version(1.0) ]
interface Common
{
/*typedef [ptr,string] char* charPtr;*/
typedef struct {
long retCode;
[string] char errorSystem[11];
[string] char errorCode[81];
} RpcReturn;
typedef struct {
long productId;
[string] char productName[21];
[string] char productDesc[201];
float productPrice;
long productInventory;
} ProductInfo;
typedef struct {
long numProds;
[size_is(numProds)] ProductInfo products[*];
} ProductList;
typedef struct {
long productId;
[string] char productName[21];
float productPrice;
long orderedQty;
} OrderedProduct;
typedef struct {
long orderId;
[string] char orderDateTime[27];
[string] char orderStatus[11];
[string] char orderStatusDateTime[27];
[string] char recipName[21];
[string] char recipAddr1[51];
[string] char recipAddr2[51];
[string] char recipCity[16];
[string] char recipState[3];
[string] char recipZip[11];
```

*Figure 72. (Part 1 of 2) The OrderProcCommon.idl File*

```
typedef struct {
long orderId;
[string] char orderDateTime[27];
[string] char orderStatus[11];
[string] char orderStatusDateTime[27];
[string] char recipName[21];
[string] char recipAddr1[51];
[string] char recipAddr2[51];
[string] char recipCity[16];
[string] char recipState[3];
[string] char recipZip[11];
[string] char paymentCardType[21];
[string] char paymentCardNumber[21];
[string] char paymentCardExpDate[11];
[string] char paymentCardHolder[21];
long numProds;
[size_is(numProds)] OrderedProduct item[*];
} OrderInfo;
typedef struct {
long numOrders;
[size_is(numOrders)] BasicOrder orders[*];
} OrderList;
typedef struct {
long numOrdersReviewed;
long numOrdersApproved;
long numOrdersFailWarning;
long numOrdersFailFatal;
} VerifyStats;
typedef [ptr] RpcReturn* RpcReturnPtr;
typedef [ptr] ProductInfo* ProductInfoPtr;
typedef [ptr] ProductList* ProductListPtr;
typedef [ptr] OrderedProduct* OrderedProductPtr;
typedef [ptr] BasicOrder* BasicOrderPtr;
```

*Figure 73. (Part 2 of 2) The OrderProcCommon.idl File*

#### 10.2.1.7 The OrderProcIF Interface

The task of coding the interface definitions in TIDL is fairly simple. We define an interface and list the functions that are provided as part of the interface. The only significant decision that has not been addressed is how we will deal with the data being returned from the functions as output parameters. The issue essentially is one of memory management, and the fundamental question is, “Who will allocate the space for a return parameter and who will free that space?” One strategy is to have the client allocate the space for the parameter and specify the parameter as a pointer to the memory allocation for the parameter. This strategy works fine as long as the client knows how much space to allocate before making the function call. In the case of a variable length list (of which we will have several), we need to use a mechanism that allows the server to allocate the space and the client to receive an arbitrary-sized return and then free the space after use. This mechanism calls for the parameter to be specified as a pointer to one of the pointer data types defined in our OrderProcCommon.idl file. For example, we would specify ProductListPtr* as the type of the parameter for the variable length list of product structures. The server will allocate the appropriate space according to the size of the list, include the list length in the long integer field defined in the ProductList structure, and let DCE marshall the return to the client. The server uses a special memory allocation call, rpc_ss_allocate(). The client uses double indirection to refer to the returned memory and a special deallocation call, rpc_ss_client_free(), to release the space. As you can see from the TIDL syntax for OrderProcIF in Figure 74 on page 219, we use this mechanism for all returns involving structures, and we use the simpler mechanism for returns that are not structures.

Other aspects of the TIDL definitions that are worth noting are the handling of the char * input parameter to the `orderItem` function, the fact that we have chosen to pass the productPrice as a long (also in the `orderItem` function), and the inclusion of the `pingOrderProcIF` function. Passing parameters that you want to be treated as character strings requires that you inform the IDL compiler about this special treatment by defining the parameter as [in,string] as we have done here. If the parameter is simply labeled as [in], the interface will compile and build with no problems, but the character string your client passes in the call will not arrive as a usable character string at the server, a subtle problem that is difficult to track down. Our choice of a long data type for the price, which is defined as a floating point number elsewhere, was based on caution alone. The handling of floating point data tends to vary on different platforms, and to make the interface definition portable across our platforms, we chose to avoid any potential problems and convert our prices to the long data type before passing them as a parameter. In fact, when we originally coded the interface with float as the data type, the IDL compiler generated a warning to the effect that special compile flags were required to properly compile the source code it produced. Finally, the `pingOrderProcIF` function is an example of a simple "hello, server" function that we regularly include in every interface as a way of initially testing the communication between the client and server. This function is implemented in the same way as all the others, except that when we get to the RPC implementation function in the server, we simply send back a valid RpcReturn structure with no additional processing (except logging to the server output file). The function is typically left in the interface definition to avoid changing the interface during development or the move to production.

```
[ uuid(000b4aa0-7911-14cf-845a-00a024c008a6), version(1.0) ]
interface OrderProcIF {
import "tpm/mon_handle.idl";
import "OrderProcCommon.idl";
/* is the server alive (ready to receive RPCs) */
[nontransactional] RpcReturn pingOrderProcIF();
/* create an order */
[nontransactional] RpcReturn createOrder(
[out] long* orderId);
/* list all the products that can be ordered */
[nontransactional] RpcReturn listProducts(
[out] ProductListPtr* prodList);
/* add quantity of a product to order */
[nontransactional] RpcReturn orderItem(
[in]  long orderId,
[in]  long productId,
[in,string]  char * productName,
[in]  long orderedQty,
[in]  float productPrice);
/* review order using orderId, returns the original order,
   and up-to-date status */
[nontransactional] RpcReturn reviewOrder(
[in]  long orderId,
[out] OrderInfoPtr* ordInfo);
/* view all orders */
[nontransactional] RpcReturn viewOrders(
[out] OrderListPtr* ordList);
```

*Figure 74. TIDL Definition for the OrderProcIF Interface (OrderProcIF.tidl)*

#### 10.2.1.8 The VerificationIF Interface

The VerificationIF interface is much simpler than the OrderProcIF interface but uses the same concepts for parameter passing. Figure 75 on page 220 shows the complete code for the VerificationIF interface.

```
 [uuid(0035b600-ba01-14d8-b5ad-00609735bb67),version(1.0)]
 interface VerificationIF {
 import "tpm/mon_handle.idl";
 import "OrderProcCommon.idl";
 [nontransactional] RpcReturn pingVerificationIF();
 [nontransactional] RpcReturn showStats(
[out] VerifyStatsPtr * statsPtr);
 [nontransactional] RpcReturn printReviewList();
```

*Figure 75. TIDL Definitions for the VerificationIF Interface (VerificationIF.tidl)*

***Client Binding to the Interfaces***

The TIDL files describe the interface, and another set of files, the TACF files, describe how clients will access the interface. There are basically two methods for client binding, explicit and implicit. By far the most commonly used method for this type of application is implicit binding. Implicit binding allows Encina to do most of the work for us by automatically finding an available server with the right interface and performing load balancing across the servers. We will use implicit binding in our case study application, to simplify the syntax for the TACF files. We define the interface and specify that an implicit RPC handle will be added automatically to the call. The two TACF files we need are identical except for the name of the interface, but there must be a TACF file for each of the TIDL files. The file names are the same as those of the TIDL file except for the extension, which is .tacf for the TACF files. The TIDL compiler knows to look for a file of this name, so it is not referenced directly during the compile processing. Figure 76 on page 220 shows the TACF syntax for our OrderProcIF interface.

```
[implicit_handle (mon_handle_t handle)]
interface OrderProcIF
{
}
```

*Figure 76. TACF Definition for the OrderProcIF Interface (OrderProcIF.tacf)*

### 10.2.1.9 Compiling the Interface Definition Files

The process of converting the TIDL and TACF syntax in the source code components consists of two steps, a TIDL compile and an IDL compile, for each interface. The command used to do the TIDL compile on the OrderProcIF.tidl file is:

```
tidl -Iencina_include_path -I. OrderProcIF.tidl
```

We show the command line as it would appear as part of a make file. The -I directives are used to ensure that the compiler can find the files we are importing, namely, the OrderProcCommon.idl file and the tpm/mon_handle.idl file. The former is in the same directory as the .tidl file, and the latter is found in the Encina include path, typically /opt/encina/include. A successful compile will produce these files:

- _OrderProcIF.idl
- OrderProcIF_client.c
- OrderProcIF_cswtch.c
- OrderProcIF_manager.c

The _OrderProcIF.idl file becomes input to the next step, the IDL compile. The OrderProcIF_client.c and OrderProcIF_cswtch.c files become part of the client programs. The OrderProcIF_manager.c file becomes part of the server program. Both of our TIDL files are processed in this way.

The command for the IDL compile is:

```
idl -no_mepv -keep c_source -cepv -Iencina_include_path \
-Idce_include_path -I. _OrderProcIF.idl
```

Notice that we include -I directives for both Encina and DCE include paths as well as our local directory. These are not always required; their use depends on what you include in your definitions. In addition, if your code management scheme has placed files you reference with import statements in other directories, you will need to include -I directives that point to these directories. We will perform the IDL compile on the _OrderProcIF.idl and _VerificationIF.idl files that result from our TIDL compiles, as well as on our OrderProcCommon.idl file. From our _OrderProcIF file, the IDL compile will produce these files:

- _OrderProcIF.h
- _OrderProcIF_cstub.c
- _OrderProcIF_sstub.c

The .h file is referenced in both the client and the server. The _cstub.c file is used in the client program, and the _sstub.c file is used in the server program. The IDL compile of the OrderProcCommon.idl file will produce only OrderProcCommon.h, because there are no functions defined in that file that would require the client and server stub files.

For our application code management scheme, we chose to keep the interface-related files separate from the server and client code. Thus the final step in the interface building process is to create a library from the .c files that can be linked with the client and server programs. We chose to build all of the files from both interfaces into a single library. This is a reasonable choice for a small project like this, but on a larger project you will want to create separate libraries for each interface, and possibly separate client and server libraries for each interface.

The standard C compiler on your target platform is used to compile the source files. The compiler switches you need to use are different for each environment, so make sure you check Appendix A of the *Writing Encina Applications* manual (SC33-1760-01) for the correct compiler switches. Here we only do a compile to create object files to include in a library, so there are no linkage issues, yet. Use the appropriate tool for your platform to create the library. For our code management scheme, we specify the output library file to be placed in the /lib directory of our source code structure.

## 10.2.2 Common Module Construction

In Section 9.4, “Common Application Components and Standards” on page 197 we describe several areas where you could consider creating common code modules that are reused across servers, and perhaps clients. In this section we describe the code modules that we chose to make part of the common code base for our case study application.

### 10.2.2.1 RPC Return Processing

We chose to use a common structure in our TIDL and IDL definitions to use as the return from each of our RPCs. Thus we will have to manipulate this structure quite a bit within both our servers and our clients. We will create a common code module named RpcReturn, which will include functions to initialize and display this structure. In addition we will define several symbols for use with the numeric return code in the structure, to enable you to use meaningful names instead of having to remember numbers when setting return values. The RpcReturn.h file contains these definitions and the function prototypes (see Figure 77 on page 223).

```
#include "OrderProcCommon.h"

/* error return code values */
#define GOODRETURN 0
#define WARNINGRETURN 1
#define FATALRETURN 2

void initRpcReturn(RpcReturn *ret);
void setRpcReturn(RpcReturn *ret, int rc, const char *es, const char
*ec);
```

*Figure 77. The RpcReturn Header File*

The RpcReturn.c file contains the function implementations (see Figure 78 on page 224).

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "RpcReturn.h"

void initRpcReturn(RpcReturn * ret) {
ret->retCode = GOODRETURN;
strcpy(ret->errorSystem, "");
strcpy(ret->errorCode, "");
}

void setRpcReturn(RpcReturn *ret, int rc, const char *es, const char
*ec) {
ret->retCode = rc;
strcpy(ret->errorSystem, es);
strcpy(ret->errorCode, ec);
}

char *formatRpcReturn( RpcReturn ret ) {
char *fmtStr;
fmtStr = (char *)malloc(150);
fmtStr = "Return Info: [";
if (ret.retCode == 0) {
strcat(fmtStr,"Good Return");
}
else if ( ret.retCode == 1 ) {
strcat(fmtStr, "Warning Return");
}
else if ( ret.retCode == 2 ) {
strcat(fmtStr, "Fatal Return");
}
strcat(fmtStr,"] [");
strcat(fmtStr,ret.errorSystem);
```

*Figure 78. RpcReturn Implementation File*

### 10.2.2.2 Message Logging

Our application servers will each have an output file for messages. Encina will use the file for severe messages generated within the server, but we can use it for our own messages as well. A good mechanism for logging messages is important for debugging during development and for monitoring the server during production. The problem with generating messages to the log for debugging the server is that you do not want to have these same messages in the log when you run in production. However, you certainly do not want to make changes to your code (commenting out the logging code, for example) between development and production. The solution we use is to provide a function that writes messages to the log and allows the messages to be categorized as informational, debugging, warnings, or errors. We will also implement a configuration variable that can be set in the environment or on the command line that specifies the type of messages to be displayed. In way we can include all of the debugging messages that we want and simply disable them with the configuration variable when we do not want them to be written out. This logging function could be used within the clients as well, because the statements are actually written to the stdout stream (which is directed to the server output file in the case of the application server). The logging common module is implemented in the LogEntry.h and LogEntry.c files. Figure 79 on page 225 shows the `LogEntry` function

```
void LogEntry(int indentLevel, int msgType, char* msgText, ...) {
va_list argList;
char fmtStr[256];
char timeStr[30];
char msgTypeStr[10], msgIndentStr[50];
time_t currTime = time((time_t *) 0);
struct tm * tms = localtime(&currTime);
int i = 0;
if ((g_sconf.msgLevelMask & msgType) == msgType) {
switch( msgType ) {
case INFOMSG:    { strcpy(msgTypeStr, "INFO   "); break; }
case ERRORMSG:   { strcpy(msgTypeStr, "ERROR**"); break; }
case WARNINGMSG: { strcpy(msgTypeStr, "WARN   "); break; }
case DEBUGMSG:   { strcpy(msgTypeStr, "DEBUG  "); break; }
case ALWAYS:     { strcpy(msgTypeStr, "------>"); break; }
}
strcpy(msgIndentStr, "");
for (i = 1; i < indentLevel; i++) {
strcat(msgIndentStr, "   ");
}
strncpy(timeStr,asctime(tms),24); /* strip off newline and null */
timeStr[24] = '\0';
sprintf(fmtStr,"%s :%s: %s%s\n",
timeStr, msgTypeStr, msgIndentStr, msgText);
va_start(argList, msgText);
vprintf(fmtStr, argList);
va_end(argList);
```

*Figure 79. Implementation File for LogEntry*

#### 10.2.2.3 Standardized Error Checking

All of the functions that we will call from the Encina libraries return result codes that describe the results of the processing. These return values are of the unsigned long data type and are typed as enumerations. For the standard Encina functions, such as the Encina Monitor functions, and the RQS functions, there is a common structure to these enumerations. The good results all have a value of 0, although the enumeration symbol is different (for example, MON_SUCCESS for the monitor calls, RQS_SUCCESS for the RQS calls). The enumeration symbols for bad returns are numbered sequentially from a base number that is assigned to the component. The result of this scheme is that the actual number that is returned from a function call is difficult to interpret unless it is a good return. In the case of PPC processing, the Encina PPC return values follow the scheme described here, but the CPI-C functions have enumerations that do not use the base number offset scheme.

Fortunately, the Encina libraries provide a set of function calls that can be used to translate the Encina errors from their numeric values into symbols or descriptive strings. Unfortunately, these calls do not work with the CPI-C return values.

To facilitate the processing of returns from these calls, we will develop some standard error checking functions based on the translation functions provided by Encina. These functions are contained in the StatusStrings common module. We will implement a function for processing standard Encina call results, and another specifically to handle the CPI-C call returns (see Figure 80 on page 227 to Figure 81 on page 228).

```
int showStatus( char *fxn, unsigned long status ) {
char *sym, *str;
unsigned long rc;

if ( status == 0 ) {
LogEntry(2,DEBUGMSG,"%s processed successfully.",fxn);
return TRUE;
}
else {
sym = malloc(ENCINA_MAX_STATUS_STRING_SIZE + 1);
str = malloc(ENCINA_MAX_STATUS_STRING_SIZE + 1);

rc = encina_StatusToSymbol( status,
ENCINA_MAX_STATUS_STRING_SIZE, sym );
if ( rc == ENCINA_STS_BUFFER_TOO_SMALL )
strcpy(sym,"*** Translation failed - buffer too small.");
if ( rc == ENCINA_STS_CATOPEN_FAILED )
strcpy(sym,"*** Translation failed - cat open failed.");
if ( rc == ENCINA_STS_UNKNOWN_ERROR )
strcpy(sym,"*** Translation failed - unknown failure.");
if ( rc == ENCINA_STS_UNKNOWN_FACILITY )
strcpy(sym,"*** Translation failed - unknown facility.");

rc = encina_StatusToString( status, ENCINA_MAX_STATUS_STRING_SIZE, str
);
if ( rc == ENCINA_STS_BUFFER_TOO_SMALL )
strcpy(str,"*** Translation failed - buffer too small.");
if ( rc == ENCINA_STS_CATOPEN_FAILED )
strcpy(str,"*** Translation failed - cat open failed.");
if ( rc == ENCINA_STS_UNKNOWN_ERROR )
strcpy(str,"*** Translation failed - unknown failure.");
if ( rc == ENCINA_STS_UNKNOWN_FACILITY )
strcpy(str,"*** Translation failed - unknown facility.");

LogEntry(2,ERRORMSG,"%s FAILED with [%d]%s!",fxn,status,sym);
LogEntry(2,ERRORMSG,"==>%s",str);
```

*Figure 80. (Part 1 of 2 ) Error Processing Functions (StatusStrings.c)*

```
int showCpicStatus( char *fxn, unsigned long status ) {
ppc_status_t pst;

if ( status == 0 ) {
LogEntry(2,DEBUGMSG,"%s processed successfully.",fxn);
return TRUE;
}
else {
LogEntry(2,ERRORMSG,"%s FAILED with CM value=[%d]!",fxn,status);
if ( status == CM_PRODUCT_SPECIFIC_ERROR ) {
pst = cpic_GetProductSpecificDetail();
showStatus("Product specific details",pst);
}
return FALSE;
```

*Figure 81. (Part 2 of 2) Error Processing Functions (StatusStrings.c)*

#### 10.2.2.4 Server Configuration

Our application servers will be fairly complex. They will need to access a RDBMS, use RQS queues, access the PPC gateway, and perform logging in addition to the regular Encina server processing. All of these activities require control information. For example, to access the database, we need the database name, a user id, and a password. For RQS we need to know the RQS server name and the names of the queues and queue sets we will access. For PPC gateway access we need to know such things as our partner logical unit name. In most cases, we could argue that it would not be the best solution to hard code the values into the programs. We could use #define statements in our code to give symbolic names that could easily be changed in one place, but this would still require a recompile of the program to effect the changes. The other option is to use environment variables or command line arguments. An application server can be configured with either of these techniques through the enconsole tool or enccp without requiring any code changes or recompiles.

We will adopt a standardized way of configuring our servers through the use of environment variables and command line arguments. First we have to define a data structure that can be used globally to contain all of this control information. Because we want to use the same structure for both of our servers (and any potential future servers), we will assess the needs of each server and create a structure that contains all of the necessary configuration information. There is considerable commonality in the configuration requirements, but sometimes only one server has need for a particular configuration variable.

We will implement server configuration strategy by creating a shared global data structure containing all of the identified variables and including the header file defining this structure with each of our server's implementation files. Because we want to get values from the environment or the command line for each variable, we will implement a common function that checks for an environment variable and a command line argument that corresponds to the configuration variable. We will implement this function so that the caller can specify which value, environment or command line, should be preferred if both are present, and we will return an empty string if neither is available. For each of the identified environment variables, we will implement a function that retrieves the value for that variable. These functions will also access the environment and the command line.

Figure 82 on page 229 shows the common structure we will use and several enumerations that are used in the implementations of the functions that manipulate the structure. The figure is an extract of the ServerConfig.h file.

```
typedef struct {
int msgLevelMask;
int retrieveEnabled;
char schedulePolicy[22];
char dbUserId[33];
char dbPassword[33];
char dbName[33];
char ppcSideInfoName[81];
char ppcLuPartner[21];
char rqsName[81];
char pendingQueueName[81];
char pendingQueueSet[81];
char reviewQueueName[81];
char reviewQueueSet[81];
int verThrdDelay;
} ServerConfig;

extern ServerConfig g_sconf;

enum VarPrefEnum { PREFER_ENV, PREFER_CMD };
```

*Figure 82. ServerConfig Structure and Associated Enumerations*

The function that actually gets the values from the environment and command line is called getVariableValue (see Figure 83 on page 231). This function takes parameters that give the command line switch, the environment variable name, the preferred source for the value, and the command line argument count and char array. The output parameters are an enum value describing the source of the returned value, and the value itself as a string.

```
void getVariableValue(char *cmdSwitch,
char *envVar,
enum VarPrefEnum preference,
int argc,
char *argv[],
enum VarSource *varSrc,
char *value)
{
char cmdVal[80], envVal[80];
int i;
strcpy(cmdVal,"");
for ( i=0 ; i < argc ; i++ ) {
if ( strcmp(argv[i],cmdSwitch) == 0 ) {
strcpy(cmdVal,argv[i+1]);
break;
}
}
if ( getenv( envVar ) )
strcpy(envVal,getenv( envVar ));
else
strcpy(envVal,"");
if ( preference == PREFER_ENV )
if ( strcmp(envVal,"") != 0 ) {
*varSrc = ENV;
strcpy(value,envVal);
}
else
if ( strcmp(cmdVal,"") != 0 ) {
*varSrc = CMD;
strcpy(value,cmdVal);
}
else {
*varSrc = NONE;
strcpy(value,"");
}
if ( preference == PREFER_CMD)
if ( strcmp(cmdVal,"") != 0 ) {
*varSrc = CMD;
strcpy(value,cmdVal);
}
else
if ( strcmp(envVal,"") != 0 ) {
*varSrc = ENV;
strcpy(value,envVal);
```

*Figure 83. Accessing the Environment Variables and Command Line Arguments*

By using `getVariableValue`, we can then implement specific functions to retrieve each of our configuration variables. Because the `getVariableValue` function needs to know the command line switch to use and the environment variable name, we have to standardize these items now. The code in Figure 84 on page 232 shows one of the configuration variable retrieval functions, setEnvSchedulePolicy.

```
/* set server scheduling policy - Defaults to MON_CONCURRENT_SHARED */
void setEnvSchedulePolicy( int argc, char **argv, enum VarPrefEnum pref
) {
char varValString[100];
enum VarSource varSrc;

getVariableValue("-sched","SCHEDULE_POLICY",pref,argc,argv,&varSrc,varV
alString);
LogEntry(2,INFOMSG,"Schedule policy string: [%s]. Source prefered:
[%s]. Actual source: [%s]",
varValString, getVarPrefString(pref), getVarSourceString(varSrc));
if ( strcmp(varValString,"") == 0 ) {
strcpy(g_sconf.schedulePolicy,"MON_CONCURRENT_SHARED");
LogEntry(3,WARNINGMSG,"Defaulting to MON_CONCURRENT_SHARED");
}
else {
strcpy(g_sconf.schedulePolicy,varValString);
LogEntry(3,WARNINGMSG,"Schedule policy set to [%s]",
g_sconf.schedulePolicy);
}
```

*Figure 84. Sample Configuration Variable Set Function*

We will use the functions associated with the configuration variables we need to set for each server within the implementation files for the server. We cover the structure of these files in detail later in this chapter; we present a section of the setup routine for the OrderProcServer (Figure 85 on page 233) now to complete the picture of the server configuration common code module.

```
.
.
.
ServerConfig g_sconf;/* global server configuration structure*/
.
.
.
/*********************************************************************
**/
/*             ORDERPROC SERVER MAIN FUNCTION
*/
/*********************************************************************
**/
int main(int argc, char* argv[]) {
LogEntry(1, ALWAYS, "OrderProcServer Start-Up Beginning...\n");
if ( !EstablishEnv(argc, argv) ) return 1;
.
.
.
/*********************************************************************
**/
/*               ORDERPROC SERVER FUNCTION IMPLEMENTATIONS */
/*********************************************************************
**/
int EstablishEnv( int argc, char **argv ){
/* merge the command line arguements and the
   environment variables to establish the working
   environment for the server */

LogEntry(1,ALWAYS, "Begining server environment set-up.");

/* write the possible config variabes to the log */
LogConfigOptions();

/* call the set up functions for the needed variables */
setEnvMsgLevel(argc,argv,PREFER_CMD);
setEnvRetriveEnable(argc,argv,PREFER_CMD);
setEnvSchedulePolicy(argc,argv,PREFER_CMD);
```

*Figure 85. Using the Configuration Common Module during Server Startup*

### 10.2.3 Database Processing

Both of our application servers will access the order database. In our analysis and design efforts we considered the database as an object with certain responsibilities. This section explains how that design approach pays off. As a result of our interaction diagramming, we know all of the functions required of the order database

#### 10.2.3.1 Order Database Functions

We will implement a common code module containing all of the functions that access the order database. This approach will help us in several ways. First, it creates a reusable module that can be accessed from both servers, and second, it facilitates the code build processing. Our build processing will involve special preprocessing of the database code before the actual C compile. By isolating this code into a single module, we prevent having to run the database preprocessing over any other code. The functions implemented as part of this common module are listed in the header file for the module, OrderDbDB2.h, shown in Figure 86 on page 234.

```
void getRmName( char *rmName );
int orderDbInit();
int orderDbClose();
int createOrderKey(long* orderId, RpcReturn* retValue);
int addToOrder(long orderId, long productId, unsigned char *
productName, long orderedQty, long productPrice, RpcReturn* retValue);
int getOrderInfo(long orderId, OrderInfoPtr* ordInfo, RpcReturn*
retValue);
int getOrderList(OrderListPtr* ordList, RpcReturn* retValue);
int finalizeCustOrder(BasicOrder* basicOrder, RpcReturn* retValue);
```

*Figure 86. Order Database Function Prototypes (OrderDbDB2.h)*

#### 10.2.3.2 Database Coding Strategy

We will use embedded SQL to access the database. The other option is to work directly with the DBMS's call level interface (CLI). The issues are not necessarily easy to sort out in deciding which option to use. Embedded SQL is much easier to write, is fairly standard across DBMS products, and provides most of the capabilities of the CLI. However, it requires extra preprocessing, and the resulting code may not be as efficient in all cases as CLI code written by an experienced database programmer. Writing CLI database access code provides full access to the capabilities of the DBMS and can be quite efficient. It is also possible to write generic routines in CLI for the common tasks that must be performed. However, CLI programming is more difficult and it tends to be very DBMS specific. For our purposes, we will use embedded SQL for the following reasons: The scope and complexity of our database access needs are limited, and we have only one database to access. In addition, we want to remain as flexible as possible in our choice of

DBMS products, so the standardization aspects of embedded SQL are important.

The contents of the order database common module are shown in their entirety in the code listings. We will select one representative function here and walk through the details. Our initial DBMS selection is the IBM DB2 Universal Database Version 5 from IBM, and the order database common module code was written to work with this DB2. As stated earlier, though, embedded SQL is similar across products.

#### 10.2.3.3 XA Database Access

First we need to address the issue of accessing the database as an XA resource. DBMS vendors must provide support for this standard and can do so in various ways. In general, the DBMS must provide a data structure known as an *XA switch* and define the structure of an *open string*, a *close string*, and a *serialization string*. The XA switch structure is used for communication between the application server and the database and is provided by the DBMS vendor within a header file that we will either include specifically or which is included automatically during preprocessing. For DB2, the includes are generated for us. We simply need to include a reference in our code to the structure defined by the DBMS vendor. We do this through the following declaration:

```
extern static xa_switch_t XASWITCH;
```

where XASWITCH is the name of the structure defined by the DBMS. For DB2, this structure is actually named db2xa_switch. In our code we include the line exactly as shown above and then use a #define statement to translate XASWITCH to db2xa_switch.

The open string, close string, and serialization string are described by the DBMS vendor. They are the character strings that can be passed to the DBMS to connect and disconnect from the database and to state whether we want to serialize the XA communication or allow parallel requests. For DB2, the open string contains the database name, the user id, and the password, separated by commas. There is no close string, and the serialization string can be omitted if we want to serialize the XA access (which we normally do). One issue is that the Encina function call that uses these strings references them all as one concatenated string that is separated by commas. Because DB2 also uses comma separation within the strings, we must “escape” the DB2 commas with a backslash to prevent the Encina function from processing them as separators. Figure 87 on page 236 shows the code that sets up the XA resource access.

```
...
rmName = getRmName();
mst = mon_RegisterRmi(XASWITCH, rmName, &rmiId);
if ( !showStatus("mon_RegisterRmi",mst) ) return FALSE;
...
```

*Figure 87. Registering the DBMS As an XA Resource*

After we register the database as an XA resource, we can write code just as we would normally, except that we do not issue any CONNECT or DISCONNECT commands, as this is taken care of for us through the XA interface. We do not issue COMMIT or ROLLBACK statements because Encina, not the DBMS, handles the transaction processing.

The code, before preprocessing, for the `getOrderItem` function shows a variety of access methods, including the SELECT ... INTO process for getting a single row of data and the CURSOR processing to retrieve multiple rows (see Figure 88 on page 237 to Figure 90 on page 239).

Refer to your DBMS documentation for details about writing embedded SQL for your particular database and for the exact information required to make an XA connection to the database. However, there is so much similarity across products that the code shown in Figure 88 on page 237 to Figure 90 on page 239 will probably work with very little change for most databases.

```
int getOrderInfo(long orderId, OrderInfoPtr* ordInfo, RpcReturn*
retValue)
{
char tmpErrorCode[80];
int indx = 0;
EXEC SQL WHENEVER SQLERROR GOTO error_exit;
EXEC SQL BEGIN DECLARE SECTION;
long db_orderId;
char db_orderDateTime[27];
char db_orderStatus[11];
char db_orderStatusDateTime[27];
char db_recipName[21];
char db_recipAddr1[51];
char db_recipAddr2[51];
char db_recipCity[16];
char db_recipState[3];
char db_recipZip[11];
char db_paymentCardType[21];
char db_paymentCardNumber[21];
char db_paymentCardExpDate[11];
char db_paymentCardHolder[21];
long db_numProds;
long db_productId;
char db_productName[21];
float db_productPrice;
long db_orderedQty;
EXEC SQL END DECLARE SECTION;
LogEntry(1, DEBUGMSG, "Begin OrderDbSqlAny::getOrderInfo()");
initRpcReturn(retValue);
db_orderId = orderId;
LogEntry(2, DEBUGMSG, "Selecting row count from OrderedProduct");
EXEC SQL
SELECT COUNT(*)
INTO :db_numProds
FROM dbo.OrderedProduct
WHERE orderId = :db_orderId;
/* allocate space */
```

*Figure 88. (Part 1 of 3) Database Access Code for the getOrderInfo() Function*

```
/* issue the select for the basic order info */
LogEntry(2, DEBUGMSG, "Selecting from CustOrder");
EXEC SQL
SELECT *
INTO
:db_orderId,
:db_orderDateTime,
:db_orderStatus,
:db_orderStatusDateTime,
:db_recipName,
:db_recipAddr1,
:db_recipAddr2,
:db_recipCity,
:db_recipState,
:db_recipZip,
:db_paymentCardType,
:db_paymentCardNumber,
:db_paymentCardExpDate,
:db_paymentCardHolder
FROM dbo.CustOrder
WHERE orderId = :db_orderId;
(*ordInfo)->orderId = db_orderId;
strcpy((*ordInfo)->orderDateTime, db_orderDateTime);
strcpy((*ordInfo)->orderStatus, db_orderStatus);
strcpy((*ordInfo)->orderStatusDateTime, db_orderStatusDateTime);
strcpy((*ordInfo)->recipName, db_recipName);
strcpy((*ordInfo)->recipAddr1, db_recipAddr1);
strcpy((*ordInfo)->recipAddr2, db_recipAddr2);
strcpy((*ordInfo)->recipCity, db_recipCity);
strcpy((*ordInfo)->recipState, db_recipState);
strcpy((*ordInfo)->recipZip, db_recipZip);
strcpy((*ordInfo)->paymentCardType, db_paymentCardType);
strcpy((*ordInfo)->paymentCardNumber, db_paymentCardNumber);
strcpy((*ordInfo)->paymentCardExpDate, db_paymentCardExpDate);
strcpy((*ordInfo)->paymentCardHolder, db_paymentCardHolder);
/* issue the select for the ordered product list */
LogEntry(2, DEBUGMSG, "Declaring cursor for product list");
EXEC SQL DECLARE c1 CURSOR FOR
SELECT productId, productName, productPrice, quantity
FROM OrderedProduct WHERE orderId = :db_orderId;
EXEC SQL OPEN c1;
EXEC SQL WHENEVER NOT FOUND GOTO not_found;
```

*Figure 89. (Part 2 of 3) Database Access Code for the getOrderInfo() Function*

```
(*ordInfo)->item[indx].productId = db_productId;
strcpy((char*)(*ordInfo)->item[indx].productName, db_productName);
(*ordInfo)->item[indx].productPrice = db_productPrice;
(*ordInfo)->item[indx].orderedQty = db_orderedQty;
}
not_found:
EXEC SQL WHENEVER NOT FOUND CONTINUE;
LogEntry(1, DEBUGMSG, "End OrderDbSqlAny::getOrderInfo()");
return TRUE;
error_exit:
EXEC SQL WHENEVER SQLERROR CONTINUE;
LogEntry(1, ERRORMSG, "OrderDbSqlAny::getOrderInfo() FAILED with %d",
sqlca.sqlcode);
sprintf(tmpErrorCode, "%d", sqlca.sqlcode);
setRpcReturn(retValue,FATALRETURN,"OrderDb",tmpErrorCode);
return FALSE;
}
```

*Figure 90. (Part 3 of 3) Database Access Code for the getOrderInfo() Function*

### 10.2.3.4 Non-XA Access to Databases

One final note about database access from Encina servers concerns using the database nontransactionally. In general, you can include code within your server to access a database just as you would if you were making this access from a standard database client program. You would issue whatever logon or connection commands that are required to establish a normal (non-XA) connection to the DBMS. You would then issue whatever database commands you wanted and disconnect. The database access code you write will look just like what you would write for the XA processing except for the inclusion of the connect and disconnect commands. It is possible to have a mix of XA and non-XA database access within the same server.

There are a number of precautions to consider when mixing XA and non-XA connections within Encina servers. First is the issue of thread safety. If the database libraries are not thread safe, you will need to ensure that you perform only one database activity at a time, regardless of the number of threads that are processing in your server. Run your server with a MON_EXCLUSIVE scheduling policy (in which case it runs only one processing thread) or use a mutex to serialize the database operations across multiple threads. Another issue arises when you attempt to mix XA and non-XA connections within the same server. The two types of access could deadlock!

### 10.2.4 RQS Queue Processing

In addition to accessing a relational database, each of our servers will also access the RQS. The OrderProcServer will enqueue a queue item containing the ID of a new order that has to be verified. The VerificationServer will periodically remove items from the queue and use the order ID to access the data in the database and verify the correctness of the order information, specifically the payment information. After making a decision about the order, the VerificationServer will either update the database status to approve the order or set the status to either a failure or warning code. For orders that are marked with a failure or warning, a queue item containing the order ID is enqueued into another queue, which serves as a work list for manual reviews of orders.

We will implement two queues, one for items pending review and one for items marked for manual review. The enqueue operations will be done to specific queues, but the dequeue processing will be done to queue sets (mostly to demonstrate queue sets and their processing).

#### 10.2.4.1 Verify Queue Functions

As with the order database, we considered the verification queues as an object during the analysis and design. We have identified all of the operations required of the queuing system to support both servers. We will implement these functions in a common code module named VerifyQueue. Figure 91 on page 241 shows the functions implemented as listed in the header file for this module, VerifyQueue.h.

```
 #include <rqs/rqs.h>
 #include "OrderProcCommon.h"

 /* global variable for RQS server handle */
 static rqs_serverHandle_t rqsHandle;

 /* structure definition for queue elements */
 typedef struct {
 int orderId;
 } QueueItem;

 /* defined values for enqueue/dequeue "work" parm */
 #define REVWARN 1
 #define REVFAIL 2

 int initVerifyQueue();
 int addToVerifyQueue( QueueItem qItem, RpcReturn* ret );
 int addToReviewQueue( QueueItem qItem, unsigned long workInd,
RpcReturn* ret );
 int removeFromVerifyQueue( QueueItem* qItem, RpcReturn* ret );
```

*Figure 91. Verify Queue Function Prototypes (VerifyQueue.h)*

#### 10.2.4.2 RQS Queue Processing Example

The programming interface to RQS is conceptually simple, consisting of functions to initialize access to RQS, enqueue, dequeue, and disconnect. However, there is some variety in how these operations can be done. You can dequeue from either a specific queue or a queue set that may contain many queues. You can associate “work” data with a queue entry when it is enqueued and retrieve this data when you dequeue the queue entry. You can also perform nonqueue operations such as searches by key. For our case study application we will use enqueuing and dequeuing, using *work data*, and process both queues and queue sets. We will not use the search capabilities or use any of the administrative APIs that RQS provides.

The function call to initialize RQS is made as part of the server’s startup processing. RQS must be initialized after the application server initialization function has been called and before the server begins processing RPCs. We will implement the initialization by calling the `init VerifyQueue` function from within the `serverPostInit` function of the server’s implementation file. Figure 92 on page 242 shows the sections of code that perform the RQS initialization.

```
from the OrderProcServer.c file:

int ServerPostInit() {
.
.
.
if ( !initVerifyQueue() )
return FALSE;

return TRUE;
}


from the VerifyQueue.c file:

int initVerifyQueue() {
rqs_status_t rst;
LogEntry(1, DEBUGMSG, "Beginning VerifyQueue init for %s.",
g_sconf.rqsName);
if ( strlen(g_sconf.rqsName) == 0 ) {
LogEntry(1,ERRORMSG,"RQS server name must be specified!");
return FALSE;
}
if ( strlen(g_sconf.pendingQueueName) == 0 ) {
LogEntry(1,WARNINGMSG,"Pending queue name not specified!");
}
if ( strlen(g_sconf.reviewQueueName) == 0 ) {
LogEntry(1,WARNINGMSG,"Review queue name not specified!");
}
if ( strlen(g_sconf.pendingQueueSet) == 0 ) {
LogEntry(1,WARNINGMSG,"Pending queue set name not specified!");
}
if ( strlen(g_sconf.reviewQueueSet) == 0 ) {
LogEntry(1,WARNINGMSG,"Review queue set name not specified!");
}

rst = rqs_GetServerHandle( g_sconf.rqsName, &rqsHandle );
```

*Figure 92. RQS Initialization*

The parameters to the `rqs_GetServerHandle()` function are a string containing the full CDS name of the server (we get this from our server configuration structure), and a pointer to an rqs_serverHandle_t variable. We defined the rqs_handle_t variable as a global static variable, which allows any of the functions within the VerifyQueue module to use it.

The queue item that we will enqueue and dequeue from our queues is very simple, a single long integer for the order id. The code used to enqueue an item to the queue for failed reviews will show both the enqueue process and the manipulation of the work data that can be associated with a queue entry. We use the work data to hold simple code specifying whether the order has a warning failure or a fatal failure (see Figure 93 on page 243).

```
 int addToReviewQueue( QueueItem qItem, unsigned long workInd,
RpcReturn* ret ) {
 rqs_status_t rst;
 int queueDataLen;
 void* queueDataPtr;
 rqs_unsignedInt64_t rqs_workInd;
 rqs_elementId_t elemId;

 LogEntry(1, DEBUGMSG, "Beginning ReviewQueue add.");
 initRpcReturn(ret);
 queueDataLen = sizeof(qItem.orderId);
 queueDataPtr = malloc(queueDataLen);
 memcpy( queueDataPtr, (void*)&(qItem.orderId), sizeof(qItem.orderId) );
 rqs_workInd.low = workInd;
 rst = rqs_Enqueue(rqsHandle,
g_sconf.reviewQueueName,
"OrderIdQueEntry",
queueDataLen,
queueDataPtr,
&rqs_workInd,
&elemId );
 if ( !showStatus("rqs_Enqueue",rst) ) {
 setRpcReturn(ret,FATALRETURN,"","");
 return FALSE;
 }
 free(queueDataPtr);
 LogEntry(1, DEBUGMSG, "End of ReviewQueue add.");
```

*Figure 93. Enqueue Processing for RQS*

The input parameters to the `rqs_Enqueue()` function are a handle to the server (established by the initialization routine), the name of the server (from our configuration structure), the name of the queue item structure as defined to RQS, the length of the data being provided, a void pointer to the data to be enqueued, and a pointer to the work data (variable of type rqs_unsignedInt64_t). The rqs_unsignedInt_64_t type is a structure consisting of two long integers, with variable names *low* and *high*. We place our integer code into the low component of the structure. A generated ID for the queue item is returned in an output parameter of type rqs_elementId_t. We have no use for the returned element ID in this case, so we simply provide a variable of the right type for the call and then ignore it. If you look at the complete code listings, you will see that the enqueue operation for the pending queue does not use a work data parameter, and the call by the rqs_Enqueue is made with the value NULL as the work parameter.

Items from the queues are dequeued by referencing the queue set to which the queue is assigned. Multiple queues may be assigned to a queue set. RQS provides facilities for specifying priorities and weighting factors that affect how items are actually dequeued from the queues within a set, but from a programming standpoint, there is little difference between dequeuing from a queue set and from a specific queue. Refer to the Encina Administration Guides for information about creating queues and queue sets and specifying the parameters associated with them. We will use queue sets to dequeue items. Figure 94 on page 245 shows the code for dequeuing from the review queue. Again, the work data is processed as part of the operation.

```
int removeFromReviewQueue( QueueItem* qItem, unsigned long* workInd,
RpcReturn* ret ) {
rqs_status_t rst;
rqs_elementDescriptor_t* elementDescriptorPtr;
rqs_unsignedInt64_t rqs_workInd;
void* queueDataPtr;

LogEntry(1, DEBUGMSG, "Beginning ReviewQueue remove.");
initRpcReturn(ret);

rst = rqs_QSDequeue(rqsHandle,
g_sconf.reviewQueueSet,
rqs_deleteElement,
FALSE,
&elementDescriptorPtr );
if ( (rst != RQS_SUCCESS) ) {
setRpcReturn(ret,rst,"","");
if (rst == RQS_EMPTY_QSET)
LogEntry(2,DEBUGMSG,"PendingReviewQSet is empty");
return FALSE;
}
else {
queueDataPtr = elementDescriptorPtr->value;
memcpy(&(qItem->orderId), queueDataPtr, sizeof(qItem->orderId));
rqs_workInd = elementDescriptorPtr->work;
*workInd = rqs_workInd.low;
rqs_Free( elementDescriptorPtr );
```

*Figure 94. Dequeuing from the Review Queue Set*

The input parameters to the `rqs_QSDequeue()` function are the server handle, the name of the queue (from our configuration structure), and a flag specifying the disposition of the element after the dequeue (here we have it deleted). The final parameter is the output pointer to a queue element description structure (type rqs_elementDescriptor_t). The element descriptor contains a number of things, including the name of the queue from which the element was taken, a pointer to the queue item itself, and the work data. After the dequeue operation, we use the pointer to the descriptor element to access the queue item data (copying it into a variable of our QueueItem type). We also access the work data, from the low component of the rqs_unsignedInt64_t structure that defines work items.

Because we are letting RQS allocate the element descriptor structure, we have to to be sure to free the memory allocation. A special function, rqs_Free(), takes a pointer to an element descriptor and safely releases the memory allocation. You must always call the rqs_Free function after a dequeue to avoid memory leaks and possible data corruption.

The last issue to cover with our RQS access is the disconnection process. We perform this process during the shutdown phase of our server processing, after the RPC listen loop has terminated and before the server program ends. We handle the disconnection similary to the initialization. We call the closeVerifyQueue() function that is part of our VerifyQueue module from within the serverCleanUp() function of the server's implementation file (see Figure 95 on page 246).

```
from OrderProcServer.c:

int ServerCleanUp() {
RpcReturn retValue;

if ( !closeVerifyQueue(&retValue) )
return FALSE;

return TRUE;
}

from VerifyQueue.c:

int closeVerifyQueue() {
rqs_status_t rst;

LogEntry(1, DEBUGMSG, "Beginning VerifyQueue close.");

rst = rqs_FreeServerHandle( rqsHandle );

if ( !showStatus("rqs_FreeServerHandle",rst) ) return FALSE;
```

*Figure 95. Disconnecting from RQS*

The rqs_FreeServerHandle simply takes a handle to the server from which you want to disconnect. Here we specify the global variable that has held the handle since the initialization process established it.

### 10.2.5 Host Access with PPC

Our product database is on a host system and is accessible through a connection to CICS. Within CICS there are programs that provide the data for the two requests we will make, a listing of products, and an update of a product's inventory quantity. We will use these programs by invoking them remotely through the services of the PPC Gateway. We will implement the PPC access to the product database within a common module named ProductDbPPC.

### 10.2.5.1 Product Database Functions

The header file for the module defines the functions we will implement (see Figure 96 on page 247).

```
#define MAX_BUFFER_SIZE 32767

typedef struct {
char rqstName[10];
} ProductListInBuf;

typedef struct {
char productId[10];
char productName[25];
char productDesc[200];
char productPrice[15];
char productInventory[10];
} ProductListEntry;

typedef struct {
char rqstResult[4];
char listCount[4];
ProductListEntry entry[1];/* list of products with 1 place holder */
} ProductListOutBuf;

typedef struct {
char rqstName[10];
char productId[10];
char qtyChange[10];
} ProdQtyUpdateInBuf;

typedef struct {
char rqstResult[4];
} ProdQtyUpdateOutBuf;

int initProductDb();
```

*Figure 96. Product Database Functions (ProductDbPPC.h)*

### 10.2.5.2 PPC Coding Example

The PPC services can be used to support Encina-to-CICS conversations, CICS-to-Encina conversations, and Encina-to-Encina conversations. We will use only the first of these. For more information about the administration and programming of PPC, see the *Encina Server Administration Guide* and the *PPC Services Programming Guide*. For Encina-to-CICS communication we will write routines in our server to make it a PPC Executive application that uses the PPC gateway. In this mode, the PPC Executive functions allow us to write a program that becomes the allocating partner in an LU 6.2 conversation. The PPC gateway serves as a bridge between the TCP/IP network in which our programs run and the SNA network through which we access CICS. The LU 6.2 protocol requires that the participants in the conversation be defined as LUs. Part of the PPC gateway administration involves establishing a connection between the PPC gateway and the SNA process running on the same computer. The SNA process and the PPC gateway must be configured to be recognized as a particular LU and to have knowledge of other LUs that are potential conversation partners. Refer to the appropriate SNA documentation for your product and the *Encina Administration Guide* for details.

The basic outline for a PPC access to a CICS program is fairly simple. The PPC communications must first be initialized. Then a conversation allocation request is made. A series of send and receive calls are made to exchange data with the conversation partner, followed by a deallocation. You may have situations where you will design a new PPC interaction, but more often you will be developing a new partner to work with an existing program. The techniques for coding an allocator program are the same regardless of whether the acceptor is a CICS program accessed through the PPC Gateway or another Encina program accessed directly through the PPC Executive. It is also possible to have an Encina application server be the acceptor for a CICS allocator. The techniques for writing an Encina acceptor are the same regardless of who the allocator is. Therefore it is possible to develop code for an Encina-to-CICS conversation by using an Encina-to-Encina conversation during the initial development. This is the strategy that we will use for the case study application.

Figure 97 on page 249 outlines the pattern of the conversation we will implement. The Partner 1 program is the allocator program, which will be the OrderProcServer in our application. The Partner 2 program is the acceptor program, which we will simulate initially with another Encina application server, but which will later be the existing CICS program.

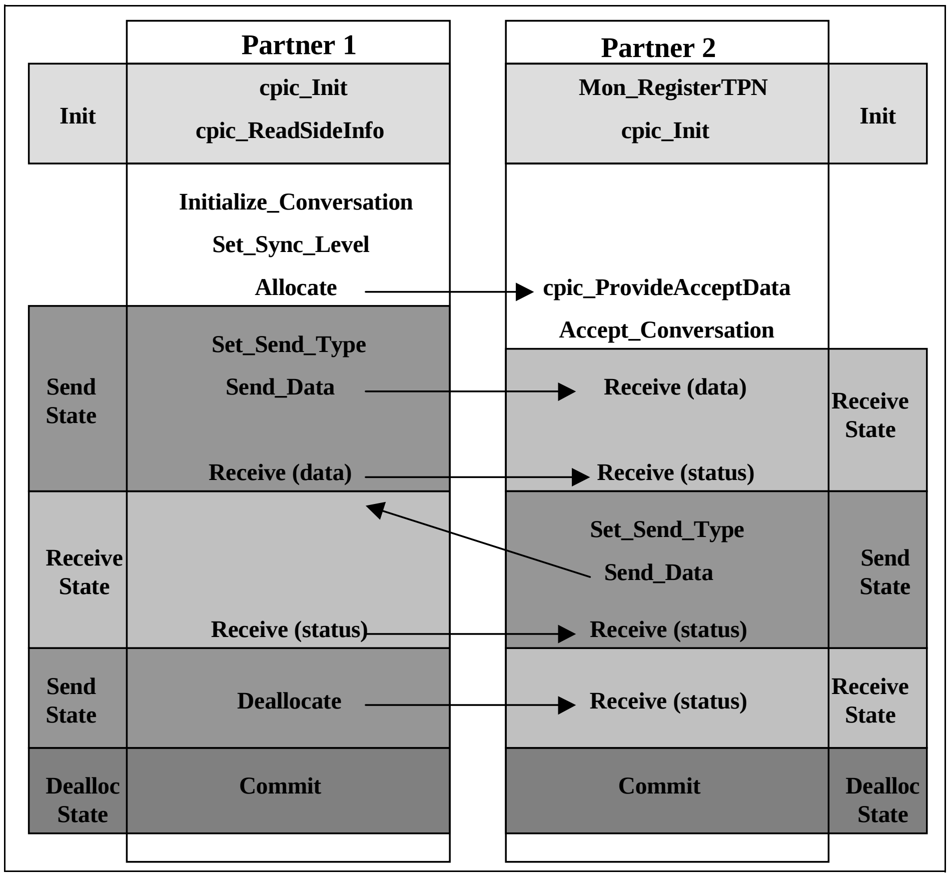


*Figure 97. Conversation Pattern for the Product Database Access*

In this section we focus on the allocator code that we will use within the OrderProcServer. Initializing a PPC allocator involves two calls to PPC Executive functions. The first call is to the `cpic_Init()` function. There is one parameter to this call; it names the LU by which it will be known to our partner in the conversation. This is the fully qualified name in the form NETWORK.LUNAME. The second call is to load a special PPC configuration file known as the *side information* file. This file contains special syntax that allows you to describe the conversation partners and the acceptor programs you want to use. The `cpic_ReadSideInfo()` function takes a single parameter that gives the file name of the side information file. The `initProductDb` function contains these two calls. This function is called as part of the server's postinitialization processing (see Figure 98 on page 250).

```
int initProductDb() {
CM_RETCODE cmr;
LogEntry(1, DEBUGMSG, "Beginning ProductDbInit() function...");
if ( strlen(g_sconf.ppcLuPartner) == 0 ) {
LogEntry(2,ERRORMSG,"PPC LU must be specified!");
return FALSE;
}
if ( strlen(g_sconf.ppcSideInfoName) == 0 ) {
LogEntry(2,ERRORMSG,"PPC side info file name must be specified!");
return FALSE;
}
LogEntry(2,INFOMSG,"Initializing PPC with LU = [%s]",
g_sconf.ppcLuPartner);

cmr = cpic_Init(g_sconf.ppcLuPartner);
if ( !showCpicStatus("cpic_Init",cmr) ) return FALSE;
LogEntry(2,INFOMSG,"Reading side info file [%s]",
g_sconf.ppcSideInfoName);

cmr = cpic_ReadSideInfo(g_sconf.ppcSideInfoName);
if ( !showCpicStatus("cpic_ReadSideInfo",cmr) ) return FALSE;
LogEntry(1, DEBUGMSG, "End of ProductDbInit() function.");
```

*Figure 98. Initializing PPC (from initProductDb.c)*

Notice that the error processing is a little different here than elsewhere. All of the calls in the CPI-C (PPC Executive) library have the potential to return an error with the enumeration symbol of CM_PRODUCT_SPECIFIC_ERROR. If this error is returned from the call, an addition function call, `cpic_GetProductSpecificDetail`, returns a value of type ppc_status_t. This return code provides more details for this type of error. The processing to get product-specific error returns is handled within the `showCpicStatus()` function.

Figure 99 on page 251 shows the side information file we are using for our case study application. You can find more details on the contents of side information files in the *Encina PPC Services Programming Guide*.

```
 /* LU6.2 Application */
 sideInfo {
 "ORDERPRC",/* symbolic dest name */
 "",/* mode name - defaults */
 "ProductDb",/* partner name */
 "CPICRCV",/* TPN - remote program */
 ENCRYPT_NONE,/* encryption - not used */
 SECURITY_NONE,/* security setting */
 "","",/* user id and password -
ignore for SECURITY_NONE */
 }

 partner {
 "ProductDb",/* partner name */
 CONNECTION_GWY_TCP,/* connection type */
 "SNAPI2",/* scheduler entry name -
```

*Figure 99. The Side Information File*

Allocating a conversation consists of several related PPC function calls. We first make a call to initialize the conversation. This call requires that we provide a parameter of type CONVERSATION_ID. Despite the name, this is a char * data type under the covers and must be initialized to an empty string. The initialization call will set a unique value in the string. This conversation id string is then used as an input function to each of the subsequent calls throughout the conversation. The call also requires a symbolic destination name. This is the key that selects a particular entry in the side information file. The sideinfo entry is then paired with a partner entry, and between these all of the information needed for the conversation is available. The next call sets the synchronization level for the conversation. There are three synchronization levels, levels 0, 1, and 2, that give the level of transaction support. Level 0 indicates no transactions, and level 2 indicates full transactional support. We will use synchronization level 2. The next call, which is optional, sets the type of the conversation. Our choices are “mapped” and “basic." Despite the names, the mapped conversation is much simpler than the basic conversation. Mapped conversations allow the participants to determine the structure of the data that is passed between them. Basic conversations use a standardized structure. We will use mapped conversations. Finally, we make a call to allocate the conversation. If this call returns with a CM_OK return code, the program on the other end has been activated and is expected to be ready to receive data. Our program is in the “send” state after the allocate call. Figure 100 on page 252 shows the code for allocating the conversation from the `exchangeBuffers()` function.

```
...
Initialize_Conversation(convId, destName, &cmr);
if ( !showCpicStatus("Initialize_Conversation",cmr) ) return FALSE;

Set_Sync_Level(convId, &sl, &cmr);
if ( !showCpicStatus("Set_Sync_Level",cmr) ) return FALSE;

Allocate(convId, &cmr);
if ( !showCpicStatus("Allocate",cmr) ) return FALSE;
```

*Figure 100. Allocating a Conversation (from ProductDbPPC.c)*

We will use a simple conversation pattern that calls for a single send and receive exchange of data with our partner. Since we are the allocating party, we send first. The first call we make is to set the type of the send. We want to send and then immediately turn the conversation around so that we can receive, so we use a send type of CM_SEND_AND_FLUSH. We then make a call to the Send_Data function to send a buffer to our conversation partner. The Send_Data function takes a pointer to a buffer of data, a pointer to an integer giving the length of the buffer, and a pointer to a variable to hold code which verifies that the conversation turnaround request has been acknowledged. Of course the conversation ID and a return code are also part of the parameter list. Figure 101 on page 252 shows the function calls related to the data send operation.

```
...
Set_Send_Type(convId, &sendType, &cmr);
if ( !showCpicStatus("Set_Send_Type",cmr) ) convError = TRUE;

Send_Data(convId, inbuf, &inbufLen, &requestToSendReceived, &cmr);
if ( !showCpicStatus("Send_Data",cmr) ) convError = TRUE;
...
```

*Figure 101. Sending Data through PPC (from ProductDbPPC.c)*

In our simple conversation pattern, we are now expecting to receive a buffer from our partner followed by a deallocation of the conversation. We handle both of these in one function, readAndEnd, because we expect that the function call we make to receive the buffer will have a return parameter verifying that our partner is deallocating following its send. This function is a little complex because of the variations in how we might be notified of the deallocation and the possibility that our partner might be notifying us that it is ending abnormally. Figure 102 on page 253 shows the Receive function call.

```
 maxLen = *outbufLenPtr;  /* use the input value as the max */

 LogEntry(3,DEBUGMSG,"Receiving data...");
 Receive(convId, outbuf, &maxLen,
&dataReceived, outbufLenPtr,
&statusReceived, &requestToSendReceived,
&cmr);
 if ( !showCpicStatus("Receive",cmr) ) convError = TRUE;

 LogEntry(3,DEBUGMSG,"Receiving status...");
 Receive(convId, outbuf, &maxLen,
&dataReceived, &dummyLen,
&statusReceived, &requestToSendReceived,
&cmr);
```

*Figure 102. Receiving Data through PPC (from ProductDbPPC.c)*

The parameters to the `Receive` function, following the conversation ID, begin with a pointer to the buffer that is received and a pointer to an integer containing the maximum length of the data buffer. The next parameter is an indicator telling us whether the data we have received is all of the data that needs to be sent. This indicator can be used to handle multibuffer transfers of very large buffers. The next parameter is an integer giving the actual length of the buffer that was transferred. This is followed by an indicator as to whether any abnormal status has been returned, and the conversation turnaround indicator as was used in the Send_Data call.

Once we have successfully received the data and the notification that we have send control, we can initiate the deallocation of the conversation. Figure 103 on page 253 shows the code that ends the conversation.

```
LogEntry(3,DEBUGMSG,"Deallocating conversation ...");
deallocType = CM_DEALLOCATE_SYNC_LEVEL;
Set_Deallocate_Type(convId, &deallocType, &cmr);
Deallocate(convId, &cmr);
if ( !showCpicStatus("Deallocate",cmr) ) convError = TRUE
```

*Figure 103. Deallocation after a Receive (from ProductDbPPC.c)*

To get a complete picture of the flow of the conversation, refer to the complete code listing for the ProductDbPPC.c file.

Before moving on to the implementation of the business functions related to the product database, we need to mention the issue of conversion between

EBCDIC and ASCII coded text data. If the CICS system that we are conversing with is running on an MVS system, its character data will be coded in EBCDIC. There are two function calls that convert between these two character coding systems. The `Convert_Outgoing` function translates ASCII data into EBCDIC data. The `Convert_Incoming` function translates EBCDIC data into ASCII data. These calls need a pointer to the data buffer, a length indicator, and return a result code. The conversion is done in place on the buffer that is pointed to - meaning that the buffer after the call will be in the altered coding scheme. The ASCII data will be null terminated after the conversion. We do not use the conversion routines in our case study code because the CICS system we are talking to is a CICS/6000 system running on AIX using the ASCII coding scheme.

As an example of how the low-level function, `exchangeBuffer(),` is used to actually transfer our product database data, we look at the `updateProductQty` function. This function passes a buffer containing the information needed by the CICS program to make the update and receives a buffer containing a result code from the CICS program. We manage the buffers by defining structures that describe the buffer we are sending and the buffer we are receiving for this request. The structure definitions we will use are coded in the ProductDbPPC.h file, shown in Figure 96 on page 247.

Notice how these buffers are defined. First, all of the data is in character format, including the quantity field. We will make the decision here to exchange all of our information as character data to avoid the problems with numeric conversions between different systems. Because our partner here is another UNIX process, there really is not that much of an issue. If our partner were on an MVS platform, it would typically be a COBOL CICS program using EBCDIC coding and a very different style of numeric representation. It is typical to convert numeric data to character data for transfers and let the partner program decide how to convert the character representation back into a numeric format. In using these structures to map the buffer for us, we must ensure that we do not allow the compiler to add extra bytes to the structure (to achieve some level of alignment on word or double word boundaries in memory). There are compiler switches that control the alignment option for structures, and different compilers have different defaults, so be careful in setting up your build options.

The code for the `updateProductQty` function begins by populating the structure it will pass to the CICS program. It then calls the functions to allocate a conversation, send the buffer, and receive the result. The buffer that is returned is interpreted using the output structure (although it is trivial, in this case we stick to the general processing pattern). The function ends after interpreting the return code it finds in the returned data. Figure 104 on page

255 to Figure 105 on page 256 shows the code for the `updateProductQty` function.

```
int updateProductQty(long productId, long qtyChange, RpcReturn *ret) {
SYNC_LEVEL sync = CM_NONE;
ProdQtyUpdateInBuf inbufStruct;
ProdQtyUpdateOutBuf outbufStruct;
char *inbuf;
char *outbuf;
int inbufLen;
int outbufLen;
char tmpStr[11];
char rqstReturnTmp[5];

LogEntry(2,DEBUGMSG,"Beginning updateProductQty function...");
/* populate the input buffer for the conversation */
strncpy(inbufStruct.rqstName, "PRODQTYUPD", 10);
sprintf(tmpStr,"%10d", productId);
strncpy(inbufStruct.productId, tmpStr, 10);
sprintf(tmpStr,"%10d", qtyChange);
strncpy(inbufStruct.qtyChange, tmpStr, 10);
inbufLen = sizeof(inbufStruct);
inbuf = malloc(inbufLen + 1);
memcpy(inbuf, &inbufStruct, inbufLen);
inbuf[inbufLen] = '\0';
outbufLen = sizeof(outbufStruct);
outbuf = malloc(outbufLen + 1);
memset(outbuf, ' ', outbufLen);
if ( !exchangeBuffers("UPDPRQTY", sync, inbuf, inbufLen,
outbuf, &outbufLen) ) {
free(inbuf);
free(outbuf);
setRpcReturn(ret,FATALRETURN,"","");
LogEntry(3,ERRORMSG,"The buffer exchange failed!");
return FALSE;
}
free(inbuf);
outbuf[outbufLen] = '\0';
memcpy(&outbufStruct, outbuf, outbufLen);
free(outbuf);
strncpy(rqstReturnTmp, outbufStruct.rqstResult, 4);
rqstReturnTmp[4] = '\0';
LogEntry(2,DEBUGMSG,"End of updateProductQty function...");
```

*Figure 104. (Part 1 of 2) The updateProductQty Function (from ProductDbPPC.c)*

```
if ( strcmp(rqstReturnTmp, "0000") == 0 ) {
setRpcReturn(ret,GOODRETURN,"","");
return TRUE;
}
else if ( strcmp(rqstReturnTmp, "0001") == 0 ){
setRpcReturn(ret,WARNINGRETURN,"ProductDb",rqstReturnTmp);
return TRUE;  /* convention here is good return for appl warning */
}
else {
setRpcReturn(ret,FATALRETURN,"ProductDb",rqstReturnTmp);
return FALSE;
}
}
```

*Figure 105.  (Part 2 of 2) The updateProductQty Function (from ProductDbPPC.c)*

### 10.2.6  Server Construction

In this section we describe the structure of the code that implements the Encina Monitor application servers. The pattern of the code is basically the same for each of our servers, but there are some differences. We consider the OrderProcServer in detail and then look at the features of the VerificationServer that are unique.

#### 10.2.6.1  Monitor Application Server Life Cycle

The basic steps in the life cycle of a monitor application server are:

- Establish the server's processing environment by retrieving any environment variables or command line arguments that will affect the subsequent processing.
- Register the interfaces the server provides with the Encina monitor.
- Perform any server-specific processing that needs to be done before the server goes through its own initialization routine. The most common activity here is the registration of XA resources that you will use in implementing the RPCs.
- Set the server options that will affect how the server initializes itself. Usually you will specifically set the server's scheduling policy. The scheduling policy determines whether or not the server will be able to handle concurrent requests.
- Initialize the server.
- Perform any server-specific processing that needs to be done after the server has initialized. The most common activity is the initialization of any

components that use native Encina transactions, such as PPC, RQS, or SFS.

- Begin the processing loop within which the server is available for processing RPCs. This function executes the entire time the server is up. During this time, clients can bind to the server and send RPCs to it.
- Perform any cleanup and shutdown processing that needs to occur before the server terminates. Typically you close any connections or files that are opened during the startup routines.

We will create a server function for each of the steps in the life cycle. Some of these functions will be the same for both servers. The main differences will be in what we do during the server pre-init and server post-init functions. Figure 106 on page 257 shows the header file that defines the functions for the `OrderProcServer`.

```
/* Function prototypes for the OrderProcServer */

#define XASWITCH db2xa_switch   /* swtich name for use with DB2 */

int EstablishEnv( int argc, char* argv[] );
int ExportInterface();
int ServerPreInit();
int SetServerOptions();
int ServerInit();
int ServerPostInit();
int ServerListenLoop();
int ServerCleanUp();
```

*Figure 106. Server Life-Cycle Functions (OrderProcServer.h)*

Notice the definition of the symbol XASWITCH that is included here. All of the functions that deal with the switch will use XASWITCH as the parameter. We define the actual name here so that we can easily change the name. Because different database products have different names for the switch, we chose not to use those names.

The server's main function simply calls each of these functions in turn to perform the required processing (see Figure 107 on page 258).

```
#include <stdio.h>
#include <tpm/mon_server.h>
#include <LogEntry.h>
#include <ServerVariables.h>
#include <MonStatusStrings.h>
#include <OrderProcIF.h>
#include <OrderDbDB2.h>
#include <ProductDbPPC.h>
#include <ServerConfig.h>
#include <VerifyQueue.h>
#include <OrderProcServer.h>

ServerConfig g_sconf;/* global server configuration */

extern struct xa_switch_t XASWITCH;

/**********************************************************************
**/
/*************** ORDERPROC SERVER MAIN FUNCTION *****************/
/**********************************************************************
**/

int main(int argc, char* argv[]) {
LogEntry(1, ALWAYS, "OrderProcServer Start-Up Beginning...\n");
if ( !EstablishEnv(argc, argv) ) return 1;
if ( !ExportInterface() ) return 1;
if ( !ServerPreInit() ) return 1;
if ( !SetServerOptions() ) return 1;
if ( !ServerInit() ) return 1;
if ( !ServerPostInit() ) return 1;
if ( !ServerListenLoop() ) return 1;
if ( !ServerCleanUp() ) return 1;
LogEntry(1, ALWAYS, "OrderProcServer stopped gracefully.\n");
return 0;
}
```

*Figure 107. OrderProcServer main() Function (from OrderProcServer.c)*

Because we defined a common structure for the server configuration variables, we simply choose which ones we need to use here and use the common code modules we created to do the bulk of the work. Figure 108 on page 259 shows the implementation of the `establishEnv` function.

```
int establishEnv( int argc, char **argv ) {
/* merge the command line arguements and the
   environment variables to establish the working
   environment for the server */
LogEntry(2,DEBUGMSG, "Begining server environment set-up.");
/* write the possible config variabes to the log */
LogConfigOptions();
/* call the set up functions for the needed variables */
setEnvMsgLevel(argc,argv,PREFER_CMD);
setEnvRetriveEnable(argc,argv,PREFER_CMD);
setEnvSchedulePolicy(argc,argv,PREFER_CMD);
setEnvDbUserId(argc,argv,PREFER_CMD);
setEnvDbPassword(argc,argv,PREFER_CMD);
setEnvDbName(argc,argv,PREFER_CMD);
setEnvPpcSideInfo(argc,argv,PREFER_CMD);
setEnvPpcLuPartner(argc,argv,PREFER_CMD);
setEnvRqsServer(argc,argv,PREFER_CMD);
setEnvPendingQueue(argc,argv,PREFER_CMD);
setEnvPendingQSet(argc,argv,PREFER_CMD);
setEnvReviewQueue(argc,argv,PREFER_CMD);
setEnvReviewQSet(argc,argv,PREFER_CMD);
setEnvThreadDelay(argc,argv,PREFER_CMD);
```

*Figure 108. Server Configuration: establishEnv() Function*

Each of our servers exports interfaces that provide RPC function entry points to client programs. During server startup, we must register the interfaces that are implemented within the server. We can register as many different interfaces as we like. We register the interface through a call to the `mon_InitServerInterface` function as shown in the implementation of the `ExportInterface()` function from our OrderProcServer in Figure 109 on page 260.

```
int exportInterface() {
mon_status_t mst;
LogEntry(2,DEBUGMSG, "Beginning interface export.");

mst = mon_InitServerInterface(MON_SERVER_INTERFACE(OrderProcIF,1,0));

if ( !showStatus("init OrderProcIF interface",mst) ) return FALSE;
LogEntry(2,DEBUGMSG, "End of interface export.");
return TRUE;
}
```

*Figure 109. Exporting the Server's Interface from OrderProcServer.c*

The parameters to the mon_InitServerInterface are specified using a macro provided with the Encina include files. The function actually takes two parameters; the interface and the entry point vector. These names are generated by the TILD compiler and can be found in the generated code. To simplify the coding of this call, the Encina header files define the MON_SERVER_INTERFACE macro, which takes the interface name as given in the TIDL source, the major version number, and the minor version number and expands them into the same “mangled” name that is produced by the TIDL compiler in the generated files. For this macro to work, you have to be careful not to include any white space between the interface name and the version numbers. After the successful execution of the `mon_InitServerInterface` function, the Encina Cell Manager has been notified that this interface will be provided by the server instance that is starting up.

The order in which we perform the interface initialization is not critical except that all interfaces must be initialized before making the call to initialize the server itself.

The next step in our startup process is to take care of any other processing that must be done before the server is initialized. Typically this involves registering any resources that we will access using the XA protocol. All XA resources must be registered before server initialization. We include all of the processing within the `serverPreInit()` function (see Figure 110 on page 261).

```
int serverPreInit() {
int retCode;
mon_status_t mst;
char* rmName;
int rmiId;

LogEntry(2,DEBUGMSG, "Beginning server pre-initilization.");

LogEntry(3, DEBUGMSG, "Beginning OrderDB XA Resource registration.");
getRmName(rmName);

mst = mon_RegisterRmi(XASWITCH, rmName, &rmiId);

if ( !showStatus("mon_RegisterRmi",mst) ) return FALSE;

LogEntry(3, DEBUGMSG, "End of OrderDB XA Resource registration.");
LogEntry(2,DEBUGMSG, "End of server pre-initilization.");
return retCode;
```

*Figure 110. XA Resource Registration serverPreInit() (from OrderProcServer.c)*

The `mon_RegisterRmi` function registers the XA resource. The parameters are the xa_switch_t variable provided by the vendor of the resource and a string providing an open string, close string, and serialization option. This string is referred to as the *rmName*. The function provides an output parameter giving the registered resource an ID number. The strategy we are using here is to use a #define statement to give the appropriate value to the XASWITCH symbol, and to call a function within the OrderDB module to give us the rmName string to use in the mon_RegisterRmi call. This allows us to make the registration code generic rather than having it written specifically for a given vendor's product. Figure 110 on page 261 shows the `getRmName()` function from the OrderDbDB2 implementation. Because DB2 uses commas to separate the database name, user id, and password in the open string, and the RM Name string is expected to be comma separated as well, we use backslash to "escape" the commas within the open string.

```
void getRmName( char * rmName )
{
sprintf( rmName , "%s\\,%s\\,%s,,",
g_sconf.dbName,g_sconf.dbUserId,g_sconf.dbPassword );
}
```

*Figure 111. Building the rmName String for DB2 (from OrderDbDB2.c)*

The final step before actually initializing the server is to set the options that affect how the server initialization will be done. Specifically we need to specify whether we want to allow environment retrieval within RPC calls and what the concurrency scheduling policy should be. With the environment retrieval option turned on, calls can be made within the implementation of an RPC that return various pieces of information about the context in which the RPC is being called. Unfortunately, making this information available involves considerable overhead, so this feature is usually disabled unless specifically required. The concurrency scheduling policy setting specifies whether the server will process RPCs one at a time or concurrently in multiple threads. If the server is set to concurrent processing, there is a default of 5 threads available for RPC processing. You can change this value by setting the value of the ENCINA_TPOOL_SIZE environment variable to the number of threads the server should support.

The calls to set these options are contained within our `setServerOptions` function, which is shown in Figure 112 on page 262. Notice that we are using the values set from our environment to make the calls to these functions.

```
int setServerOptions() {
mon_status_t mst;

LogEntry(2,DEBUGMSG, "Beginning set up of server options.");

/* set the environment retrieval flag */
mst = mon_RetrieveEnable( g_sconf.retrieveEnabled );
if ( !showStatus("mon_RetrieveEnable",mst) ) return FALSE;

/* set the scheduling policy for the server */
if ( strcmp(g_sconf.schedulePolicy,"MON_CONCURRENT_SHARED") == 0 ) {
mst = mon_SetSchedulingPolicy( MON_CONCURRENT_SHARED );
}
else if ( strcmp(g_sconf.schedulePolicy,"MON_EXCLUSIVE") == 0 ) {
mst = mon_SetSchedulingPolicy( MON_EXCLUSIVE );
}
else {
LogEntry(3,ERRORMSG,"The ServerConfig value for schedulePolicy is not
'MON_CONCURRENT_SHARED' or 'MON_EXCLUSIVE'.");
LogEntry(3,ERRORMSG,"   Check the command line options and environment
variables.");
}
if ( !showStatus("mon_SetSchedulingPolicy",mst) ) return FALSE;
```

*Figure 112. Setting Server Options: setServerOptions() Function*

Now we are ready to initialize the server with a call to `mon_InitServer`. This call takes one parameter and after successful completion, the server is initialized and capable of doing work, including transactional work. The `serverInit()` function, shown in Figure 113 on page 263, performs this processing for our server.

```
int serverInit() {
mon_status_t mst;

LogEntry(2,DEBUGMSG, "Beginning server initialization.");

mst = mon_InitServer();

if ( !showStatus("mon_InitServer",mst) ) return FALSE;

LogEntry(2,DEBUGMSG, "End of server initialization.");
return TRUE;
```

*Figure 113. Initializing the Server: initServer() Function*

Once the server is initialized, we can perform any processing that requires transactions or any other type of server processing. This is the point in the startup process where we need to initialize any of the native Encina resources such as SFS, RQS, and PPC. In our case we will be using RQS and PPC, so these facilities are initialized in our `serverPostInit()` function (see Figure 114 on page 263).

```
int ServerPostInit() {
LogEntry(2,DEBUGMSG, "Beginning server post initilization.");

if ( !initProductDb() )
return FALSE;

if ( !initVerifyQueue() )
return FALSE;

LogEntry(2,DEBUGMSG, "End of server post initilization.");
return TRUE;
```

*Figure 114. Post-Initialization Processing: serverPostInit() Function*

Our design calls for the actual initialization code to be part of the modules that deal with the product database (for PPC) and the verification queue (for RQS), so here we simply call the functions from these common modules. The implementations of these functions are covered in the common code sections for the product database and the verification queue.

Finally we are at the point where the server can begin to handle RPC requests, assuming that all has gone well so far. To begin the process of listening for and processing RPCs, we make a call to the `mon_BeginService` function. This function executes until the server shuts down. This processing is contained within our `serverListenLoop()` function shown in Figure 115 on page 264.

```
int serverListenLoop() {
mon_status_t mst;

LogEntry(2,DEBUGMSG, "Beginning server RPC listen loop.");

mst = mon_BeginService();

if ( !showStatus("mon_BeginService",mst) ) return FALSE;
LogEntry(2,DEBUGMSG, "End of server RPC listen loop.");
return TRUE;
```

*Figure 115. Listening for RPCs: serverListenLoop() Function*

While the server is in the RPC listening loop, it monitors the IP ports that were assigned to its interfaces during the previous steps in the startup. Through the services of the Encina Cell Manager, clients are bound to the server. When a bound client calls an RPC that is part of the server's interface, the server passes the RPC parameters to the function we have written to implement the RPC. The server will invoke our function from the main server thread or within a new thread, depending on whether the server is operating in MON_EXCLUSIVE or MON_CONCURRENT_SHARED mode. The way in which we implement RPCs is covered in the next section.

Server shutdown takes place because a shutdown request is received from the Encina Cell Manger or due to a call to the `mon_TerminateSever` function. Typically, the server waits for a shutdown request instead of having code to shut itself down. The processing that needs to be performed as part of the shutdown process depends on what is being done in the server. in general, any resource that you have specifically opened should be closed now. The shutdown processing for the OrderProcServer has only to end its access to RQS. This is done through a call to the `closeVerifyQueue` function implemented within the `VerifyQueue` common code module (see Figure 116 on page 265).

```
int serverCleanUp() {
RpcReturn retValue;

if ( !closeVerifyQueue(&retValue) )
return FALSE;

return TRUE;
}
```

*Figure 116. Server Shutdown Processing: closeVerifyQueue Function*

#### 10.2.6.2 Implementing the Interface Functions

Each of the functions in each of the interfaces that the server supports must be implemented by a function we code. The TIDL and IDL compilers produced a set of code that handles the process of passing control and data from a received RPC to the function that we have implemented. In general, there is not much difference between coding a function to support an RPC and coding any other function.

The signature of the function is defined by the function definition in the TIDL file. If you look at the generated code, you will find that the functions you defined in the TIDL are changed to include some additional parameters. These additional parameters are added to the function call by the generated code used on the client-side. The additional parameters are received and processed within the server-side generated code so that when our function is actually called, only the parameters specified originally in the TIDL source are passed to us.

We have chosen to structure our code in such a way that all of the RPC implementation functions are in a separate source code file. This is not required, but it is a useful code organization strategy, because most of the development activity will involve the RPC implementations, whereas the server startup code will rarely change.

The functions that we need for RPC handling are defined in the generated code, and so we do not code our own header file to define them. Sometimes when you compile a server, your RPC implementation functions do not match the signatures of the generated functions, or an interface function is not implemented. When you try build the case study servers with a mistake of this type, the compiler generates errors for all of the interface functions *except* for the function that is in error or missing!

If we have done a good job in modularizing our code and identifying common code elements, our RPC implementations should be straightforward. Most of the work will be done in the code modules already written. As an example of an RPC implementation function, let's start with the `orderItem` function. Figure 117 on page 266 shows the TIDL definition for this function.

```
 [nontransactional] RpcReturn orderItem(
[in]  long orderId,
[in]  long productId,
[in,string]  char * productName,
[in]  long orderedQty,
[in]  float productPrice);
```

*Figure 117. TIDL Definition for the orderItem() Function*

Figure 118 on page 266 shows the implementation of the `orderItem` function.

```
 RpcReturn orderItem( long orderId,
long productId,
unsigned char *productName,
long orderedQty,
float productPrice ) {
 RpcReturn temp;

 initRpcReturn( &temp );

 LogEntry(1, INFOMSG, "OrderProcIFMgr::orderItem()");

 transaction {
 addToOrder(orderId, productId, productName,
orderedQty, productPrice, &temp);
 if ( temp.retCode != 0 ) {
 abort("addToOrder abort");
 }
 } onCommit {
 LogEntry(2,DEBUGMSG,"addToOrder transaction committed.");
 } onAbort {
 LogEntry(2,ERRORMSG,"addToOrder transaction aborted.");
 LogEntry(3,ERRORMSG, abortReason());
 temp.retCode = FATALRETURN;
```

*Figure 118. RPC Implementation of the orderItem Function*

In this case, we have already identified and implemented the process of adding an item to an existing order. This was a responsibility of the OrderDB module and was implemented in the `addToOrder()` function. So our implementation of the orderItem RPC simply has to call that function. We begin by initializing the RpcReturn structure, which will be the return value from our function and eventually the return value supplied to the client. We implemented the `initRpcReturn()` function for this purpose, and so we use that common code module here. Because we are accessing the order database as an XA resource, we need to establish a transaction context by placing our call to the `addToOrder()` function in the body of the transaction block. Notice that the symbols *transaction*, *onCommit*, and *onAbort* are used as if they were language elements. These symbols are actually macros that are defined in the Encina header files. The *transaction* macro establishes a transaction context. Any commands within the braces following *transaction* are executed as part of that transaction. When the flow of program execution reaches the closing brace, Encina attempts to commit the transaction. Each of the resources referenced during the transaction must become part of a distributed two-phase commit process. If all of the resources can commit, control will pass to the commands following the onCommit symbol. If any of the resources either explicitly or implicitly abort, control passes to the commands following the onAbort symbol. Explicit aborts occur as a result of a call to the `abort()` function. You can see that we make a call to this function, with a string message as the parameter, if we get a bad return from the `addToOrder()` function. We also could have made the `abort()` function call from within the `addToOrder()` function. The message supplied when the `abort()` function is called can be retrieved through the `abortReason()` function call. We use this function call in `onAbort` processing to provide meaningful error messages in the case of an aborted transaction.

The design approach we are using here calls for the RpcReturn structure to be passed down to the lower-level functions from the RPC handling function. The lower-level functions are expected to fill in appropriate values for the return code, error system, and error code. This approach allows the component that encountered the error to fill in meaningful error results while the calling function need only check the return code. In the event of an abort, we simply make sure the return code is set to the fatal error value and return the RpcReturn structure.

The structure of each RPC implementation function is basically the same. The complete contents of the `OrderProcIFMgr` function is included with the source code for the case study application. You can download the source code from http://www.redbooks.ibm.com "Additional Material".

### 10.2.6.3 Non-RPC Server Processing

In addition to performing processing as a result of RPCs, a server may have to perform activities for other reasons. An example of this in our case study application is the VerificationServer's queue processing. This server is responsible for getting queue items from the verification queue and performing some business on the associated order. We will implement this as a separate thread of control within the server. Thus the VerificationServer will provide an RPC interface as well as maintain an autonomous processing thread for queue processing.

The strategy for implementing the queue processing thread is to create a thread during the server startup processing, before entering the server's RPC listening loop, specifically, in the `serverPostInit()` function. Figure 119 on page 268 shows the VerificationServer's `serverPostInit()` function.

```
int serverPostInit() {
int rc;
LogEntry(2,DEBUGMSG, "Beginning server post-init processing.");
if ( !(rc=initVerifyQueue()) ) {
LogEntry(3,DEBUGMSG, "Server post-init processing:
initVerifyQueue():%d",rc);
return FALSE;
}
if ( !(rc=initGlobalStats()) ) {
LogEntry(3,DEBUGMSG, "Server post-init processing:
initGlobalStats():%d",rc);
return FALSE;
}
if ( !(rc=startVerifyThread()) ) {
LogEntry(3,DEBUGMSG, "Server post-init processing:
startVerifyThread():%d",rc);
return FALSE;
}
LogEntry(2,DEBUGMSG, "End of server post-init processing.");
```

*Figure 119. The VerificationServer's serverPostInit Function*

The `startVerifyThread()` function uses DCE pthread functions to start a new thread to execute the function that monitors the verification queue. The function that executes within the thread is fairly straightforward. Here is an outline of the function:

1. Pause for a short time before doing anything to allow the server to come up fully.

2. Set the normal pause between loop iterations on the basis of the server configuration variable for the thread delay.
3. Attempt to dequeue an item from the queue.
4. If dequeuing is successful, process that item. Read the order from the order database, make a decision on the validity of the order, and place an item on the review queue if the order is not valid.
5. Continue to dequeue items and process them until a dequeue fails because of an empty queue.
6. Sleep for the specified time before checking the verify queue again.

The code for the entire function is a bit lengthy for inclusion here (see the complete code listings available for download on http://www.redbooks.ibm.com), but we show the pertinent sections in Figure 120 on page 270.

```
void * verifyLoop( void * dummy ) {
struct timespec delaytime;

...

LogEntry(1,INFOMSG,"Beginning VerifyLoop() function.");
/* delay initially by 2 minutes after the server starts */
delaytime.tv_sec = 120;
delaytime.tv_nsec = 0;
pthread_delay_np( &delaytime );
/* set the subsequent delays based on the server variable */
delaytime.tv_sec = g_sconf.verThrdDelay;
LogEntry(2,DEBUGMSG,"verifyLoop() delaytime is %d seconds",
delaytime.tv_sec);
indx = 0;
do {
LogEntry(2,DEBUGMSG,"Looping in verifyLoop() function");
pthread_delay_np( &delaytime );
do {
transaction {
if (removeFromVerifyQueue( &qItem, &ret ) ) {
if ( !getOrderInfo( qItem.orderId, &ordInfoPtr, &ret ) )
abort("FailedOrderDbRead");
/* actual verification would go here.
For this example we
simply use a 'round robin' scheme to get
different decisions */

...

if ( !updateCustOrder(&basicOrder, &ret) )
abort("FailedOrderUpdate");
}
else if (ret.retCode != RQS_EMPTY_QSET) {
abort("VerifyDequeueAbort");
}
} onCommit { ...
} onAbort { ...
}
} while ( (qItem.orderId != 0) && (ret.retCode == 0) );
```

*Figure 120. The Queue Processing Thread Function (from VerifyLoop.c)*

This function is structured to execute indefinitely. It will actually end when the thread is stopped during server shutdown.

The thread is created, causing the function in Figure 120 on page 270 to begin execution within the `startVerifyThread()` function shown in Figure 121 on page 271.

```
static pthread_t verThread;

...

int startVerifyThread() {
if (pthread_create( &verThread,
(struct __pt_attr* const*)pthread_attr_default,
verifyLoop,
NULL ))
return FALSE;  /* non-zero is bad return */
else
return TRUE;
```

*Figure 121. Starting the Processing Thread: startVerifyThread() Function*

The thread is stopped during server clean-up by calling the `stopVerifyThread` function shown in Figure 122 on page 271.

```
int stopVerifyThread() {
if (pthread_cancel( verThread ))
return FALSE;  /* non-zero is bad return */
else
return TRUE;
}
```

*Figure 122. Stopping the Processing Thread: stopVerifyThread() Function*

#### 10.2.6.4 Shared Structures within the Server

The VerificationServer code contains an interesting example of how to manage shared structures within a server. The VerificationServer compiles statistics on the results of its verification activities. A client program can request these verification statistics, using the `showStats` RPC function in the VerifiticationIF interface. The statistics are updated during the processing of the verification thread. Thus there is the potential of conflicting access to this structure. We solve this problem by assuring that only one thread at a time accesses the structure. We guarantee this single-threaded access through the use of a mutual exclusion (mutex) structure from the DCE pthread library. Each of the functions that touches the statistics structure must first obtain a mutex lock on the mutex structure associated with the statistics structure.

The statistics structure and the mutex are declared as static variables in the code module containing the statistics structure processing code. We begin by declaring the statistics structure and the mutex as static variables for this code module. These structures are initialized in unison within the `initVerifyStats()` function and cleaned up in the `cleanupGlobalStats()` function (see Figure 123 on page 272).

```
/* global mutex to control access to stats structure */
static pthread_mutex_t g_Stats_mutex;
/* global stats structure */
static VerifyStats g_Stats;

int initGlobalStats() {
g_Stats.numOrdersReviewed = 0;
g_Stats.numOrdersApproved = 0;
g_Stats.numOrdersFailWarning = 0;
g_Stats.numOrdersFailFatal = 0;
if (pthread_mutex_init( &g_Stats_mutex, pthread_mutexattr_default ))
return FALSE;  /* non-zero is bad return */
else
return TRUE;
}
int cleanupGlobalStats() {
if (pthread_mutex_destroy( &g_Stats_mutex ))
return FALSE; /* non-zero is bad return */
else
```

*Figure 123. Initializing the Shared Structure and Mutex (from VerifyStats.c)*

Initializing the structure simply gives the structure's elements their initial values. The mutex is initialized with a call to the `pthread_mutex_init()` function.

Now each of the functions that accesses the statistics structure must include the mutex locking and unlocking requests. When we process an RPC to access the statistics, we will call the `copyStats()` function to get a current copy of the statistics structure. The verification thread function will call `updateStats()` to indicate the results of its processing. Note that we have designed the interface to the statistics structure to be "large grained." Instead of implementing accessor functions for each of the elements, we implement functions that get and set the structure as a whole. This approach is appropriate for a situation where the structure as a whole must be synchronized across accesses from different threads. Figure 124 on page 273 shows the `copyStats()` and `updateStats()` functions.

```
int copyStats( VerifyStats * stats ) {
if (pthread_mutex_lock( &g_Stats_mutex ))
return FALSE;
stats->numOrdersReviewed = g_Stats.numOrdersReviewed;
stats->numOrdersApproved = g_Stats.numOrdersApproved;
stats->numOrdersFailWarning = g_Stats.numOrdersFailWarning;
stats->numOrdersFailFatal = g_Stats.numOrdersFailFatal;
if (pthread_mutex_unlock( &g_Stats_mutex ))
return FALSE;
else
return TRUE;
}
int updateStats( int reviewed, int approved, int warn, int fatal ) {
if (pthread_mutex_lock( &g_Stats_mutex ))
return FALSE;
g_Stats.numOrdersReviewed += reviewed;
g_Stats.numOrdersApproved += approved;
g_Stats.numOrdersFailWarning += warn;
g_Stats.numOrdersFailFatal += fatal;
if (pthread_mutex_unlock( &g_Stats_mutex ))
return FALSE;
```

*Figure 124. Accessing the Shared Structure (from VerifyStats.c)*

The `pthread_mutex_lock` function is called at the beginning of the `copyStats()` and the `updateStats()` functions to get exclusive access to the structure, and the `pthread_mutex_unlock` function is called at the end of each function to free the structure for use by another thread.

The approach shown here is appropriate for structures that are shared within a given processing agent. There are facilities within Encina for creating structures that are shared across all of the processing agents of an application server. Refer to the topics on “monitor shared memory regions” in the *Encina Monitor Programming Guide*. For shared data that spans servers, you have to use an external resource manager such as a database. In this case the resource manager supplies the locking capabilities.

#### 10.2.6.5 Building the Server Executable

Building the server executable program is somewhat platform dependent, but there is a common strategy regardless of the platform. The code modules that need to be compiled as part of the server are the server-side files generated by the TIDL and IDL compilers. For the OrderProcIF interface and the OrderProcServer, these files are:

- From the TIDL compiler: OrderProcIF_manger.c
- From the IDL compiler: _OrderProcIF_sstub.c
- Specific to the OrderProcServer:
  - OrderProcServer.c
  - OrderProcIFMgr.c
- Common Code Modules:
  - ServerConfig.c
  - StatusStrings.c
  - OrderDbDB2.c
  - ProductDbPPC.c
  - RpcReturn.c
  - LogEntry.c

If there are other server specific files, they must be included as well. This is the case for the VerifyLoop.c and the VerifyStats.c files of our VerficationServer.

Specific compiler switches may be required to properly compile the source code for certain platforms and for different DCE implementations. Check Appendix A of *Writing Encina Applications* and the DCE vendor's documentation.

For our build processing, we first compiled the TIDL and IDL generated files into a library called OpsInterfaces and the common code modules into a library called OpsCommon. Compiling the files and modules into libraries facilitates the building of the server, because only the server-specific code is actually compiled as part of the build.

The linkage stage of the build must pull in all of the libraries we have built: the required Encina libraries, the DCE library, and any additional libraries required by other products. The following Encina and DCE libraries are required:

- EncMonServ
- EncServer
- EncClient
- Encina
- DCE

Because we are also accessing RQS and PPC from the OrderProcServer, as well as a DB2 database, we also link with the following libraries:

- EncRqs
- EncPpcExec
- db2

These are not the actual names of the library files, of course. If you are building on the Windows NT platform, the actual files will have the names listed above with an extension of .lib. On UNIX platforms, the files will be prefixed with “lib” and have an extension of .a.

### 10.2.7 Standard Client Construction

In this section we explain how to access the servers from a standard client program written in C.

#### 10.2.7.1 Initializing the Client

Initializing an Encina monitor client is simply a matter of making a call to the `mon_InitClient()` function. The parameters to this function call are a string with a name of your choosing for the client, and the name of the Encina cell to which the program will be a client.

You can establish the name of the Encina cell in various ways. The name can be hard coded, read from the environment, or read from an initialization file. We expect that the ENCINA_TPM_CELL environment variable is set before program execution. We will use the value read from that environment variable as the cell name to which we will connect. Figure 125 on page 276 shows the code from our standard client program that reads the environment variable and initializes the Encina client.

```
char cellName[80];

...

strcpy(cellName, getenv("ENCINA_TPM_CELL"));
if ( strcmp(cellName,"") == 0 ) {
fprintf(stdout,"No value set for ENCINA_TPM_CELL environment variable.
Program Exiting.\n");
return 0;
}

status = mon_InitClient("OrderProc Test Client", cellName);
if ( status != MON_SUCCESS ) {
fprintf(stdout,"Error initializing client for cell name=%s\n",
cellName);
return 0;
}
else {
fprintf(stdout,"Client Initialized to Encina cell name=%s\n",
```

*Figure 125. Initializing an Encina Client (from OrderProcClient.c)*

#### 10.2.7.2 Calling RPCs

Any of the functions defined in an interface can be accessed (assuming the security authorizations allow it) simply by making a call as if the function were within the current program. The first such call that is made to an interface involves some overhead as the actual binding to the server is made at the time of the first RPC request to that interface. To avoid having a user affected by this initial delay, you can make an initial RPC call as part of the client program's startup. The "ping" function that we coded with each of our interfaces is an ideal candidate because it does not require parameters and does not do any meaningful processing on the server. Thus, in our client program we make the RPC call shown in Figure 126 on page 277 immediately after we initialize to the cell.

```
RpcReturn retValue;

...

/***** ping *****/
retValue = pingOrderProcIF();
rpcReturnDisplay(retValue);
```

*Figure 126. Making the Initial RPC Call (from OrderProcClient.c)*

Making the RPC requests that actually implement the business logic is not much more difficult. We have to manage the parameters that are being sent in the call. We use the convention that simple data types are passed by value if they are input only and by single indirect reference if they are input/output parameters. The code for the createOrder() RPC request has a simple data type parameter, the order ID, that is set on the server (see Figure 127 on page 277). In this example, the parameter is specified as [out] in the TIDL. The treatment would be basically the same for an [ingot] parameter except that the variable would be given a value before the call.

```
long orderId;

...

/***** create order *****/
retValue = createOrder(&orderId);
rpcReturnDisplay(retValue);
```

*Figure 127. Calling an RPC with an Output Parameter*

We determined that when we were passing structures as parameters we would use a double indirect reference to the structure to allow for variable size returns where the size is determined on the server. In this case, there is a special treatment of the memory allocation for the structure. Consider the viewOrders RPC that returns a list of BasicOrder structures. We first declare a variable that is a pointer to an OrderList structure. We then pass the address of the pointer as a parameter to the RPC, giving the RPC a double-indirect reference to the structure. We have not made any memory allocation for the structure within our client code. On the server, the `rpc_ss_allocate()` function will be used to allocate the appropriate number of bytes based on the size of the list to be returned. This allocated memory is transferred to the client where a matching allocation is made by the underlying DCE functions. We then manipulate this memory through the pointer variable we declared initially. When we are done, we tell DCE to free the memory that it allocated for us by calling the `rpc_ss_client_free()` function. Figure 128 on page 278 shows the code that performs this processing in the client.

```
OrderList * orderList;

...

/***** view orders *****/
printf("Calling viewOrders() ...\n");
retValue = viewOrders(&orderList);
rpcReturnDisplay(retValue);
printf("There are %d orders in the CustOrder table\n",
orderList->numOrders);
for (indx = 0;indx < orderList->numOrders; indx++) {
printf("   Order id = %d\n", orderList->orders[indx].orderId);
printf("      Order date = %s\n",
orderList->orders[indx].orderDateTime);
printf("      Order status = %s\n",
orderList->orders[indx].orderStatus);
}
rpc_ss_client_free(&orderList);
```

*Figure 128. Using Double Indirection and Server-Side Memory*

That is really about all there is to RPC processing within a client program. The complexity of the client lies mostly in the presentation logic and in managing the user interaction.

### 10.2.7.3 Building the C Client Executable

Building the C client is a matter of compiling the client source with the TIDL and IDL client-side generated files. For the OrderProcClient, these files are:

- OrderProcClient.c
- OrderProcIF_client.c
- OrderProcIF_cswtch.c
- _OrderProcIF_cstub.c

Specific compiler switches may be required to properly compile the source code for certain platforms and for different DCE implementations. Check Appendix A of *Writing Encina Applications* and the DCE vendor's documentation.

Linking the program requires that you include the following libraries:

- EncMonCli
- Encina
- DCE

Of course, if additional products are used in the client, additional libraries may be required.

#### 10.2.7.4 Encina 2.5 and Microsoft Visual C++ with MFC

One final issue that bears mentioning concerns possible conflicts between Encina, DCE, and the Microsoft Visual C++ development environment. Although the clients we are building for the case study application do not use the Microsoft Foundation Classes (MFC), other experiences have shown that care must be taken in how code is managed in this environment. In particular, there are name conflicts between the Microsoft RPC header files and the DCE RPC header files. To compile a program that uses both Microsoft MFC and Encina/DCE, you have to completely separate the code into different files that are compiled separately. We have successfully done this in the past by creating a set of wrapper functions that use structures as parameters that are declared as standard C structures as opposed to IDL structures. The C structures match the IDL structures exactly, and the wrapped functions match the RPCs exactly except for the structure names (the wrapper functions using the non-IDL structures). The wrapper functions simply pass the data and control between the rest of the client code and the RPC functions. A header file is created that declares the wrapper functions and a library is built containing the wrapper functions and the RPC functions. The MFC client code includes the header and links with the library. Calls to the wrapper functions then become calls to the RPC functions, but the compile dependencies between the DCE components and the MFC components are eliminated. This is a messy situation that results from the way the Windows header files are used within the Encina code in Version 2.5. This problem will be resolved in future releases.

### 10.2.8 Web Client Construction

We will implement our Web client using the facilities of the DE-Light Gateway. Specifically we will use the facilities within DE-Light that enable us to produce Java language versions of the stub files similar to those we used to build the C client. A special version of the IDL compiler, *drpcidl*, takes the TIDL source code for an interface and produces Java language stub files. The following command is used to produce the Java stub files for the OrderProcIF interface:

```
drpcidl -Iencina_include_path -Idce_include_path -stub java
OrderProcIF.tidl
```

The drpcidl compiler produces a number of different files. There is a file for each of the structures that are defined in the OrderProcCommon.idl. These files contain the Java code for a class that provides a wrapper for the structure defined in the IDL source. A file is created that defines a Java class for the interface as a whole, providing a member function for each of the functions in the interface.

The structure wrapper classes provide accessor functions for the fields in the structure. For each field there is a generated member function named by prefixing the field name with *get_* and *set_*. For example, for the retCode field of the RpcReturn structure, the functions would be `get_retCode()` and `set_retCode(value)`. If the field is an array, the accessors include an additional parameter for the index value.

The generated class that describes the interface itself has member functions for processing the RPC function calls and accessing the parameters to those functions. The style used here is that each of the parameters specified in the TIDL source for the function has an accessor function in the interface class. Each of the functions in the TIDL source becomes a member function of the interface class. However, the interface class function has no parameters. To actually call one of the RPC functions, you have to use the parameter accessor functions to set values of the input parameters, call the RPC function, and use the accessors for the output parameters to get the returned data.

Here is the flow of processing in the Java program that uses these generated classes:

1. Establish a connection to the DE-Light gateway by instantiating a DrpcConnection object and specifying the gateway to which you want to connect.
2. Instantiate an object from the generated interface class and give it the connection object created above.
3. Call the RPCs by setting the input parameters, calling the interface object’s RPC function, and accessing the return parameters.

For our case study Java applet, we structured the code such that all of the RPC-related code is contained in a class named RpcHandler. We get the DE-Light Gateway information from the parameter tags within the applet tag on the HTML page. Figure 129 on page 281 shows the function from the RpcHandler class that gets the gateway information.

```
private static String getGatewayName() {
String name = (
RPCHandler.getContext().getParameter("protocol") + ":" +
RPCHandler.getContext().getParameter("host") + "[" +
RPCHandler.getContext().getParameter("port") + "]");
return name;
```

*Figure 129. Getting the Gateway Information from the HTML Page*

The connection is established through another member function of the RpcHandler class which uses the `getGatewayName()` function (see Figure 130 on page 281).

```
private static boolean getDrpcConnection() {
boolean stat = true;
if (RPCHandler.c_drpc == null) {
try {
RPCHandler.c_drpc = new
DrpcConnection(RPCHandler.getGatewayName());
RPCHandler.c_drpc.setDceSecurityLevel(
DrpcConnection.DRPC_DCE_PROTECT_NONE);
RPCHandler.c_drpc.setSecurity(DrpcConnection.SEC_NONE);
} // end-try
catch (final DrpcException e) {
stat = false;
RPCHandler.c_drpc = null;
RPCHandler.getContext().showStatus(
"RPCHandler::getDrpcConnection() DrpcException: " +
e.getMessage());
} // end-catch
} // end-if
```

*Figure 130. Establishing the Connection to the DE-Light Gateway*

The next step is to instantiate the interface object and set its connection object. This is shown in Figure 131 on page 282.

```
private boolean getOrderProcIF() {
boolean stat = RPCHandler.getDrpcConnection();
if (stat) {
if (this.m_stub == null) {
this.m_stub = new OrderProcIF();
this.m_stub.setHandle(RPCHandler.c_drpc);
//Server name must not be specified for RPCs to
// Encina Monitor servers
// the following line is for non-monitor servers
//this.m_stub.setServer(RPCHandler.getServerName());
} // end-if
} // end-if
return stat;
```

*Figure 131. Instantiating the Interface Object*

Now we can make the calls required to issue an RPC through the gateway, using the `reviewOrder` function. Figure 132 on page 283 shows the code for calling this function.

```
private boolean reviewOrder() {
boolean stat = false;
if (this.m_orderChanged) {
if (this.getOrderProcIF()) {
try {
// set the input parameter
this.m_stub.set_reviewOrder_orderId(
this.m_orderInfo.get_orderId());
// call the RPC function
OrderProcIF_RpcReturn ret = this.m_stub.reviewOrder();
if (ret.get_retCode() == 0) {
// get the output parameters
this.m_orderInfo =
this.m_stub.get_reviewOrder_ordInfo();
this.m_orderChanged = false;
stat = true;
} // end-if
else {
RPCHandler.getContext().showStatus(
"RPCHandler::reviewOrder() failed");
} // end-else
} // end-try
catch (final DrpcException e) {
RPCHandler.getContext().showStatus(
"RPCHandler::reviewOrder() DrpcException: " +
e.getMessage());
} // end-catch
catch (final Exception e) {
RPCHandler.getContext().showStatus(
"RPCHandler::reviewOrder() Exception: " +
e.getMessage());
} // end-catch
} // end-if
} // end-if
else {
```

*Figure 132. Calling the reviewOrder() RPC Function*

# Part 4. Maintenance

A staggering number of applications are developed with little or no focus on their postdevelopment life. We believe that development teams should always address three major issues from the very beginning of their project -- application deployment, system administration, and troubleshooting. Proper planning for these phases of the project life cycle sharply decreases the risks involved in deploying and operating the application; hence, fewer people and monies are burned, and more customers are satisfied.

Encina is no exception when it comes to life after development. Arguably Encina systems are even more vulnerable to problems in the deployment and operational phases because of the heterogeneous and highly distributed nature of the Encina applications.

In Part 4 we discuss various aspects of administering, deploying, and troubleshooting Encina applications. Our focus is not so much on procedures and syntax but on the concepts to consider when designing the application and planning for its rollout.

# Chapter 11. Administration

The term “administration” is used for a wide variety of activities related to support of an application in production. In this chapter we elaborate on a range of topics that could loosely be considered as part of system administration. Our focus is on presenting the issues and providing general guidelines for dealing with them rather than spelling out rigorous procedures for monitoring and analyzing problems. For more details on resolving Encina application problems, see Chapter 13, “Troubleshooting” on page 347.

## 11.1 Naming Conventions

It is easy to overlook the issue of choosing and sticking to a sound naming convention. There are more than plenty of reasons to enforce a sound set of naming conventions, but we just mention few of them here. Following a good naming convention makes your system administrator’s life much easier when adding new objects to the system. It reduces unnecessary confusion between the team members, and it makes it a lot easier to maintain knowledge about your application.

Encina systems generally contain a large number of named objects, and a sound naming convention is a definite must. When putting together the first draft of your naming convention, think about these issues:

- Encina naming is an integral part of your overall naming scheme.
- Shorter names are easier to type and display.
- Names should facilitate script writing.
- Naming conventions should be easy to follow by humans too.
- Naming conventions must allow for explosive future growth.

Now, given all that good advice, let us try to put together a naming convention for our case study application.

Here are the rules we use:

- Each Encina cell name is of the form /.:/<application>/enc_<cell type>, where <application> is the name of the application, and <cell type> describes what the cell is used for, prod (production), stage (staging), release (official release).
- All Encina servers are named <server>; the first letter of each word is in upper case.

- All backup servers are named <server>B.
- The RQS, SFS, PPC, and other special servers are named <server>Rqs, <server>Sfs, <server>Ppc, and so forth.
- The interfaces are named <interface>IF, the first letter of each word is in upper case.
- The default Encina values are used for all user names, directories, and CDS names.

This convention is quite simple. If your organization has different departments managing different Encina applications, you may want to add an extra layer in your Encina cell names, for example, /.:/orders/enc_prod could become /.:/operations/orders/enc_prod.

Table 8 on page 290 presents the names of the objects we use for our case study application.

*Table 8. Case Study Application Encina Objects*

| | Encina Object | CDS Name |
|---|---|---|
| Cell | cell | /.:/orders/enc_prod |
| Node | prod_one | /.:/orders/enc_prod/node/prod_one |
| | prod_two | /.:/orders/enc_prod/node/prod_two |
| Server | OrderProcServer | /.:/orders/enc_prod/server/OrderProcServer |
| | VerificationServer | /.:/orders/enc_prod/server/OrderProcServe |
| | OrderProcServerB | /.:/orders/enc_prod/server/OrderProcServerB |
| | VerificationServerB | /.:/orders/enc_prod/server/OrderProcServerB |
| Interface | OrderProcIF | /.:/orders/enc_prod/ecm/interface/OrderProcIF |
| | VerificationIF | /.:/orders/enc_prod/ecm/interface/VerificationIF |
| RQS server | ordersRqs | /.:/orders/enc_prod/server/ordersRqs |
| PPC server | ordersPpc | /.:/orders/enc_prod/server/ordersPpc |
| DE-Light gateway | ordersGtw | /.:/orders/enc_prod/server/ordersGtw |

The application also uses two RQS queues and two RQS queue sets. The names of the queues are PendingVerifyQueue and ReviewVerifyQueue. The two queue sets are named PendingVerifyQueueSet and ReviewVerifyQueueSet.

The CDS names for these fours objects are, respectively:

```
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueue
/.:/orders/enc_prod/server/ordersRqs/queue/ReviewVerifyQueue
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueueSet
/.:/orders/enc_prod/server/ordersRqs/queue/ReviewVerifyQueueSet
```

## 11.2 System Security and User Administration

In this era of increased system openness and tight Internet integration, you can never spend enough effort on securing your system against unauthorized access. From putting together a simple security procedure to inviting a full-blown external security audit, it all helps. The trick is to realize that it is an ongoing process and you should plan to dedicate sufficient resources to security for as long as you intend to have your system operational.

For a nondistributed application, the operating system can be trusted to protect resources from unauthorized access. This is not the case in open distributed systems, however. Communications take place over an accessible network, where messages between machines can be observed or forged. An additional security system is required to control access to resources in a distributed environment.

### 11.2.1 Encina Security Model

One of the major advantages of using Encina as your distributed transaction and process monitor system is its strong security model based on the DCE Security Service.

The DCE Security Service ensures that client processes can securely access server processes and that servers can securely pass information to clients based on their identity. Therefore each two participants in a communication (called *principals*) can authenticate their counterparts and exchange information in a secure fashion.

In addition to authentication and secure information exchange, Encina uses Access Control Lists (ACLs) which allow you to specify which services are available to which users. Therefore, each Encina application maintains three levels of security -- operating system security, DCE security, Encina server security. The operating system security facilities are used to allow users to access the application machines. DCE security is used to allow users to authenticate themselves to Encina servers across the entire distributed environment. Finally, each Encina server is configured to provide a different set of interfaces to the different authenticated users and groups.

### 11.2.2 Operating System Security

Before we explain which operating system users and groups you need to create for Encina, we want to emphasize the importance of maintaining proper operating system security, especially on the machines running the DCE Security Service. Any compromise of security on your DCE security servers automatically compromises the security of your entire Encina application. Therefore, we advise you to dedicate a physically secure machine as your DCE security server and ensure that no external access is granted to anyone on that machine except for clients talking to the DCE Security Service.

In most cases you need only two user accounts to run Encina applications on your machine: the superuser account and the Encina account. The superuser account is needed to install, configure, and run DCE (see Chapter 12.3, “DCE and Encina Installation and Configuration” on page 324). You have to use the superuser account because DCE provides the underlying infrastructure to Encina, including security.

The superuser account is also used to configure the disk space required by Encina and install the Encina software. Installing Encina as the superuser ensures that no one else can alter any Encina files: executables, scripts, and configuration files.

Once you have DCE up and running and Encina installed, do not use the superuser account . Configuring the Encina cell and running the Encina servers is done through the Encina account.

The Encina account is maintained by the Encina administrator, who may have access to the superuser account. Nevertheless, a separate account should be used for running and managing Encina to ensure that the Encina application does not interfere with the operating system in any distracting way.

The Encina account belongs to an Encina administration group. Other members of this group are the operators. You have to decide whether you need to separate the tasks of the operators from those of the administrators.

### 11.2.3 DCE Security

After you install and configure DCE (see Chapter 12.3, “DCE and Encina Installation and Configuration” on page 324), only one DCE account is in place, the DCE cell administrator. We recommend that you stick to the default name for this account, which is cell_admin.

Encina is designed to handle some of the DCE administration required for its proper functioning. You have to perform a simple initial DCE setup before you create the Encina cell. Part of this initial setup is creating the Encina administrator user and the Encina administration group. This initial DCE setup is performed for you by enconsole when you create the Encina cell. Once DCE has been set up for Encina, you start using the Encina administrator account for an Encina cell configuration and maintenance.

#### 11.2.3.1 DCE Setup for Encina Servers

When you configure the Encina Cell Manager, the Node Managers, and the application servers, Encina automatically creates the required DCE users and DCE principals. A user is created for every server and manager. A random DCE password is also generated for each user. DCE keeps the information about users and their passwords in its security registry.

When an Encina server or manager is started, it has to provide its password to DCE security for authentication. Therefore, the server has to “know” its password. When Encina creates a server, it stores the generated password in a file that resides on the machine running the server. This file is readable only by the Encina user. It is called a keytab file and it is located in the server working directory. Encina periodically changes the password of each of its servers for additional protection. The keytab file is updated when the password is changed.

The CDS names used for the Encina server principals can be configured manually as part of the server configuration (see Chapter 12.3.6, “Encina Server Configuration” on page 342). We recommend that you keep the default convention; the name of the Encina server is used to generate the CDS name for its principal:

```
/.:/<Encina cell name>/server/<server name>
```

The Cell Manager and Node Managers use DCE principals named, respectively:

```
/.:/<Encina cell name>/ecm
/.:/<Encina cell name>/node/<node name>
```

For example, our Encina cell /.:/orders/enc_prod uses these principals for the Cell Manager, Node Managers, RQS server, and PPC gateway:

```
/.:/orders/enc_prod/ecm
/.:/orders/enc_prod/node/prod_one
```

```
/.:/orders/enc_prod/node/prod_two
/.:/orders/enc_prod/server/ordersRqs
/.:/orders/enc_prod/server/ordersPpc
```

Any objects managed by an Encina server contain the CDS name of the server as a prefix, for example, the RQS PendingVerifyQueue uses this CDS name:

```
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueue
```

#### 11.2.3.2 DCE Setup for Encina Clients

Encina automatically creates DCE users and DCE passwords for the Encina servers, but you have to manage the DCE accounts and DCE passwords for the Encina client applications. Therefore you have to create DCE accounts for all users allowed to use the application. The users would then DCE login (authenticate themselves to DCE) and run the client program that talks to the Encina servers. These servers support different interfaces, and the interfaces must be configured to allow users to access them. For example, when you create the server OrderProcServer and add the OrderProcIF interface to it, the ACL for the interface is automatically set to:

```
enccp -c acl show /.:/orders/enc_prod/ecm/interface/OrderProcIF
{group encina_admin_group x}
{group encina_servers_group x}
```

As you can see, all Encina servers have access to the interface; however, clients are not allowed to use it. You have to manage the interface ACLs to allow client applications to access the interfaces.

Instead of managing users on an individual level, you should create a few DCE groups to reflect the different functions available to your users. For example, our case study sample application provides two types of functions: requests for ordering products and requests for processing the orders. The first group of requests is supported by the OrderProcIF interface, and the second group, by the VerificationIF interface. Users who have access to the OrderProcIF are not necessarily allowed to access the other interface. Therefore, we create two DCE groups (as the cell_admin DCE user) to model the two types of users we have identified, the ops_customer group and the ops_order group:

```
enccp -c create group ops_customer
enccp -c create group ops_order
```

We now modify the interface ACLs (as the encina_admin DCE user ) to allow users from their corresponding groups to access the interfaces:

```
enccp -c acl modify /.:/orders/enc_prod/ecm/interface/OrderProcIF \
-add {group ops_customer x}

enccp -c acl modify /.:/orders/enc_prod/ecm/interface/VerificationIF \
-add {group ops_order x}
```

Every Encina interface is configured to allow access from the groups it serves and nobody else. Every new user you add to the system is added to its corresponding DCE groups. In this way you do not have to modify the Encina server ACLs every time you add a new user. For example, a new customer is introduced to the system, and you create (as cell_admin) a DCE account for it, using enccp or dcecp:

```
enccp> principal create jsmith
enccp> group add -member jsmith
enccp> org add none -member jsmith
enccp> account create jsmith \
>     -group ops_customer -password tempPass -organization none
```

DCE user jsmith is now ready to access the OrderProcIF without further DCE setup because jsmith belong to the ops_customer group, which is allowed to access the interface.

## 11.2.4 Encina Server Security

Although Encina sets up the correct ACLs for all its objects, it cannot anticipate the entire ACL setup, and you have to modify the default ACLs as you configure your servers and their objects. For example, when you add a new server, you may want to restrict access to its interfaces according to client identity. When you add an RQS queue, you have to decide who can access it and change its ACL to reflect your setup.

### 11.2.4.1 Monitor Application Server (MAS) Interfaces

The default ACL for each interface exported by an Encina Monitor Application Server (MAS) allows only the Encina administrator and the Encina operator to access the interface. You have to establish which clients and other servers need access to the interface and modify its ACLs. For example, the OrderProcIF is created with the following default ACL:

```
enccp> acl show /.:/orders/enc_prod/ecm/interface/OrderProcIF
{group encina_admin_group x}
{group encina_servers_group x}
```

Therefore none of our users will have access to the interface unless we modify the ACL. Since we want the OPS customers to be able to access the interface, we modify the ACL to allow the ops_customer group to execute the interface:

```
enccp> acl modify /.:/orders/enc_prod/ecm/interface/OrderProcIF
          -add {group ops_customer x}

enccp> acl show /.:/orders/enc_prod/ecm/interface/OrderProcIF
{group encina_admin_group x}
{group encina_servers_group x}
{group ops_customer x}
```

### 11.2.4.2  RQS and SFS

The RQS server itself has an ACL that determines which servers can access the RQS server. You have to modify this ACL to allow other servers to access it. You also have to find out which RQS queues and queue sets are to be accessed by which servers and set their corresponding ACLs. For example, our RQS server, ordersRqs, is set up in this way:

```
enccp -c acl modify /.:/orders/enc_prod/server/ordersRqs \
       -add {group encina_servers_group ---tq}
```

As a result of modifying the RQS server ACL, all Encina servers can now access it:

```
enccp -c acl show /.:/orders/enc_prod/server/ordersRqs
{unauthenticated -----}
{group encina_admin_group caxtq}
{group encina_operator_group ----q}
{group encina_servers_group ---tq}
{any_other -----}
```

In addition to the server ACL, we modify the queue and queue set ACLs by adding the following entry to each of them:

```
enccp> acl modify \
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueue \
       -add {group encina_servers_group ladoxnemrpq}
enccp> acl modify \
/.:/orders/enc_prod/server/ordersRqs/queue/ReviewVerifyQueue \
       -add {group encina_servers_group ladoxnemrpq}
enccp> acl modify \
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueueSet \
       -add {group encina_servers_group ladoxnemrpq}
enccp> acl modify \
/.:/orders/enc_prod/server/ordersRqs/queue/ReviewVerifyQueueSet \
       -add {group encina_servers_group ladoxnemrpq}
```

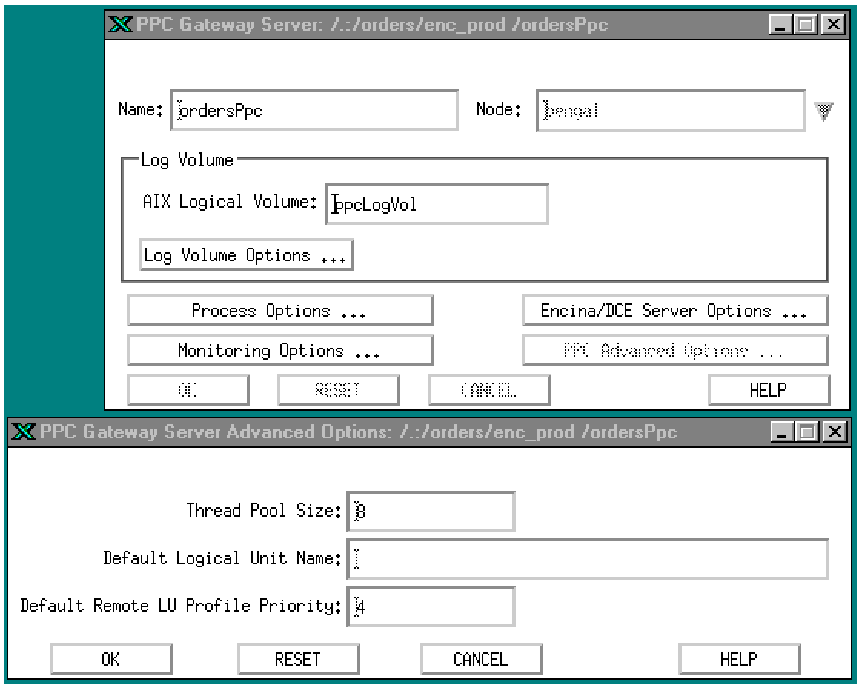
Notice that in this example we take the easy way out and just allow all types of access for all servers to all queues and queue sets. You can be more restrictive by determining exactly which queues should be accessed by which servers and grant the ACLs accordingly.

Similarly, you have to modify the ACL of your SFS server and the ACLs of all its objects to allow proper access to them.

For complete information about the Encina ACLs, see Chapter 7, “Controlling Access to Encina Resources,” in Transarc’s *Encina Administration Guide Volume Two: Basic Administration*.

### 11.2.4.3 PPC

The PPC gateway requires some extra security configuration to deal with the SNA access to a mainframe. Figure 133 on page 297 shows the two extra parameters you have to specify when configuring the gateway: the Default Logical Unit Name, and the Default Remote LU Profile Priority. Consult with your mainframe personnel to find out which LU has been assigned to your application and which profile priority you should use.


PPC Gateway Server: /.:/orders/enc_prod /ordersPpc

Name: ordersPpc
Node: bengal

Log Volume
AIX Logical Volume: ppcLogVol

PPC Gateway Server Advanced Options: /.:/orders/enc_prod /ordersPpc

Thread Pool Size: 8
Default Logical Unit Name:
Default Remote LU Profile Priority: 4

*Figure 133. PPC Gateway Definition Window*

### 11.2.5 DE-Light Clients and Gateways

Any DE-Light gateways you might have in your system provide two levels of security: between the gateway and the DE-Light client, and between the gateway and the Encina application. The Secure Sockets Layer (SSL) is used to ensure the security of the connection between the DE-Light clients and the gateway. DCE security is used to authenticate the gateway to the Encina servers it talks to. Therefore, from an Encina standpoint, the DE-Light gateway is considered a client to all other servers and its principal should appear in the user groups already set up for the application. For example, our DE-Light gateway uses the DCE principal /.:/orders/enc_prod/server/ordersGwy. We have a group called ops_customer, which contains all DCE users allowed to send customer requests. Therefore, we add the ordersGwy principal to the ops_customer group. This activity must be performed by the DCE cell administrator, unless another user has granted encina_admin privileges to modify this group:

```
enccp -c group add ops_customer -member \
       /.:/orders/enc_prod/server/ordersGwy
```

As a result of this operation the DE-Light gateway has the same privileges as any other ops_customer user and can perform requests on behalf of those customers.

For more information about the DE-Light security mechanisms see Chapter 7.5, “Java Client Security” on page 167.

### 11.2.6 Encina++ and CORBA

The Encina++ components are offered in two flavors, DCE and CORBA. The DCE implementation of Encina++ relies on DCE for binding and security. The CORBA implementation of Encina supports the development of transactional, object-oriented applications for the CORBA environment. Encina++ /CORBA applications rely on an object request broker (ORB) for communication between clients and servers. This dependency on an ORB affects interface definition, binding, and exception handling. Encina++ completely relies on the Orbix ORB to provide authentication between clients and servers.

## 11.3 Encina System Monitoring

Encina applications consist of a potentially huge number of components. Each of these components is defined by a variety of parameters that you need to monitor. Chapter 13, “Troubleshooting” on page 347 describes in detail how you can find out what is happening with your Encina application. In particular, it provides a roadmap to all message log files generated by Encina.

Although Encina takes care of its own servers, you should ensure that the encinaNanny processes are always running. They in turn ensure that the Encina Cell Manager and the Encina Node Managers are always running.

You should constantly monitor the enconsole View-->Serious Messages window (also found in the file ecml.log), where all unexpected events encountered by Encina are recorded.

Several components are of specific interest to monitoring. The RQS queues, for instance, should never be full. Use the rqsadmin tool to find out the status of a particular queue:

```
rqsadmin query queue -server <RQS server name> <queue name>
```

Disk storage space should also be closely monitored. First of all, you have to ensure that there is always enough space for the Encina run-time files, such as the message log files. We recommend that you periodically offload the server.out files for all servers and restart the servers with an empty server.out file, to keep track of the messages without filling in the disk.

In addition, you have to monitor the Encina volumes. They also should never be full. If you need more space for any Encina volume, you can enlarge it. For details on managing Encina volumes, see sections 2 and 3 of Transarc's *Encina Administration Guide Volume Two: Basic Administration* .

We recommend that you carefully consider which Encina parameters you need to monitor and then put together a set of scripts to perform the checks for you. You can run these scripts on a regular basis and view their output.

## 11.4 Fault Tolerance and Encina

One of the most important components of any application design is the ability of the application system to handle failures. Encina applications consist of many different components and run on different machines possibly linked through different networks. Thus, proper Encina application design geared toward fault tolerance is a definite must.

Encina provides several mechanisms to deal with a variety of system and application failures. In this section we discuss automatic server restart, interface redundancy, and Encina volume mirrors and show how you can apply them to design and configure a robust, fault-tolerant Encina application.

### 11.4.1 Automatic Server Restart

The Node Managers control the Encina Servers. They ensure that the servers are running at all times. On server failure, the Node Manager restarts the server up to a specified number of times. Therefore, any software or hardware failure that causes a server to fail is overcome by Encina.

The only Encina processes that are not managed by the Encina Node Managers are the node managers themselves and the Cell Manager. Those processes are looked after by the *encinaNanny*, which is spawned when you start Encina from the rc.encina script (see Chapter 12.3.5, “Automatic Restart Setup” on page 331 for more details on this script). We also recommend that you use some automatic means of monitoring the encinaNanny processes as well. Be aware that should the encinaNanny terminate, no immediate harm is done to the system; however, any consequent failure of a Node or Cell Manager goes unnoticed.

### 11.4.2 Multiple Server Instances

All Encina interfaces can be served by more than one Encina server. A properly written client, as discussed in Chapter 10.2.7, “Standard Client Construction” on page 275, attempts to bind to a server and, should it fail, it tries to reach another server that provides the same interface. This mechanism not only increases the fault tolerance of your application but also provides for higher availability and better load balancing.

### 11.4.3 Encina Volume Mirrors

On the AIX operating system, you mirror an AIX logical volume by using an AIX utility (you cannot use Encina). On other systems, you can mirror data by using either an operating system facility or Encina's Volume Service. The Windows NT operating system provides mirroring by using partitions in mirrored sets.

Also, some machines are equipped with special hardware that transparently replicates data. Before choosing Windows NT mirrored sets or hardware-based replication in place of Encina replication, you should understand the recoverability issues described below.

There are several factors to consider when selecting mirroring policies and mechanisms. First, recall the following typical properties of disks:

- They are permanent, except in the event of a failure. Mirroring, backups, and logging are meant to protect against failures. In the case of Encina, logging is also involved in providing transactional guarantees and can be used to bring backups up to date so that nothing is lost between the time of the last backup and the time of a failure.

- They store very large amounts of data and therefore rely on secondary storage (typically disk-shaped magnetic media).
- They are slow (compared to primary storage). In particular, they often must wait for an internal component (the head) and the media they use to physically move so that the head is over that very small portion of the media involved in each individual data transfer. This operation typically involves moving the head radially in or out and then waiting for rotation to bring the proper portion of the spinning media under the head. This is a high-latency operation and reduces the potential throughput for the entire disk.
- They store data in pages (4 KB). Each page includes extra information to ensure that it either contains consistent data or is detectably bad and contains no usable data. The page is thus an atomic unit of data storage. An entire page is either successfully stored with the extra information or it is not and is reported as bad when a read is attempted.

Mirroring introduces additional complexities. When data is mirrored, writing to all copies of the data at once is not desirable. Simultaneous writes to disk greatly increase the likelihood of multiple copies becoming bad should a failure occur while the data is being written.

Besides the risk of bad copies, mirroring introduces the possibility of unmatched copies. Comparing each copy for every read is expensive. The typical way to deal with this problem is to find (and correct) any inconsistencies when a system is brought back up after a failure. When the amount of data is large and the importance of bringing the system up quickly is high, a simple scheme such as scanning each page breaks down. Encina implements a sophisticated logging scheme that addresses this and other issues.

Another potential problem can occur when a write that needs to be atomic spans multiple pages, and only some of those pages get written to a copy of the data. This problem is addressed by Encina and may not be by other mirroring schemes.

These considerations generally apply to both operating-system and hardware-based data mirroring for two or more copies of the data. In summary, Encina mirroring is likely to be the most reliable approach. The performance difference between Encina and hardware-based schemes is potentially significant. However, before choosing a hardware-based scheme, be sure that it provides the required reliability. The performance difference between Encina and operating system schemes may favor Encina in cases where both the operating system and Encina are logging data; if the operating system is not logging data, the reliability of its mirroring is reduced, but the performance may be slightly higher.

We recommend that on your production machines you use mirrored disks. In addition, Encina provides an extra mechanism for protecting your Encina volumes. Encina allows you to use two raw disk partitions for an Encina volume. You can add a mirror to a volume when the server is being defined or when it is running. The two raw partitions are managed as a mirrored pair. Therefore any updates to the primary Encina volume partition are made to its mirror as well. Should one of the paired volumes become unavailable, Encina switches automatically to the remaining partition, and an error message appears in the Encina message log files (see Section 13.3, “Encina Message Log Files” on page 351). Thus, you have to place the two partitions of a mirrored Encina volume on two separate physical disks to benefit from the Encina mirroring scheme.

You do not have to use disk mirroring and Encina volume mirroring at the same time. A disk mirroring system that provides the necessary level of integrity on failure is recommended because it usually provides better performance and the mirrored disks can be used to store the rest of the application (DCE, Encina, executables, configuration files, Encina and DCE run-time files), which further increases the system fault tolerance. If the mirroring software you have does not meet all the criteria listed at the beginning of this section, we recommend that you use Encina mirroring.

### 11.4.4 Examples of Failures

Let us go back to our example and find out what happens during a single component failure. Figure 143 on page 323 shows the production configuration of our ordering system. We focus only on failures on machines running the Encina components. Dealing with failures on any other machines is beyond the scope of this book.

#### 11.4.4.1 CPU Failure Recovery

As far as any Encina cell is concerned, there are two categories of CPU failures. A CPU failure of the Encina Cell Manager machine leads to unavailability of all Encina servers on that machine, including the Encina Cell Manager. Any transactions in progress during the failure are aborted and then rolled back.

The Encina Cell Manager itself is used for server management as well as the ACL manager for all Encina objects. Hence, when the Cell Manager is down, clients cannot establish new bindings with Encina servers. However, the existing connections to servers on other CPUs are not be affected. A properly written client application binds to the server once and then uses the handle to maintain the communication.

A CPU failure on a machine that does not run the Encina Cell Manager does not affect the Encina Cell Manager. Therefore server and ACL management through Encina is still possible. The interfaces provided by the application servers on the failed CPU are also supported by the servers running on the other CPU, and the Encina client applications are written in a way that allows them to attempt to reconnect to a server. Therefore, all clients using the servers upon failure will reconnect to the servers on the remaining CPU providing the same interfaces. Any transactions in progress during the failure are aborted and then rolled back. Hence, the application continues to operate after a short period of failover time.

#### 11.4.4.2 Communication Failure Recovery

Any communication failure results in machine unavailability. As far as a client application is concerned, the communication failure means that a machine is not available. Therefore, we treat the communication failures as CPU failures.

To reduce the risk of machine unavailability, we recommend using two physical communication connections on each machine, for example, two Ethernet cards with two IP addresses, so that if there is failure on one of the channels, you have another communication link to the machine.

Encina automatically handles the two links (IP addresses). When an Encina server is started, it registers its application interfaces with DCE. If the machine happens to serve two IP addresses, the application interfaces are advertised on both IP addresses. Should one of these IP addresses become unavailable, clients can still access the server providing the advertised application interface through the second IP address. Again, your client application has to be written such that it can rebind to an application interface. This ensures that when the connection to an Encina server is lost due to a communication failure the client attempts to restablish it. Similarly to CPU failures, a communication glitch does not affect the atomicity of the transactions in progress. When the failure occurs, they are all aborted and rolled back.

#### 11.4.4.3 Disk Failure Recovery

The best guard against disk failures is to use mirrored disks. The recovery procedure then depends on the particular setup you have. In addition, Encina provides volume partition mirroring and restart file mirroring. For both mechanisms Encina automatically detects a failure and picks up the remaining undamaged partition. An error message is logged, and you can restore the failed partition later on. As soon as the failed partition is restored and ready for use, you can add it as a mirror to the remaining running partition of your Encina volume. See Figure 144 on page 326 for a definition of Encina mirror volumes.

### 11.4.5 Volume Backup and Recovery

Although the DCE and Encina executables, configuration files, and run-time files are backed up as part of the regular backup procedure you establish for your machines, Encina volumes require special care.

Encina volumes run on raw disk partitions (logical volumes on AIX), and their format is different from the format of the files supported by your file system. Encina volumes contain either data (such as the data volumes of the Encina Cell Manager, RQS, or SFS) or data logs (such as the log volumes used by various Encina components). You must back up all these volumes in order to recover after a serious system failure that cannot be handled by any of the mechanisms discussed in this chapter. Encina volumes cannot be backed up by the operating system backup utilities because of their different format. You have to use Encina to back up the volumes.

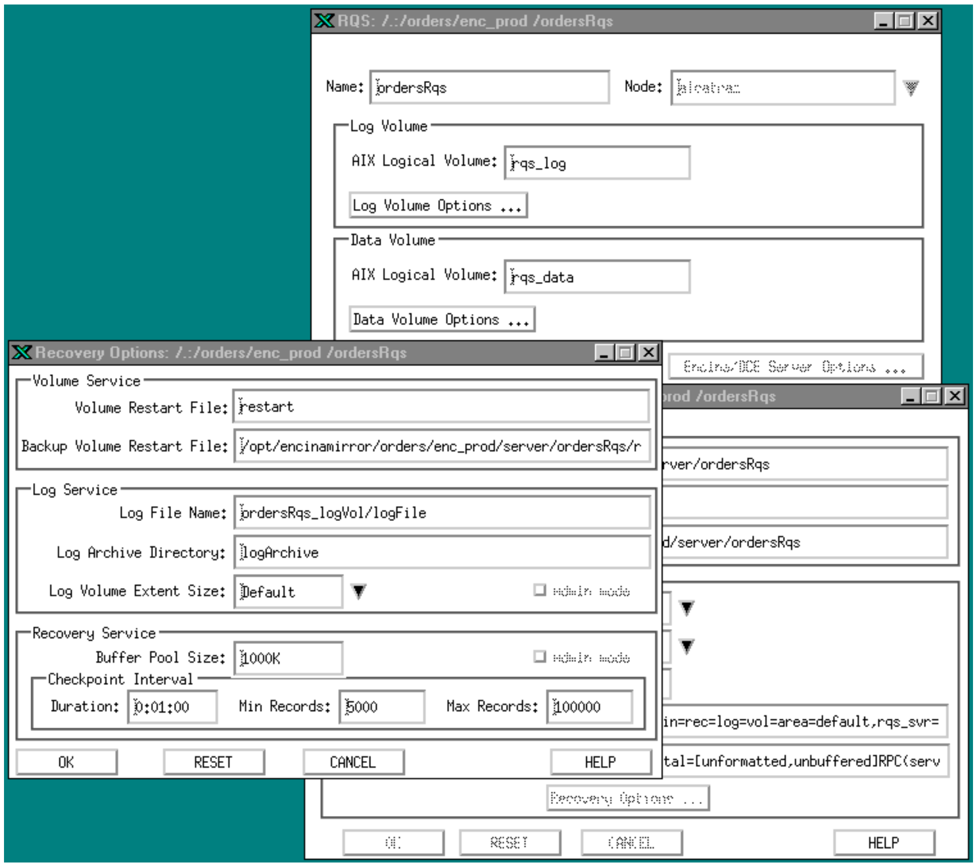
The Encina backup facility allows you to create two types of backup files: log archive files and backup files. Log archive files back up log data in a server's log volume. They are automatically generated when media archiving is enabled for a given server (Figure 134 on page 305). Log archive files are stored in the server's working directory by default. You can modify the location of these files through the Encina/DCE Options->Recovery Options Window for your server (Figure 135 on page 305). For our example, the log archive files for the RQS server ordersRqs reside in this directory:

```
/opt/encinalocal/orders/enc_prod/server/ordersRqs/logArchive
```

You can move the log archive files to offline data storage, using the regular system backup procedure that is in place for a particular machine.

```
$ tkadmin query mediaarchiving -server
/.:/orders/enc_prod/server/ordersRqs
Media archiving is disabled.
$ tkadmin enable mediaarchiving -server
/.:/orders/enc_prod/server/ordersRqs
$ tkadmin query mediaarchiving -server
/.:/orders/enc_prod/server/ordersRqs
Media archiving is enabled.
```

*Figure 134. Enabling Media Archiving*



RQS: /.:/orders/enc_prod /ordersRqs

Name: ordersRqs
Node: alcatraz

Log Volume
AIX Logical Volume: rqs_log
Log Volume Options ...

Data Volume
AIX Logical Volume: rqs_data
Data Volume Options ...

Encina/DCE Server Options ...

Recovery Options: /.:/orders/enc_prod /ordersRqs

Volume Service
Volume Restart File: restart
Backup Volume Restart File: /opt/encinamirror/orders/enc_prod/server/ordersRqs/r

Log Service
Log File Name: ordersRqs_logVol/logFile
Log Archive Directory: logArchive
Log Volume Extent Size: Default
admin mode

Recovery Service
Buffer Pool Size: 1000K
admin mode
Checkpoint Interval
Duration: 0:01:00
Min Records: 5000
Max Records: 100000

*Figure 135. Recovery Options Window*

The backup files back up application data in a server’s data volume. A complete backup (consisting of one or more backup files) covers an entire volume. You must manually create the backup files, using the tkadmin backup lvol command. You can create a script that backs up all Encina data volumes on a regular basis and stores the resulting backup files on the disk. Then you have to move the backup files offline, using the regular system backup procedure.

The backup files along with the corresponding log archive files are required to restore a failed Encina data volume. To bring the data volume up to its latest consistent state, you also need the log volume associated with the failed data volume. For detailed procedures on Encina backup and restoration, see Chapter 5, “Performing Backups,” of the *Encina Administration Guide Volume Two: Basic Administration*.

### 11.4.6 Robust Fault-Tolerant Configurations

As you can certainly appreciate by now, designing and configuring a fault tolerant system is not as simple as it may initially appear. Many components can fail, and you have to think of the recovery needed after different types of failures occur.

Our sample ordering system does not provide the fullest fault tolerance available for an Encina application. The database server has no backup, and the communication links are not duplicated. In addition, a failure of the machine running the Encina Cell Manager will jeopardize any Encina server management.

Figure 136 on page 307shows a more robust configuration that can provide a 10 to 15 minute failover time for any single component failure of the system.

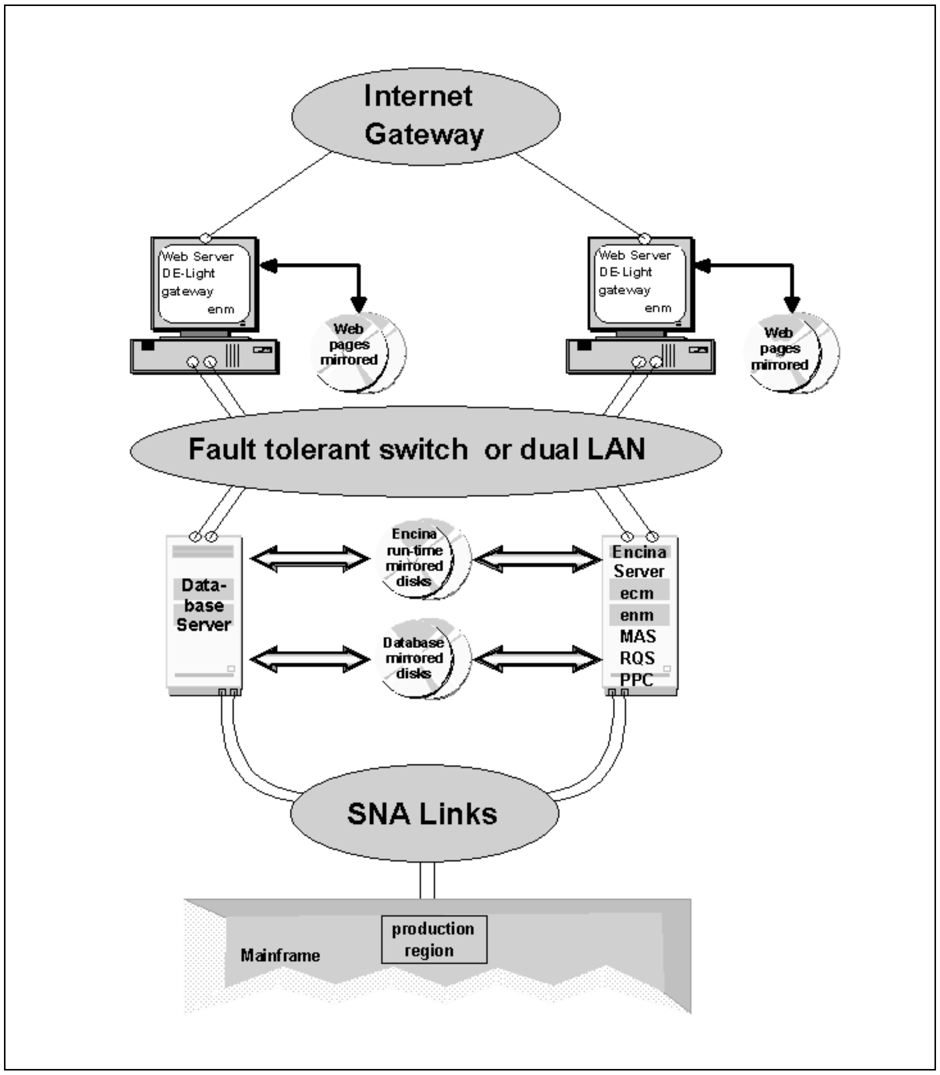


*Figure 136. Fault Tolerant Production Configuration*

All disks in the fault-tolerant production configuration are mirrored (see Section 11.4.3, “Encina Volume Mirrors” on page 300 for mirroring considerations). The Web servers run on two separate machines, using their own copies of the replicated Web pages. Should one of them fail, the Internet gateway (probably a fault-tolerant intelligent router) will forward all Web requests to the remaining Web server. Each Web server runs an Encina Node Manager (enm) and a DE-Light gateway.

Both Web servers are linked to the Internet gateway by a single network interface. Should that interface fail, the other Web server will pick up the traffic from the Internet gateway in the same way a CPU failure is handled. Each Web server uses two network links to the local fault tolerant switch (or dual LAN). Should any of those two links fail, the other one can be used to access the Encina server.

The rest of the Encina cell runs on the Encina server machine. The database server runs on a machine of its own, the database server. The database and Encina disks are shared between the Encina server and the database server. Each machine is connected to the local switch (or dual LAN) through two links. A failure of any single link still allows the machine to continue its service. Both machines are connected to the back-end mainframe, using separate pairs of SNA links. The Encina machine runs the Encina Cell Manager (ecm), the RQS server, the PPC gateway, and all MAS servers.

If the Encina server fails, the database server takes over its IP addresses and starts running the Encina cell straight from the Encina disks. Thus, as far as the outside world is concerned there is only a glitch in service as opposed to a total blackout. The database server now takes on the load previously shared between itself and the Encina server. Hence, the performance is affected but the service is uninterrupted (see Figure 137 on page 309).

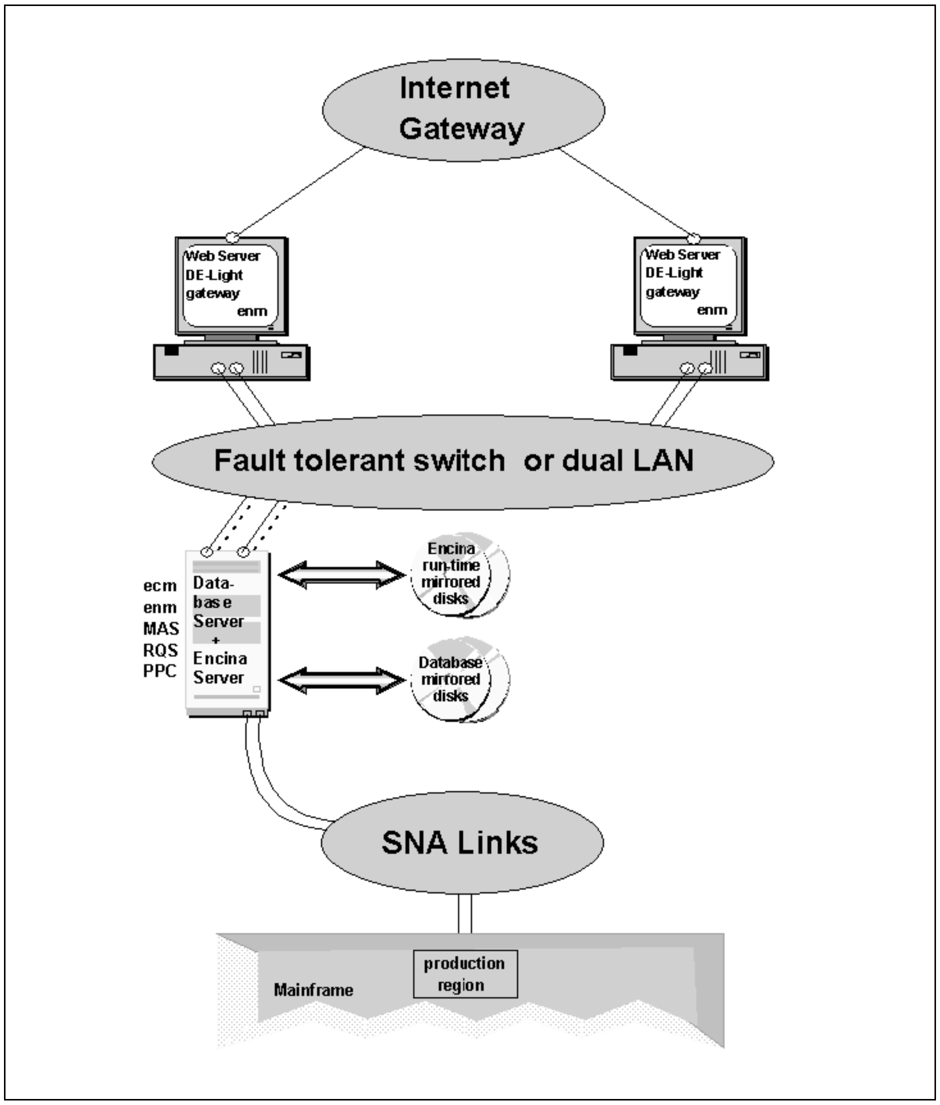

Internet Gateway
Web Server DE-Light gateway enm
Web Server DE-Light gateway enm
Fault tolerant switch or dual LAN
ecm
enm
MAS
RQS
PPC
Data-base Server + Encina Server
Encina run-time mirrored disks
Database mirrored disks
SNA Links
Mainframe
production region

*Figure 137. Encina Server Failure Configuration*

We present the Encina server failure configuration to give you an idea of what a single point of failure fault tolerant system looks like. You can further expand this scenario by adding more machines for performance reasons. For example, you can introduce a third Web server (see Figure 138 on page 310).

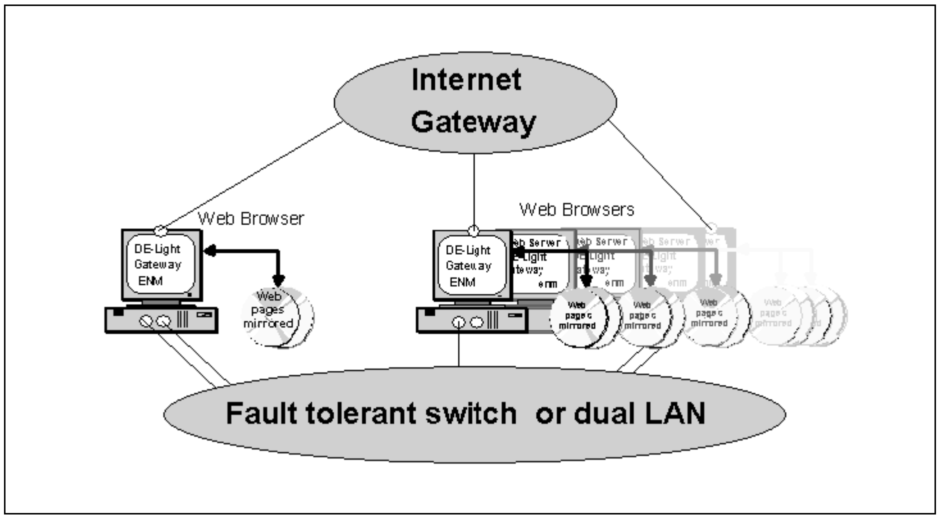


*Figure 138. Adding Extra Web Servers*

You can also increase the number of machines running MAS servers and replicate the interfaces across several different MAS servers on those machines (see Figure 139 on page 311).

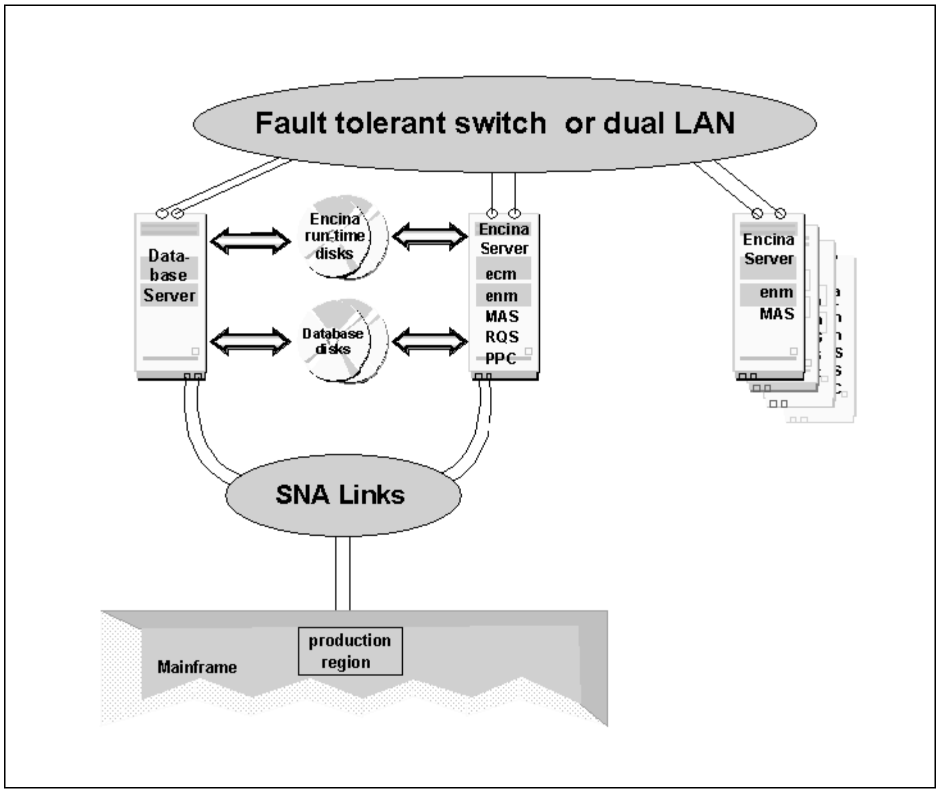


*Figure 139. Adding Extra Encina Monitor Application Server (MAS) Machines*

Add new machines to the Encina cell in a way that ensures that the database manager and the Encina Cell Manager can fail over to a backup machine. The same applies to RQS and SFS servers. Because all these components store data, if they fail, their backup processes must use the same data depository. None of them has the capability of running a backup server sharing the same data. The rest of the Encina servers only transfer data, and if they fail, a backup running on another machine can take over without the need to access the same data depository. Therefore, always keep the Cell Manager and your RQS and SFS servers on a machine sharing its disks with the database server, to fail over successfully.

## 11.5 Performance

Complex systems such as most Encina applications require performance tune-up before production deployment. In addition, consequent performance adjustments need to be made to keep up with the changing demands on the system.

Given the many factors that affect the performance of a complex, distributed multitier client/server system, we focus on a few techniques that you can use to improve the performance of your Encina applications.

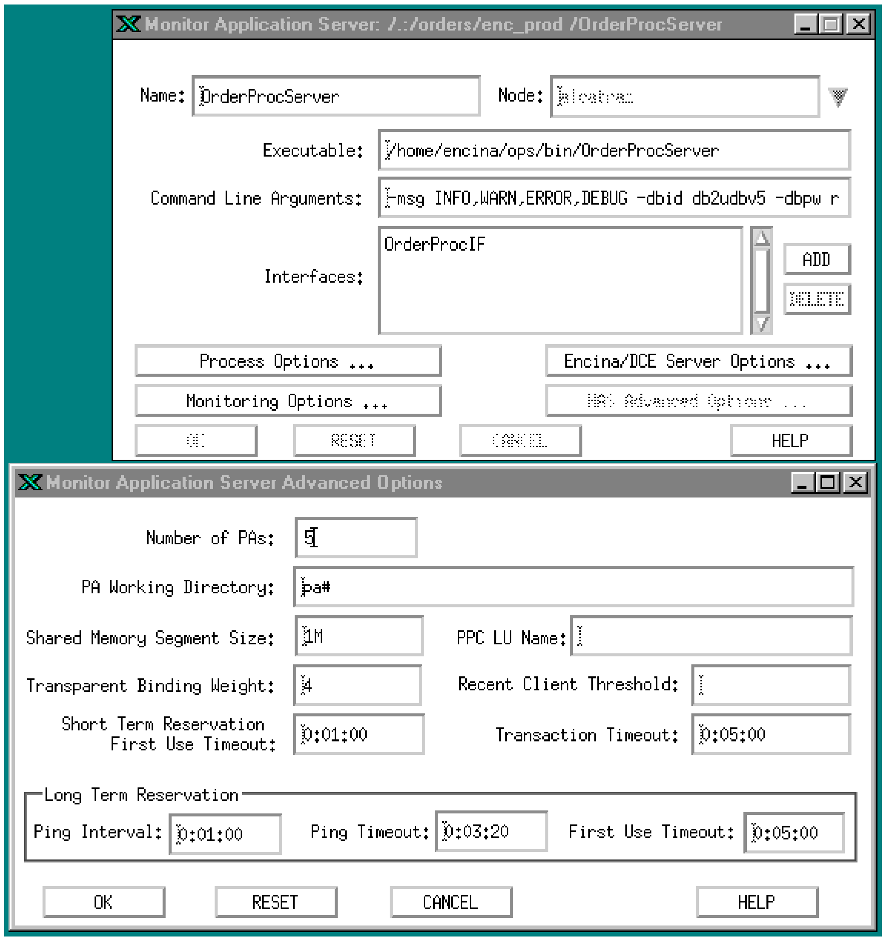
We recommend that you set up a special suite of performance or load tests on the production configuration before the deployment date. We also recommend that you use peak loads of a factor of 2 for the load tests to ensure that your system is prepared to handle the unexpected.

Our experience shows that the performance bottleneck in most Encina applications turns out to be either the communication infrastructure or the database. Once you have shortened the communication paths between your busiest components, increased the bandwidth available to you to the maximum, and optimized database queries, it is worth looking at further increasing your performance by tuning up the Encina components.

All Encina components communicate with each other and with DCE, but they do not heavily use the disk. You can increase the performance of the Encina servers and managers by examining the most loaded communication links between them and then remapping them to different machines.

Encina provides a mechanism for balancing the load between the different servers exporting the same interface. You can specify a priority when configuring a server. The Monitor uses this priority to balance requests among application servers that export the same interface. Client requests are distributed over a group of application servers on the basis of their priority. The priority is used only by clients that use transparent binding.

You can also increase the performance of an application server by using more than one instance of that server running on the same machine. These instances are called processing agents (PAs). You can define the number of PAs for each of your application servers, using enconsole (see Figure 140 on page 313).


Monitor Application Server: /.:/orders/enc_prod /OrderProcServer

Name: OrderProcServer Node: alcatraz

Executable: /home/encina/ops/bin/OrderProcServer

Command Line Arguments: -msg INFO,WARN,ERROR,DEBUG -dbid db2udbv5 -dbpw r

Interfaces: OrderProcIF

Monitor Application Server Advanced Options

Number of PAs: 5

PA Working Directory: pa#

Shared Memory Segment Size: 1M PPC LU Name:

Transparent Binding Weight: 4 Recent Client Threshold:

Short Term Reservation First Use Timeout: 0:01:00 Transaction Timeout: 0:05:00

Long Term Reservation

Ping Interval: 0:01:00 Ping Timeout: 0:03:20 First Use Timeout: 0:05:00

*Figure 140. Monitor Application Server Advanced Options Window*

In addition, you can specify how many threads you want for each PA. Encina automatically manages multiple threads running within the same PA.

You can have one PA running one thread, in which case all requests sent to any of the interfaces supported by the application server are executed sequentially. By increasing the number of threads, you can have the same PA handle several simultaneous requests from different clients. Because the threads share the same process space, their management is fast, and spawning a new thread requires little overhead. At the same time, your executables must be thread safe, in order to use multiple threads. You must write your own code in a thread safe fashion, and all libraries you link with (most notably the database libraries) must be thread safe.

Alternatively, you can increase the number of PAs and keep each of them with one thread only. The advantage of this scheme is that you do not need thread-safe code. Also, if a PA dies for one reason or another, the remaining PAs will handle the subsequent requests while the Encina Node Manager is bringing the failed PA up. The disadvantage of using many PAs is that the more processes you have on your system, the harder it is to manage them, and the more time it takes for the system to switch context between them.

Our experience is that in many situations it is more effective to spend some extra money on a better performing machine than to go through the (expensive) route of developing and testing thread-safe application code and third-party libraries. The latter approach is usually reserved for high-performance-oriented real-time systems.

In addition to using the techniques described in this chapter, you can increase the performance of your system by scaling it up. Introducing new hardware is in most cases less expensive than spending time on performance tune-up and testing.

# Chapter 12. Application Deployment

Applications become increasingly sophisticated with the development of technology and the rise of customer demands. They run in heterogeneous environments and interface with a wide variety of users and other applications. At the same time, project deadlines are shrinking and budget pressures are intensifying. As a result, the complexity of the systems delivered increases, and their deployment into production sometimes takes as long as their development.

In the face of this reality the issues of application deployment should be considered from day one of the application design and development. The application design should reflect the desired process of deploying the application into its target production environment.

In this chapter we provide a brief overview of the application deployment issues you should be aware of from the very start of your project. Then we focus on three particular aspects of deployment on which Encina has the most impact: staging methods, production environment installation and configuration, and replicating the environment configuration.

## 12.1 Overview

When deploying applications you have to deal with a large variety of issues. To fully appreciate the job ahead and to minimize the number of unpleasant surprises, we recommend the following approach to application deployment:

- Assign a deployment team leader on day one of your project.
- Determine the deployment issues as early on as humanly possible. Perform site surveys if you do not have the whole picture or if you doubt its correctness.
- Put together a living deployment plan. Get your customer to buy into it.
- Start addressing the deployment issues as early as possible and in parallel with your development effort. Dry run various deployment phases.
- Track the progress of your deployment effort religiously.

It is hard to list all possible deployment issues you need to deal with but we mention here the most common ones. All these issues are not limited to Encina. In subsequent section of this chapter we discuss in greater detail some of these issues in light of Encina deployment.

***Psychological Aspects***
Before you start planning any deployment ask yourself a simple question: What does this deployment mean to my customer? Your number one priority should be to plan and schedule the deployment around the customer’s expectations and based on the context of your application. For example, if you are deploying a new ticket reservation system, do not deploy in December. If this is the first computer system your customer is about to use, plan more time for user training. When assessing each of the deployment issues in this chapter, make sure you think about its psychological aspects. After all, the success of your system is measured only by the satisfaction of your customer.

***External Interfaces***
Find out to what your system interfaces. Make sure to obtain as strict technical specifications of the interfaces as you can. It is of utmost importance to involve the customer in this process and to obtain its written consent with your findings. Be prepared for the interfaces to change during the development of the application and plan for changing your application accordingly. Schedule an interface test of your application as early as possible. Frequently a beautiful application cannot even start because, let’s say, the external data feed no longer comes over X25 but over TCP/IP instead.

***Documentation and Procedures***
Plan ahead for putting together decent documentation to go along with your application. The documentation should not only describe the functionality of the application but also the necessary procedures for using it and maintaining it. Developing documantation is usually expensive; to compensate for unexpected costs many managers simply force their technical staff to put together a collection of bulleted lists which is then handed to the puzzled customer.

Professional documentation enhances the effectiveness of your presentation. Remember, the customer’s users have no idea what they are getting into, and they tend to forget what they heard in the training courses as soon as they walk out the door. You may have been focused on the application for months and years. The new users, however, have other things to worry about, and they will only have a short period of time to like the application or hate it. Your documentation may be the turning point.

***Acceptance Tests***
You and your customer need to prepare test suites to be run during application acceptance. It is important to prepare tests that not only allow you to test the system requirements one after another but also give the customer

a feel for how to use the new application. In this way, you have a better chance to "buy" the key customer personnel in the application. These people later on will become your best supporters and will spread the word among their colleagues.

### *Performance Tests*

Performance tests could be viewed as part of the acceptance tests, but we separate them to emphasize the importance of testing the ability of your application to handle peak loads. A carefully prepared set of tests should be planned to demonstrate to your customer that the application behaves as expected during peak hours. Think how you are going to measure the performance. You may have to build measurement capabilities into your system or purchase a third-party measurement product. Performance tests require a lot of preparation and precise setup. Plan for several dry runs and reruns.

### *Staging Methods*

Whatever tests you do during development, you still have to test the application in the target production environment before the rollout date. This may not always be possible, so you have to emulate the production environment in one way or another. Choose your staging strategy early on and stick to it. See Section 12.2, “Staging Methods” on page 319 for more detail on staging methods for Encina applications.

### *Installation and Configuration*

Surprisingly many people start thinking about the installation and configuration of the target operational environment two weeks before the rollout date. Do not let this happen to you. You should be thinking about the production configuration as soon as you start designing your application. See Section 12.3, “DCE and Encina Installation and Configuration” on page 324 for details on configuring Encina applications.

### *Re-creating the Configured Environment*

You may have to build your staging and production environments more than once, so you must be prepared to do it quickly and without errors. Your ability to re-create your environments is an issue that becomes even more pronounced with Encina applications given the complexity of configuring Encina cells. For further details see Section 12.4, “Replicating Encina Cell Configuration” on page 343.

### *Initial Data Load*

Many applications require some preexisting data for their proper functioning. Sometimes this data is taken for granted, and you may not realize when you first roll out your application that this data must be available to you. For some applications, such initial data may take weeks and months to gather. You have to find out whether you need an initial data load, how long it is going to take, who is going to collect the data, how the data is going to be incorporated into the new application, and how are you going to verify its correctness.

***System Upgrades***

What better time to think about system upgrades than the time when you start planning your initial installation. Every system needs upgrades even if you are certain that yours will be a one shot deal. This issue has many aspects such as upgrading a live system and dealing with inconsistent data. Be prepared to incorporate mechanisms for hardware and application upgrades into your application design.

***Users and Training***

It is important to realize that different types of users will use the application. They can be categorized by their roles, computer literacy, and stage of involvement. Make sure to find out who the users will be and possibly meet some of them early on to get a better feel for the kind of training they need to start using your application. Remember that it is in your best interest to find early supporters among users. You should try to identify those supporters before the actual deployment and possibly involve them in your design, tests, and other development stages.

***Administration Routines***

Topics such as data archiving, application security, and disaster recovery always seem to come up at the end of the project and somehow nobody likes to deal with them. Developers find it boring to think about backups, the customer assumes the application will take care of the administrative details, and everybody seems to believe that administration will just happen when the system is rolled out.

You will be in a much better position if you plan for assigning a person responsible for all aspects of system administration from the very beginning of your project. Try to nail down an explicit set of administration routines that you expect your customer to assume after the rollout. In addition, try to involve the customer’s system administrators in the deployment process as early as possible so that they can assume their roles as planned.

***Deployment Team***

Finally, but not lastly, you have to pick a good deployment team on the basis of the issues you believe have to be dealt with during the rollout. A friendly installation manager can be worth his or her weight in gold when unexpected problems crop up and your technically skilled deployment team has been onsite for three weeks longer than initially anticipated. Think about the fact that the first impression your customer has of your system is the most lasting one, and your deployment team will be the one delivering that impression.

## 12.2 Staging Methods

Having a good staging strategy is an issue of project management persistently underestimated by software developers. Numerous examples of projects exist that proceed to develop everything without even thinking of how the application will be actually brought into production. Deadlines come and pass, and there is never enough time left for proper deployment, let alone staging. Developers always somehow manage to deliver the application, and that is when all the trouble begins. Users eagerly put their hands on their brand new, expensive acquisition only to find out that it does not behave the way the developers thought it would.

There are many reasons for such problems. Whatever they are there is a cure: testing the application in the production environment. No matter how rigorous the unit tests or any other tests for that matter, the application must be tested in the actual production environment, which is always different from the development and testing environments.

Of course, this is much easier said than done. In many cases it is impossible to have access to the production environment long enough to test the system properly. That is why the application must be staged into the production environment. After the developers have completed their testing and the application is believed to be ready for production, it should be installed in a staging environment.

The staging environment should be as close to the production environment as possible. All files installed on it should come from the project code management system and should be version controlled. This includes not only the executables but also configuration files, administrative scripts, data files, and other support files needed for the proper operation of the application. The software products required by the application, such as the operating system, database engine, DCE, and Encina, should be installed according to a documented procedure. The application itself as well as any initial data loads should follow a documented procedure. The entire process should be planned well ahead of development completion and handled by the people responsible for application deployment.

While this may sound like overkill for small projects, we are certain that it is a “necessary evil” regardless of project size. Staging provides the developers and the people deploying the system with a clear picture of what the system will actually look like when users get their hands on it. It also greatly reduces the risk of accidental misconfiguration, not to mention the well known “ooops, we forgot about that” factor.

Encina applications are no exception. The way to handle the staging depends on the size of your application. Below we use our case study application to illustrate the main idea behind staging.

We have three different environments: the development machine, the staging environment, and the production environment. The development machine is configured as part of a development DCE cell (see Figure 141 on page 320). We run several different Encina cells on this machine. The cells are used by different developers to unit test their software. One of the cells is designated as the release cell, /.:/orders/enc_release. This cell is configured to use a set of executables that are built by the code management librarian using controlled code obtained from our code management library. This cell is used for regression testing and for demos of the current state of the application.

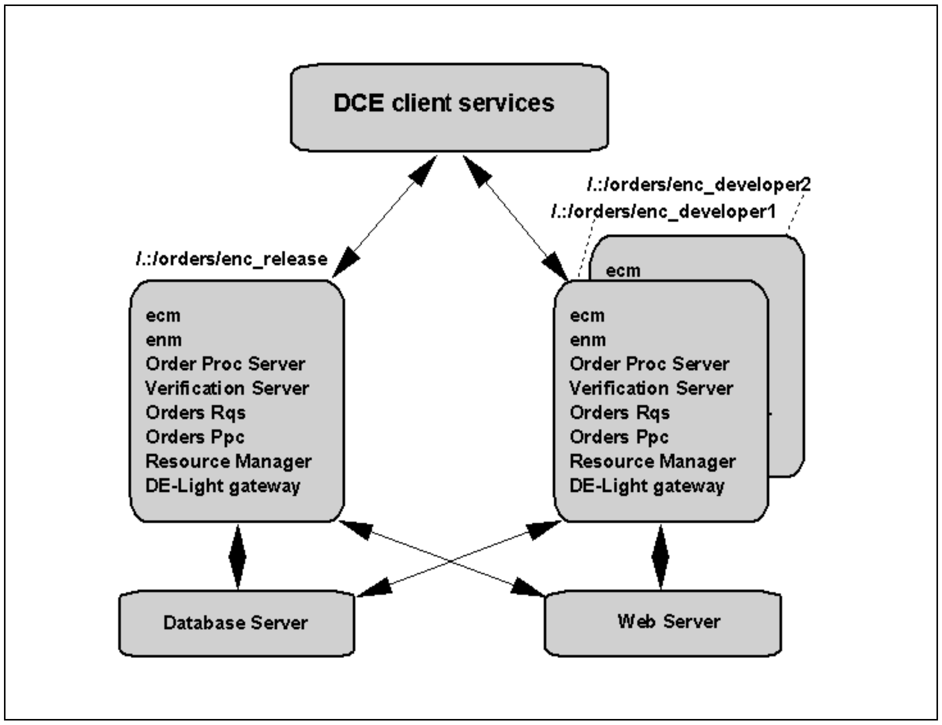


*Figure 141. Application Development Machine Configuration*

The rest of the cells are used by the developers to run executables linked against their code and the libraries from the latest release. One of those cells is used to test the Encina server configuration scripts.

The staging environment consists of a stand-alone staging DCE cell (Figure 142 on page 322). The database server is identical to the production database server. The Encina server machine is set up the same way the production Encina server is set up. The Web server machine is configured identically to one of the Web server machines we use for production. The difference between the staging environment and the production environment is the lack of a second Web server. We do not have it because we could not afford it or so we thought. The other difference is that we use internal browser machines for testing the Web interface through DE-Light and a staging region on the mainframe.

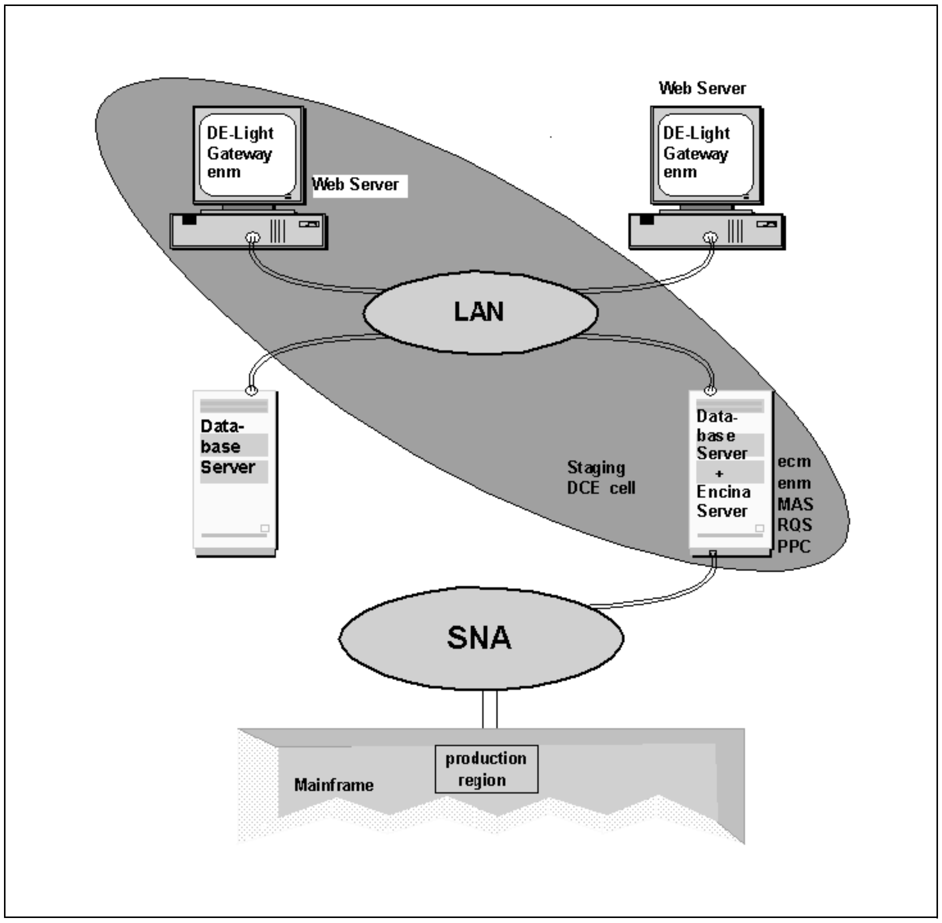


*Figure 142. Application Staging Environment*

The staging environment is configured using documented procedures for setting up the operating system, the database, DCE, Encina, and the other third-party products we use. The Encina servers are configured using the configuration scripts developed and tested on the development machine. The application files are pulled from the code management library and built and released as per our official build release procedure. The staging database is loaded with operational data. It is connected to a staging region on the mainframe that also contains operational data.

We are going to run the entire suite of acceptance tests on the staging environment. In addition, we are going to run some tests that the customers are not going to see but which are needed for our own peace of mind. Every time we release a new version of our application, it gets tested in the staging environment before it is released into production.

The production environment is run as a separate DCE cell (Figure 143 on page 323). It is configured exactly as we configured the staging environment. The only difference is that we configure two Web servers instead of one.

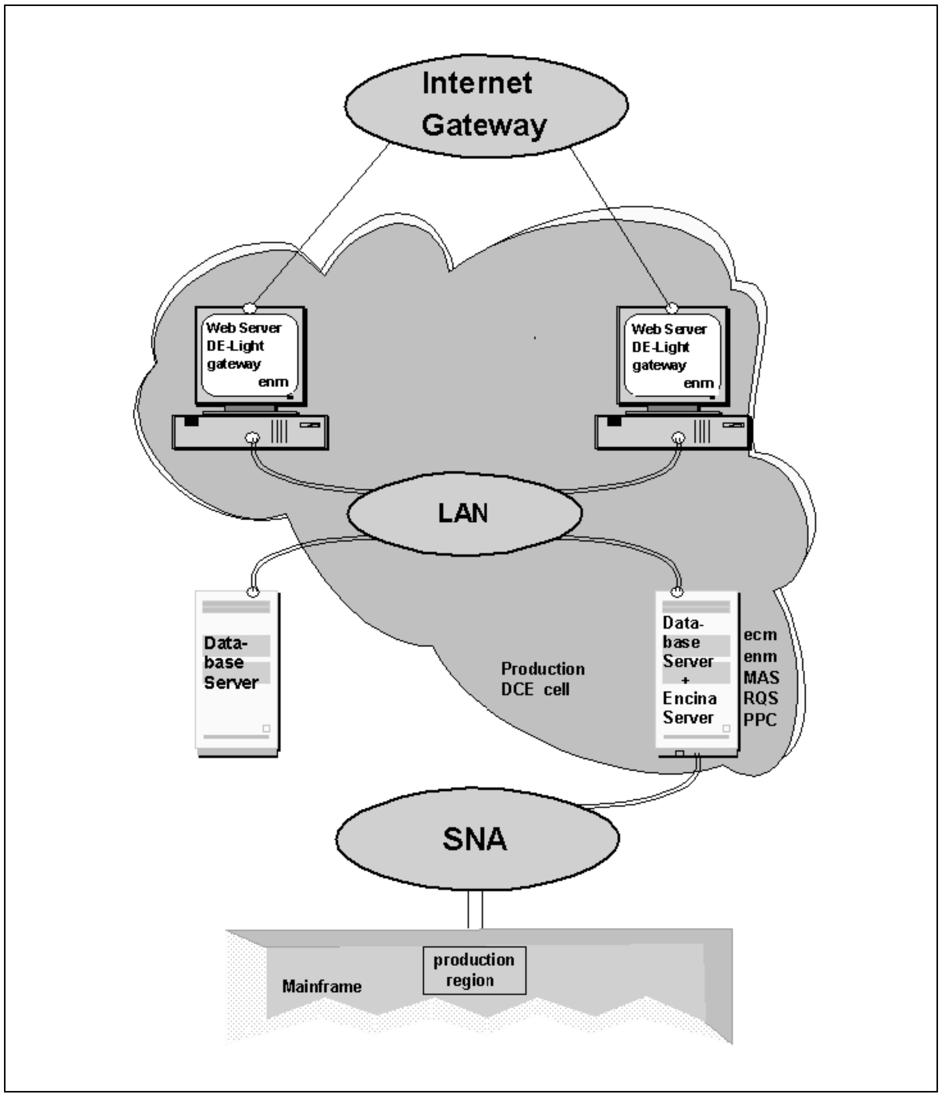


Figure 143. Sample Production Environment

Any configuration changes required in the production environment are included in the configuration update script. The script is then tested in the development environment and submitted to the code management system. It becomes part of the next release. The release could be a full release or just a patch release (a subset of the full release). The next release is installed on the staging environment first where we perform the necessary set of regression tests (possibly a subset of the acceptance tests). As soon as we are satisfied with the correctness of the new release, we move it into the production environment in a scheduled manner during offpeak hours.

The same mechanism is used for introducing patches and new releases of the third-party software products, such as the operating system, the database engine, DCE, and Encina.

As you can see from the description of our staging process, correcting errors after the initial release is an expensive process. There are always areas for improvement in software applications, and new patches and software versions are released regularly. You cannot avoid releasing new versions of software. You can, however, increase the time between two consecutive releases by strictly executing a thoroughly planned set of acceptance and load tests before application rollout.

## 12.3 DCE and Encina Installation and Configuration

You have to be very careful when configuring DCE and Encina. Remember that DCE and Encina are performed by the superuser and the Encina cell is configured by the Encina user. Be sure to test your setup after each installation and configuration step. Ideally, you should have a written procedure describing the environment configuration. DCE and Encina must be part of that procedure. In the sections that follow we provide an overview of the Encina and DCE installation steps.

### 12.3.1 Operating System Preparation

Everything described in this section is performed by the superuser. We do not recommend proceeding to any of the other steps before you have prepared the operating system for DCE and Encina.

#### 12.3.1.1 System Users and Groups

We suggest that you use three different operating system users in your production environment; the superuser, and the Encina user, the operator user. You may find that you need other users as well, a database user for instance.

The superuser owns all software installed on your system, such as DCE and Encina. In addition, the superuser runs DCE.

The Encina user runs all Encina managers and servers. It is also used to configure the Encina cell and the Encina servers. The actual owner of the Encina account could be the same people who administer the operating system, such as the system administrators.

The operator user has access to the Encina message log files. The operator is allowed to start and stop Encina servers through enconsole; however, the operator is not allowed to change the Encina configuration.

Before the installation of DCE and Encina, the system administrator creates the encina and encinaop accounts for the Encina user and operator, respectively. The system administrator also creates the group encina. The initial two members of the group are the encina and encinaop users. For user encina we recommend choosing a default file creation mask, such as 027 on UNIX, that grants read access to the group only. This approach will eliminate any unsanctioned access to files produced by Encina, such as the message log files.

Once you have created the Encina user (encina), you must set up its environment. The environment setup depends on your operating system, but the following variables must be set:

- PATH -- must include /opt/dce/bin, /opt/encina/bin, and /opt/encina/etc
- ENCINA_TPM_CELL -- must be set to the full CDS name of the Encina cell, such as /.:/orders/enc_prod. All Encina tools use this variable

In addition to these two variables you may want to set up some of the variables discussed in Section 13.1, “Environment Setup” on page 347.

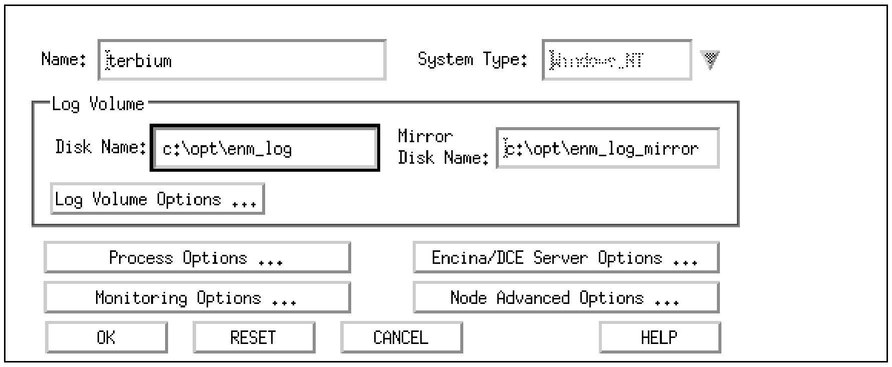
#### 12.3.1.2 Disk Space Allocation

You have to allocate space for the DCE and Encina software, space for the Encina and DCE run-time files, and space for the Encina volumes. For fault-tolerance purposes, consider using mirrored disks for your entire system. For more information on fault-tolerant design, see Section 11.4, “Fault Tolerance and Encina” on page 299.

The amount of space required for the DCE and Encina software is specified in the release notes for those products. All you have to do here is ensure that there is enough space left on the devices that will host the software.

The same applies to the DCE and Encina run-time files, which are located under /opt/dcelocal and /opt/encinalocal by default. Make sure you have sufficient space left in your /var/dce directory used by DCE. You even may want to create an entire partition dedicated to /var/dce to avoid conflict between DCE and other processes using /var.

Encina uses its own file system type for managing the data and log volumes needed by several of its components. Unless your operating system provides disk mirroring you should also create mirror volumes for all Encina volumes (see Figure 144 on page 326).


Name: terbium
System Type: Windows_NT

Log Volume

Disk Name: c:\opt\enm_log
Mirror Disk Name: c:\opt\enm_log_mirror

*Figure 144. Definition of Encina Mirror Volumes for Node Managers*

Encina uses raw disk partitions (logical volumes on AIX) for creating its volumes. Therefore, you must allocate a sufficient number of raw disk partitions of the right size before installing Encina. You need a raw partition for each volume used by Encina. There are two types of Encina volumes: data volumes and log volumes. Data volumes are used by the Encina Cell Manager, RQS, and SFS for storing Encina data. All these processes also use log volumes to keep a log of any changes made to their data volumes. The Encina Node Manager and the PPC gateway use log volumes only to maintain transaction state information.

In our example, machine prod_one runs the Encina Cell Manager and the Encina Node Manager. Therefore, you have to allocate three raw partitions, two for the Cell Manager and one for the Node Manager. We can name these partitions ecm_data, ecm_log and enm_log. If you do not have support for mirrored disks, you have to allocate another three raw partitions, ecm_data_mirror, ecm_log_mirror, and enm_log_mirror, for the mirror volumes.

If you want to add an RQS server to this machine later on, you have to create two more raw partitions, rqs_data, and rqs_log. Again, if system mirroring is not provided, you have to create two more raw partitions, rqs_data_mirror and rqs_log_mirror, as mirror images for the RQS data and log volumes.

For more information about the volumes used by Encina see Transarc's *Encina Administration Guide Volume Two: Basic Administration*.

#### 12.3.1.3 Directory Structure

You must grant the right ownership to all partitions you created in Section 12.3.1.2, "Disk Space Allocation" on page 325. Be aware that the Encina user (encina) must exclusively own all device files associated with these raw partitions. To prevent anyone else from accessing the devices, use the following commands:

```
cd /dev
chown encina:encina ecm_data ecm_log enm_log
chown encina:encina ecm_data_mirror ecm_log_mirror enm_log_mirror
chmod 600 ecm_data ecm_log enm_log
chmod 600 ecm_data_mirror ecm_log_mirror enm_log_mirror
```

If you are running AIX and using the AIX mirroring capabilities, you do not need the *_mirror volumes. AIX can automatically maintain mirror images of the file partitions you have created. If you are using this feature, you have to change the permissions and ownership of these devices as well:

```
chown encina:encina recm_data recm_log renm_log
chmod 600 recm_data recm_log renm_log
```

On Solaris machines you have to create raw partitions when formatting the disks for the first time. You must create soft links to these partitions in /dev and use those link names when configuring Encina. This allows for better readability and configuration portability.

On Windows NT machines you must create one or more fully allocated operating system files and use them as your Encina volumes. You can use the Encina fileVol program (or your own program) to create the files. The fileVol program creates a fully allocated operating system file. The command syntax is:

```
fileVol filename filesize
```

Specify the name of the file to create as the filename argument and the size of the file (in bytes or kilobytes) as the filesize argument. Specify bytes as an integer and kilobytes as an integer followed by the letter k. For example, the following command creates a fully allocated 4000 KB operating system file named D:\rqs_data:

```
fileVol D:\rqs_data 4000k
```

The next step is to link the Encina and DCE software directories to /opt/dce and /opt/encina, respectively. Then you have to create the Encina working directories, /opt/encinalocal and /opt/encinamirror, and change their ownership to encina:

```
cd /opt
mkdir encinalocal encinamirror
chown encina:encina encinalocal encinamirror
chmod 750 encinalocal encinamirror
```

At this point you are ready to proceed to the installation of DCE and Encina.

### 12.3.2 DCE and Encina Installation

The superuser installs DCE and Encina. Follow the installation notes provided for your platform. Make sure that you install all components needed on each machine that is part of your environment. When in doubt, install more. With DCE and Encina it does not hurt to install more components than you need, provided you have sufficient disk space.

### 12.3.3 DCE Configuration

After you have installed the DCE and Encina packages, you can proceed to configuring them. The exact mechanism for DCE configuration depends on the DCE vendor and the target platform. The installation guides that come with DCE cover the configuration aspects.

Before you start the DCE configuration you need to design the DCE topology prior to configuring DCE. For more information about designing DCE cells, see the redbooks *Administering DCE and DFS 2.1 for AIX* (SG244714-0*)* and *DCE Cell Design Considerations* (SG244746-0).

The main rules we recommend are these:

- Use as small a number of DCE cells as possible. One is enough for a single site configuration. Production cells should be separate from development cells.
- Place your master CDS, security, and time servers on the most secure machine on your system. Make sure the machine is physically secure too.

- Use at least one replica of CDS, the security server, and the time server. The replicas should be accessible even if the link to the master is unavailable.

Once you have decided on your DCE topology and read through the DCE configuration procedure in your DCE installation guide and release notes, you can proceed and configure the master CDS and security server. Make sure that only the DCE administrator has access to the cell_admin (DCE superuser) account. The DCE administrator will perform the initial Encina cell configuration.

### 12.3.4 Initial Encina Cell Configuration

There are two stages to configuring your Encina application. During the first stage the Encina Cell Manager and the Encina Node Managers are configured. Also several DCE objects are set up for Encina. During the second stage all Encina servers are configured. Section 12.3.6, “Encina Server Configuration” on page 342 focuses on the second stage. Both stages are performed by the Encina user.

To execute the initial Encina cell configuration, you need to have access to the DCE *cell_admin* account because at this stage you set up various DCE objects in the DCE CDS.

To login to your system as the Encina user, obtain the DCE cell administrator credentials by logging into DCE as cell_admin (or whatever the cell administrator account name might be), and start enconsole.

When enconsole asks you whether you want to define an Encina cell, answer “Yes." The only option presented to you at this point by the main enconsole window is Define->Cell. Examine all fields presented to you and type in the correct information. You should have the information required for each field ready by the time you start configuring the Encina production environment. Generally, you can leave the default values. The fields that you do need to fill in are the data and log volume names and the Encina cell name. Make sure to set a correct value for the priority field under Process Options.

You are ready for the cold start of your cell as soon as you have defined it. Encina distinguishes between cold and warm starts. The first time you start any Encina server, it performs a cold start. The cold start involves some DCE configuration and the creation of certain files, most notably the keyfile for the started server and the server restart file. Every consequent startup is a warm start. No configuration is performed. The server restart file is used instead.

You can perform the cold start of your cell by selecting Actions->Start->Cell. It takes some time to go through all the steps. If you encounter errors, read the messages carefully and remove the cause of the errors. During the cold start you are asked to enter the name of the Encina administrator DCE principal and its password. If the name and password have not been created, enconsole creates them for you.

The next step is to create and configure the Encina Node managers on all machines that belong to the Encina cell. This process is similar to configuring the Encina Cell Manager. With the Cell Manager running, select Actions->Define->Node to define the Node Managers and Actions->Start->Node to cold start them. Although you can define the Node Managers from any machine within the Encina cell, you have to perform the cold start from the machine on which the Node Manager is running. In order to perform the cold start, you have to install DCE and Encina on that machine and configure DCE.

Once you bring up the Encina cell, restart it, using the Cell Manager restart script, *rc.encina.cell*. The startup script invokes the encinaNanny program, which runs the Encina Cell Manager. Should the Cell Manager terminate for any reason, encinaNanny will restart it. The Cell Manager restart script is created during the Encina Cell Manager cold start and is located in its working directory (/opt/encinalocal/<cell name>/ecm by default). You can stop the Cell Manager by running:

```
rc.encina.cell stop
```

You can start the Cell Manager by running:

```
rc.encina.cell start
```

Similarly, restart the Node Managers on each machine by using the Encina Node Manager restart scripts created during the cold start of the Node Managers. These restart scripts, rc.encina.<node name>, are located in the Node Managers working directories, /opt/encinalocal/<cell name>/node/<node name> on each machine. Again, the two parameters that you can use are stop and start:

```
rc.encina.prod_one stop
rc.encina.prod_one start
rc.encina.prod_two stop
rc.encina.prod_two start
```

**IMPORTANT**: Always start the Encina Cell Manager and the Encina Node Manager with the startup scripts to ensure that they are controlled by an encinaNanny.

Use enconsole to manage the Encina servers but not the Cell Manager and Node Managers. Section 12.3.5, “Automatic Restart Setup” on page 331 explains how to set up your system to automatically start the Cell and Node Managers.

### 12.3.5 Automatic Restart Setup

Before proceeding with the setup of the Encina servers, configure DCE and Encina for automatic startup on reboot. We recommend that you perform the setup now so that you can easily test it without having to start up all servers. Once you are confident that your autorestart configuration works, you can finish off the application configuration.

Your operating system provides a means of running a defined set of command scripts on reboot. AIX uses the /etc/inittab file to describe all commands to be executed when the system is brought up. Sun Solaris uses the /etc/rc*.d directories to describe which scripts need to be started. Windows NT uses the startup folder located in the Start Menu folder for the Administrator profile:

```
C:\WINNT\Profiles\Administrator\Start menu\Programs\Startup.
```

Your task is to determine the exact startup mechanism and then plug in the DCE and Encina startup scripts in their appropriate place. DCE must be started before you start Encina.

The DCE distribution for your operating system comes with a script called rc.dce. You can use this script to start or stop DCE on your machine. The exact location of the script depends on the vendor from whom you purchased DCE. You have to test the script before using it for automatic startup. Login as the superuser and run this command to shut down all DCE processes on your machine:

```
rc.dce stop (on non-AIX machines)
dce.clean (on AIX)
```

Run this command to start up all DCE processes configured for your machine:

```
rc.dce
```

Once you are convinced that rc.dce works properly, you can simply add a reference to it in the system startup location. No further modification is necessary.

As with DCE, you need to provide an Encina rc.encina startup script to the system startup facility. Unlike rc.dce, it is your responsibility to create the rc.encina script because you have to decide what Encina processes should be started on system reboot.

We recommend that you automatically start your Cell and Node Managers as well as all your servers on reboot. Use the script shown in Figure 145 on page 333. The script is prepared for the machine running the Encina Cell Manager. You can tailor it to any other machine on your system by removing the Cell Manager startup section.

```
#!/bin/sh
#
# Encina starter and stopper
#
#
#---------------------------------------------
# Rado Nikolov, Encina Support Group 12/02/97
#---------------------------------------------
COMMON_ENV_DIR=/opt/dcelocal/etc
#######################################################
# Import the definitions of the DCE/Encina Environment
#######################################################
if [ -r $COMMON_ENV_DIR/DCE_ENCINA_ENVIRON ]; then
   . $COMMON_ENV_DIR/DCE_ENCINA_ENVIRON
fi
CONFIG_FILE=/opt/encinalocal/CELL_LIST
if [ ! -f $CONFIG_FILE ]; then
   echo "No Encina cells configured in $CONFIG_FILE on this host"
   exit 0
fi
CELL_LIST=‘sed -e '/^#/d' $CONFIG_FILE‘
if [ -z "$NODE" ]; then
    NODE=‘uname -n|sed -e s/\\\\..*//‘
fi
PATH=/opt/encina/bin:/opt/encina/etc:$PATH; export PATH
NLSPATH=/opt/encina/msg/%L/%N:/opt/dce/nls/msg/en_US.ASCII/%N;
export NLSPATH
LANG=C; export LANG
SHPS_IN_RC=3; export SHPS_IN_RC
killproc() {# NUKE the named process(es)
pid=‘/usr/bin/ps -ef |
     /usr/bin/grep $1 |
     /usr/bin/grep -v grep |
     /usr/bin/awk '{print $2}'‘
[ "$pid" != "" ] && kill -TERM $pid
sleep 5
pid=‘/usr/bin/ps -ef |
     /usr/bin/grep $1 |
     /usr/bin/grep -v grep |
     /usr/bin/awk '{print $2}'‘
[ "$pid" != "" ] && kill -KILL $pid
}
```

*Figure 145. (Part 1 of 4) Sample rc.encina Script*

```
#
# Main routine
#
case "$1" in
start)
for USER_CELL in $CELL_LIST
do
    set ‘echo $USER_CELL | sed 's/:/ /'‘
    CELL=$2 ; USER=$1
            CELL_DIR=/opt/encinalocal/$CELL
            ENCINA_TPM_CELL=/.:/$CELL
            export ENCINA_TPM_CELL
    echo "Monitor cell \"/.:/$CELL\" managed by user $USER"
    #
    # Start up ecm
    #
    if [ -x $CELL_DIR/ecm/rc.encina.cell ]; then
       su $USER -c "$CELL_DIR/ecm/rc.encina.cell start"
    fi
            #
            # Wait up to 120 secs for the cell manager to start
            #
            echo "  # Waiting for Cell Manager to become fully
                      operational..."
            ecm_wait /.:/$CELL/ecm -t 120 -s ecm
            echo "  # Cell Manager Operational - Continuing..."
    #
    # Enable and then start enm
    #
            echo "  # Enabling Node Manager..."
    enccp -c encinaNodeManager enable $NODE
    if [ $? -ne 0 ]; then
       echo "Error enabling node $NODE. Exiting."
       exit 1
    fi
    if [ -x $CELL_DIR/node/$NODE/rc.encina.$NODE ]; then
       su $USER -c "$CELL_DIR/node/$NODE/rc.encina.$NODE start"
    fi
```

*Figure 146. (Part 2 of 4) Sample rc.encina Script*

```
            #
            # Wait up to 60 secs for the node manager to start
            #
            echo "  # Waiting for Node Manager to become fully
                    operational..."
            enm_wait /.:/$CELL/node/$NODE -t 60 -s enm
            echo "  # Node Manager Operational - Continuing..."
            #
            # Stop any servers that might still be running
            #
            if [ -x $CELL_DIR/node/$NODE/rc.encina.servers ]; then
               echo "  # Shutting down any hanging servers ... "
               $CELL_DIR/node/$NODE/rc.encina.servers stop
            fi
            #
            # Start all servers
            #
            if [ -x $CELL_DIR/node/$NODE/rc.encina.servers ]; then
               echo "  # Starting all servers ... "
               $CELL_DIR/node/$NODE/rc.encina.servers start
            fi
done
;;
stop)
        for USER_CELL in $CELL_LIST
        do
            set ‘echo $USER_CELL | sed 's/:/ /'‘
            CELL=$2 ; USER=$1
            CELL_DIR=/opt/encinalocal/$CELL
            ENCINA_TPM_CELL=/.:/$CELL
            export ENCINA_TPM_CELL
            echo "Monitor cell \"/.:/$CELL\" managed by user $USER"
            #
            # Stop all servers
            #
            if [ -x $CELL_DIR/node/$NODE/rc.encina.servers ]; then
               $CELL_DIR/node/$NODE/rc.encina.servers stop
            fi
            #
            # Stop the node and cell managers
            #
```

*Figure 147. (Part 3 of 4) Sample rc.encina Script*

```
            #
            if [ -x $CELL_DIR/node/$NODE/rc.encina.$NODE ]; then
               $CELL_DIR/node/$NODE/rc.encina.$NODE stop -all
            fi
            if [ -x $CELL_DIR/ecm/rc.encina.cell ]; then
               $CELL_DIR/ecm/rc.encina.cell stop
            fi
        done
        #
        # kill any leftover encina manager processes
        #
        killproc encinaNanny
        killproc enm
        killproc ecm
;;
*)
echo "Usage: $0 { start | stop }"
;;
esac
exit 0
```

*Figure 148. (Part 4 of 4) Sample rc.encina Script*

To use the rc.encina script as it is presented here you also have to create two other files. One file, /opt/encinalocal/CELL_LIST, is owned by the Encina user and contains the list of cells running on your machine (see Figure 149 on page 337 for a sample CELL_LIST file). This file eables you to automate various administrative tasks across different Encina cells, but you must keep it up to date.

```
#
# This file contains the list of Encina cells running on this machine.
# Each cell is specified on a single line.
# There are no other lines in the file.
# Each line has the following format:
#    <uid>:<Encina cell>
#    where <uid> is the UNIX user id running the cell and
#          <Encina cell> is the Encina cell name excluding the /.:/
prefix.
#
encina:orders/enc_prod
```

*Figure 149. Sample CELL_LIST File*

The other file you have to create is rc.encina.servers (see Figure 150 on page 338). Place this file in /opt/encinalocal/<Encina cell name>/node/<node name> on each of the machines in your Encina cell. The file is tailored to each Encina cell and node. It describes the order in which the different Encina servers on a given node are started.

```
#!/bin/sh
#
# Encina server start-up script
#
###############################################################

#--------------------------------------------
# Rado Nikolov, Encina Support Group 12/02/97
#--------------------------------------------

#
# List all servers which must be running before any other servers are started

FIRST_SERVERS="ordersPpc"

#
# Define the startup/shutdown order for all server groups

START_ORDER="node/bengal"
STOP_ORDER="node/bengal"

#
# Define global variables

# file used for temporary purposes
TMP_FILE=/tmp/pingtest.$$

# sleep time (in secs) between two attempts to check a server status
SLEEP_TIME=3


# Check if the specified server is running; return 1 if it is not
# $1 -- server name

is_running()
{
   CURR_STATE=‘enccp -c genericServer show $1 -attribute currentState 2>&1 | \
               cut -d' ' -f2 | cut -d'}' -f1‘

   # Return success if the current state equals 3 (running)
   if [ "$CURR_STATE" = "running" ]; then
      return 0
   else
      return 1
   fi
}
```

*Figure 150. (Part 1 of 4) Sample rc.encina.servers Script*

```
#!/bin/sh
#

# Check if the specified server is listening; return 1 if it is not
# $1 -- server name

is_listening()
{
   # Ping the server
   cdsping $ENCINA_TPM_CELL/server/$1 > $TMP_FILE 2>&1

   # Find out how many end points are listening
   POINTS=‘grep '_ip_' $TMP_FILE | wc -l | awk '{print $1}'‘
   LISTENING=‘grep ' is listening' $TMP_FILE | wc -l | awk '{print $1}'‘

   /bin/rm $TMP_FILE

   # Return success if at least one point is listening and none are not
   if [ "$POINTS" = "$LISTENING"  -a  ! "$LISTENING" = "0" ]; then
      return 0
   else
      return 1
   fi
}

# Wait for the specified server to come up
# $1 -- server name
# $2 -- what to wait for: is_running  or  is_listening
# $3 -- max number of attempts to check the server status

server_wait()
{
   # assume the server is down
   STATUS=1
   ATTEMPTS=0
   while [ $ATTEMPTS -lt $3  -a  $STATUS -eq 1 ]; do
      # Check the server status using the input method
      $2 $1
      STATUS=$?
      # wait for a while if it is not up yet
      if [ $STATUS -eq 1 ]; then
         sleep $SLEEP_TIME
      fi
      ATTEMPTS=‘expr $ATTEMPTS + 1‘
   done

   return $STATUS
}
```

*Figure 151.  (Part 2 of 4) Sample rc.encina.servers Script*

```
#
# MAIN
#

#
# Use the first Encina cell in the CELL_LIST
# unless ENCINA_TPM_CELL is already defined

if [ -z "$ENCINA_TPM_CELL" ]; then
   CONFIG_FILE=/opt/encinalocal/CELL_LIST
   if [ ! -f $CONFIG_FILE ]; then
      echo "No Encina cells running on this host"
      exit 1
   else
      CELL=‘head -1 $CONFIG_FILE | sed -e '/^#/d' | cut -d: -f2‘
      ENCINA_TPM_CELL=/.:/$CELL
      export ENCINA_TPM_CELL
   fi
fi

#
# Select action based on the first argument

case "$1" in
start)
        echo "\n##################\nServer Startup\n##################\n"

        #-----------------------------------------------
        # Start the servers in FIRST_SERVERS
        #-----------------------------------------------
        for server_name in $FIRST_SERVERS
        do
            echo "  >> Starting server $server_name..."
            enccp -c genericServer start $server_name
        done

        #-----------------------------------------------
        # Wait for all FIRST_SERVERS to come up
        #-----------------------------------------------
        for server_name in $FIRST_SERVERS
        do
            echo "  >> Waiting for server $server_name to come up..."
            # check up to 15 times if the server is running
            server_wait $server_name is_running 15

            # check if the server did start
            if [ $? -eq 0 ]; then
               echo "     Server $server_name is up and running."
            else
               echo "     WARNING: Server $server_name is NOT running yet."
            fi
        done
```

*Figure 152. (Part 3 of 4) Sample rc.encina.servers Script*

```
        #----------------------------------------------
        # Start each server group in the order specified
        #----------------------------------------------
        for server_set in $START_ORDER
        do
            echo "  >> Starting server set $server_set..."
            enccp -c serverSet start $server_set
        done

        echo "############################\nServer Startup Completed"
        echo "############################\n"

        ;;
stop)
        echo "\n##################\nServer Shutdown\n##################\n"

        #----------------------------------------------
        # Stop each server group in the order specified
        #----------------------------------------------
        for server_set in $STOP_ORDER
        do
            echo "  >> Shutting down server set $server_set..."
            enccp -c serverSet stop $server_set -gentleTimeout 5
        done

        echo "#############################\nServer Shutdown Completed"
        echo "#############################\n"
        ;;

*)
        echo "Usage: $0 { start | stop }"
        ;;
esac
exit 0
```

*Figure 153. (Part 4 of 4) Sample rc.encina.servers Script*

In addition to creating the Encina startup files, to ensure that the superuser of each machine has sufficient DCE privileges to start the Encina cell. By default, the superuser of a machine is granted privileges as the so-called self principal. For example, the superuser on machine prod_one is automatically granted a ticket as nodes/prod_one/self. The Encina cell is managed by the members of the encina_admin_group. Therefore, you have to add the self principal of each Encina machine to the encina_admin_group:

```
dce_login cell_admin
rgy_edit change -p node/prod_one/self -ng encina_admin_group.
```

Restart DCE on the machine to pick up the new credentials. Then open a new terminal window as the superuser on your machine, and check the DCE credentials (use the klist) to make sure that encina_admin_group is in the list of groups. Run the encina startup/shutdown script to test it:

```
rc.encina stop
```

The script, using the *stop* parameter, should shut down all servers on that machine and the Encina Node Manager. If the machine is running the Encina Cell Manager, it should be shut down too. Now run the script using the *start* parameter:

```
rc.encina start
```

All Encina processes on that machine should come up in the correct order.

Reboot the machine after you are satisfied with the performance of the rc.encina and rc.encina.servers scripts.

### 12.3.6 Encina Server Configuration

Configure the Encina servers by scripts, not with enconsole. Although enconsole makes it easier to configure servers and change their configuration, the configuration cannot be efficiently controlled. Imagine that you configured 25 different servers in your development environment and that you are now ready to move to your test (or staging) platform. All of a sudden you are faced with the problem of reproducing the configuration of 25 servers manually through enconsole. Apart from that being a tedious task, it is also very error prone.

We suggest that you use enconsole for configuring the servers in your development environment and experimenting with their configuration. Once you have decided on the parameters you want to set for all servers, you are better off creating an enccp script that creates the servers for you.

By running a script you can version control your server configuration and thus re-create a clean configuration in your production should you begin to doubt that a server has been misconfigured.

You can prepare the server configuration script in two ways. You can write the script yourself and then maintain it when new servers are added, old ones removed from the cell, and so on. Alternatively, you can put together a tool that extracts the server configuration of a given Encina cell and generates an enccp script for re-creating the cell (see Section 12.4, “Replicating Encina Cell Configuration” on page 343).

Figure 154 on page 343 shows some sample configuration code. Refer to the Encina documentation on enccp for more information about enccp syntax and capabilities. If you are running an older version of Encina, you can use the emadmin tool to do the job. Do not use emadmin for server configuration if you have enccp. Some incompatibilities between the two will jeopardize your effort.

```
monitorApplicationServer create OrderProcServer -verbose -attribute \
   {executable /home/encina/ops/bin/OrderProcServer} -attribute \
   {interfaces OrderProcIF} -attribute \
   {node prod_one}

physicalVolume create ordersRqsData_pvol -regions /dev/rdsk1/disk1
logicalVolume create ordersRqsDataVol -physicalVolumes
ordersRqsData_pvol

physicalVolume create ordersRqsLog_pvol -regions /dev/rdsk1/disk2
logicalVolume create ordersRqsLogVol -physicalVolumes
ordersRqsData_pvol

rqs create testRqs -verbose -attribute \
   {dataVolumes ordersRqsDataVol} -attribute \
   {logVolume ordersRqsLogVol} -attribute \
   {executable /usr/bin/rqs} -attribute \
   {node prod_two}
```

*Figure 154. Sample Server Configuration Code*

The DE-Light gateways described in Chapter 7, “Internet Access for Java Clients” on page 157 are configured similarly to any other Encina server. You can define them either as Toolkit servers through enconsole or, preferably, through the configuration scripts. You need to use the command line options -L and -X of the DE-Light gateway *drpcgwy* executable to specify the TRPC interfaces you would like to load into the gateway. For more information about configuring the DE-Light gateway, see the DE-Light documentation provided with the product.

## 12.4 Replicating Encina Cell Configuration

During deployment it is often the case that you have to replicate the configuration of a given machine (the original machine) to another machine (the target machine). For example, you have run your system tests in the staging environment and now you want to make sure that you configure the production environment in exactly the same way. Or, you may have some production problems and want to re-create the production environment somewhere else and then try to reproduce the problems.

There are three approaches to replicating the configuration of a given machine to another machine:

1. You can manually configure the target machine on the basis of the configuration of the original machine.
2. You can prepare a script that generates a configuration file for the original machine and another script that configures the target machine on the basis of the configuration file.
3. You can have a set of configuration scripts that you use to configure the original machine and apply those scripts to the target machine.

All three approaches are applicable to Encina applications. Some of them require more upfront cost (writing and testing scripts); others require more manual work during configuration, hence a higher chance of misconfiguration.

Our recommendation is to use a set of configuration scripts on your original and target machines. The advantage of this approach over the manual configuration is that the configuration scripts can be code controlled, thereby guaranteeing that you get the configuration you have envisioned. In addition the approach forces you to think about the target configuration well before the staging takes place, and it enables you to fall back to previous configurations should a problem occur with a new setup you are trying to implement.

The disadvantage of using configuration scripts is that it is hard to modify them. If you just want to change a single attribute you still have to go through the process of changing the script, submitting it to your code management system, fetching it, and releasing it. Although these activities may slow you down, they guarantee that your change has been recorded and will be looked at by at least one more person, the code management approver, thereby reducing the possibility of introducing errors when changing the configuration.

You can combine the manual and configuration scripts approach. For example, you can configure DCE manually, perform the initial Encina cell configuration manually, and use configuration scripts for the Encina servers, resource managers, and other objects. Be sure to include the manual steps of your configuration in a published configuration procedure so that anyone can redo it when necessary.

For our case study sample application, you can maintain the server configuration scripts described in Section 12.3.6, “Encina Server

Configuration” on page 342 under code control. Let’s say you have to change the priority of the server ordersRqs to 35. You lock server configuration file in your code management system, change the value of the priority attribute in the file, submit it to your code management system, have your approver agree on the change, and release the configuring file to the production environment. When you re-create the server, the attribute change takes effect.

As you can see the above approach can be tricky because you may have some data in your queues that you do not want to lose. It may also be quite tedious to maintain different scripts for the different servers, and you may not want to reconfigure all your servers just because you changed one attribute of one server.

To avoid these problems, you can use a variation of the configuration script approach. You can have two configuration scripts. One script describes the initial server configuration and is used when you first configure an Encina cell. The other script (the configuration update script) contains any extra changes you made to attributes that can be changed dynamically. You then apply the configuration update script to your running servers. In this way you do not lose data, and you may not even have to restart the servers.

You may have to further develop your set of configuration scripts according to the complexity of your application. The main point is, do not use enconsole to permanently change any server attributes. You are likely to forget what you have done, but even if you do not forget, the administrator doing the job after you will not even know about the changes.

# Chapter 13. Troubleshooting

Troubleshooting Encina applications may prove a daunting task given the complexity of distributed applications. Not only is the application spread over a number of machines but also the machines may run different operating systems and communicate through different protocols. In addition, the support responsibilities may be distributed, so different organizations and groups could be responsible for different machines and servers that are part of one application.

Encina offers several powerful facilities that enable operators and system administrators to monitor an Encina application. In this chapter we describe the Encina message log files and trace facility and explain how to debug Encina applications and troubleshoot some common DCE problems related to Encina. We also provide a general framework for troubleshooting Encina problems, present some of the most common problems encountered in Encina applications, and suggest how to deal with those problems.

## 13.1 Environment Setup

You need to ensure that your environment variables are set up correctly before you can do any monitoring and troubleshooting. The correct setup depends on your particular application support infrastructure. We recommend that you set the environment variables listed in Table 9 on page 347.

*Table 9. Encina Account Environment Variables*

| **Variable Name** | **Recommended Value** |
|---|---|
| ENCINA_TPM_CELL | /.:/<your Encina cell name> |
| PATH | Add the following directories to your existing PATH: /opt/encina/bin /opt/encina/etc /opt/dce/bin. |
| NLSPATH | Set according to the release notes. |
| LANG | Set according to the release notes. |
| SMVARS | Set according to the release notes. |

## 13.2 Overall Encina Cell Status

Before you delve into tracking down a problem, you must ensure that the Encina cell itself is installed, configured, and running properly. We cannot overemphasize how often severe system problems turn out to be rooted in improper installation and cell configuration.

Many parameters can be misconfigured or accidently altered in a complex Encina environment. We strongly recommend that you develop a well-thought-out procedure for checking the Encina cell configuration before the application deployment and perform the checks on a regular basis.

Ideally, you should prepare a script that compares the cell configuration to a “good” configuration profile which you generate on system deployment. In addition, you should have a script that checks the cell status. You can even configure the scripts to be run automatically and have their output sent to the operators and the administrators.

### 13.2.1 DCE and Encina Patch Levels

Make sure that the system is running the correct patch levels for DCE and Encina. You can find out the DCE patch level by examining the /opt/dce/PATCH.LEVEL file. Make sure that the patch is for the OS version of the examined machine.

The Encina patch level is determined by the fixes applied. You have to list all files in /opt/encina/fixes to find out the latest applied fix.

### 13.2.2 Cell Configuration

Although you can obtain the cell configuration by using enconsole, we recommend using the enccp tool. It will allow you to perform the checks faster, and you can use it to prepare configuration check scripts (see the beginning of Chapter 13.2, “Overall Encina Cell Status” on page 347 for more information about these scripts.)

The following commands provide you with the configuration of the Encina Cell Manager and the Encina Node Manager:

```
enccp -c ecm show
enccp -c enm show <node name>
```

You can use enccp to obtain a list of all servers and resource managers configured for your cell:

```
enccp -c genericServer list
```

Once you have determined which servers are configured for your cell you can look up their setup by running this enccp command for each server:

```
enccp -c genericServer show <server name>
```

Similarly, you can find out the configuration of the resource managers by running these commands:

```
enccp -c rm list
enccp -c rm show <resource manager name>
```

The output of these commands is quiet overwhelming. You are probably not interested in all the attributes. Table 10 on page 349 lists the attributes to pay attention to.

*Table 10. Important Encina Server Attributes*

| **Attribute** | **Description** |
|---|---|
| environment | List of all variables passed to the server |
| cdsPathName | Full CDS name of the server object |
| commandLineArgs | Command line arguments passed to the server |
| executable | Full executable name |
| interfaces | List of all interfaces supported by the server |
| maxStartupAttempts | Maximum number of startup attempts |
| name | Server's name |
| node | Node on which the server is running |
| paCount | Number of PAs for this server |
| principal | DCE principal used by the server DCE account |
| processPriority | Priority of server processes |
| type | Server type, for example, generic, DCE, MAS, RQS |
| userName | Operating system user running the server |
| authorizationLevel | DCE authorization required for the server |
| protectionLevel | Level of communication security protection |

## 13.2.3 ACL Setup

The Encina objects Access Control Lists (ACLs) are a common cause for concern. The most common ACL problems with Encina are associated with RQS queues, especially when clients access RQS servers from other Encina cells.

Generally, each client should be allowed to access the servers it uses, and each server using RQS should have access to the RQS server as well as to the RQS queues and queue sets it needs. For more information see *Encina 2.5 Administration Guide Volume 2: Basic Administration.*

All ACLs are manipulated through enccp. You can find out what an ACL looks like by running the following command within enccp:

```
enccp -c acl show <object>
```

In this example <object> is the full CDS name of the server, queue, and queue set you are interested in, for example,
/.:/orders/enc_prod/server/ordersRqs,
/.:/orders/enc_prod/server/ordersRqs/queue/PendingVerifyQueue.

The default CDS name used by Encina for an RQS queue <queue name> controlled by an RQS server <RQS server name> is:

```
<Encina cell name>/server/<RQS server name>/queue/<queue name>
```

You can find out which queues belong to a given RQS server by using rqsadmin:

```
rqsadmin list queues -server <Encina cell name>/server/<RQS server name>
```

## 13.2.4 Endpoint Map

One common problem with Encina applications is a polluted DCE endpoint map. This situation occurs when you change the IP address of a machine or when a machine can serve more than one IP address while Encina is using only some of them. These multiple IP addresses are usually used for enhanced system availability. Improper DCE and Encina configuration in such situations results in incorrect entries in the endpoint map, which leads to improper server binding.

A quick way to find out the IP addresses advertised in the endpoint mapper is to search for the word “binding” in the output of:

```
enccp -c endpoint show
```

Only the correct IP addresses should appear in the bindings.

## 13.2.5 DCE Processes and Servers

The /opt/dcelocal/etc/setup_state file (on AIX, the mkdce.data file) lists all DCE processes configured to run on your machine. You can find out which DCE processes are currently running by executing this command:

```
dce.ps (on Solaris)
ps -ef | grep dce (on AIX)
```

If all processes listed in the setup state file are indeed present and you suspect further DCE problems, refer to any DCE troubleshooting information you may have access to such as Transarc's DCE troubleshooting online guide:

```
http://www.transarc.com/afs/transarc.com/Public/Support/dce/trouble/trouble.html
```

### 13.2.6 Encina Nodes and Servers

You can obtain the current status of all Encina nodes and servers through enconsole. As soon as you bring up enconsole you will see a list of all servers and their status displayed to the left.

If the servers seem to be in the correct state then double check their availability by pinging them. Use the Ping menu item on the Actions menu to ping the cell manager, all nodes, and all servers.

You can also use enccp to check whether a particular server or Cell Manager is up and running. For example, you can check the Cell Manager for cell /.:/orders/enc_prod in this way:

```
enccp -c server ping /.:/orders/enc_prod/ecm
```

Similarly, you can ping the OrderProcServer server and node prod_one:

```
enccp -c server ping /.:/orders/enc_prod/server/OrderProcServer
enccp -c server ping /.:/orders/enc_prod/node/prod_one
```

## 13.3 Encina Message Log Files

Every application using Encina consists of several processes running on different machines. All Encina processes running under the Encina Monitor generate messages that are sent to the standard output. The standard output can be directed either to a file, which we refer to as a *message log file*, or to the Encina Cell Manager through a series of RPCs.

The default name of the Encina Cell Monitor message log file is ecm.log. The default name of the message log file for any other server and any Encina Node Manager is server.out.

The default location of a process message log file is the working directory of the process. The default working directories for the Encina Cell Manager, the Encina Node Manager, and the Encina application servers, respectively, are:

- /opt/encinalocal/<Encina cell name>/ecm
- /opt/encinalocal/<Encina cell name>/node/<node name>
- /opt/encinalocal/<Encina cell name>/server/<server name>

where <Encina cell name> is the name of the Encina cell, <node name> is the name of the node on which the Encina Node Manager is running, and <server name> is the name of the Encina application server.

For example, our sample application consists of an Encina cell named /.:/orders/enc_prod. The cell runs on machines prod_one and prod_two. Machine prod_one runs two applications servers: OrderProcServer, and VerificationServer. Machine prod_two has only one server items. It also runs the Encina Cell Manager. Both machines run an instance of the Encina Node Manager.

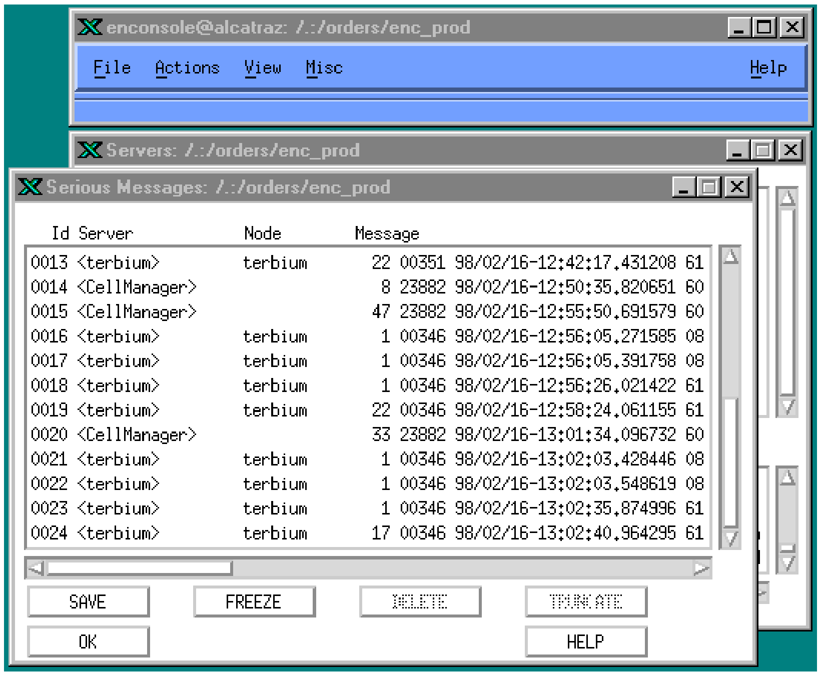
The default names of all message log files on machine prod_one are:

- /opt/encinalocal/orders/enc_prod/node/prod_one/server.out
- /opt/encinalocal/orders/enc_prod/server/OrderProcServer/server.out
- /opt/encinalocal/orders/enc_prod/server/VerificationServer/server.out

The default message log file names on machine prod_two are:

- /opt/encinalocal/orders/enc_prod/ecm/ecm.log
- /opt/encinalocal/orders/enc_prod/node/prod_two/server.out
- /opt/encinalocal/orders/enc_prod/server/ordersRqs/server.out
- /opt/encinalocal/orders/enc_prod/server/ordersPpc/server.out

The Encina message log files contain messages generated by the application code as well as by the Encina functions. All important messages generated by the Encina functions are automatically sent to the Encina Cell Manager and appear in its ecm.log message log file. All messages placed in the ecm.log file are also displayed in the Serious Messages window found under the View main menu item of enconsole (Figure 155 on page 353).


enconsole@alcatraz: /.:/orders/enc_prod
File Actions View Misc Help
Servers: /.:/orders/enc_prod
Serious Messages: /.:/orders/enc_prod
SAVE
FREEZE
DELETE
TRUNCATE
OK
HELP

```
 Id Server               Node         Message
0013 <terbium>            terbium      22 00351 98/02/16-12:42:17.431208 61
0014 <CellManager>                      8 23882 98/02/16-12:50:35.820651 60
0015 <CellManager>                     47 23882 98/02/16-12:55:50.691579 60
0016 <terbium>            terbium       1 00346 98/02/16-12:56:05.271585 08
0017 <terbium>            terbium       1 00346 98/02/16-12:56:05.391758 08
0018 <terbium>            terbium       1 00346 98/02/16-12:56:26.021422 61
0019 <terbium>            terbium      22 00346 98/02/16-12:58:24.061155 61
0020 <CellManager>                     33 23882 98/02/16-13:01:34.096732 60
0021 <terbium>            terbium       1 00346 98/02/16-13:02:03.428446 08
0022 <terbium>            terbium       1 00346 98/02/16-13:02:03.548619 08
0023 <terbium>            terbium       1 00346 98/02/16-13:02:35.874996 61
0024 <terbium>            terbium      17 00346 98/02/16-13:02:40.964295 61
```

*Figure 155. Serious Messages Window*

The information you can find in these message log files usually refers to unexpected events that have been captured by the Encina server generating the file. For example, when the Encina Node Manager on machine prod_one starts up server OrderProcServer, it writes a “Started Server OrderProcServer” message into the file:

`/opt/encinalocal/orders/enc_prod/node/prod_one/server.out`

Because this is an important event, a “Server OrderProcServer is now available” message appears on the Serious Messages window and it is stored in the file:

`/opt/encinalocal/orders/enc_prod/ecm/ecm.log.`

Some of the messages in the message log files and the Serious Messages window may contain an error code. If you want to find out more information about the error code issue this command:

```
translateError <Encina or DCE error code>
```

The command displays a description of the error that you specify on the command line. You can obtain an even more detailed explanation of your problem by issuing this command:

```
pdgquery <DCE error code>
```

Notice that the pdgquery facility only works with DCE error codes. It displays a description of the error along with suggestions on what you can do to deal with the problem.

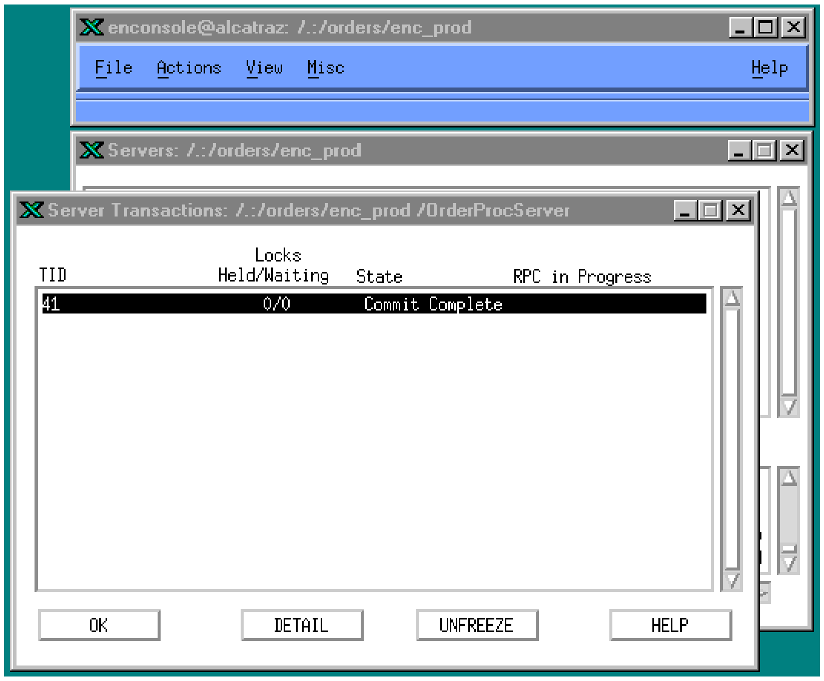
## 13.4 Transaction Status

The Encina Monitor coordinates the execution of transactions across several resource managers. It guarantees that no participating transaction is committed unless all participants have agreed to commit.

Managing distributed transactions, however, is quite complex because each participant may abort the transaction at any time. In addition, the communication between the Encina Monitor and the transaction participants may also be interrupted at any time.

Sometimes an error occurs during the second phase of the commit protocol, causing the Encina transaction to remain “hung” in a certain state for an extended period of time. In those (rare) situations you have to examine the “hung” transaction and find out who the transaction participants are. Then you have to abort manually all participating transactions as well as the Encina transaction.

You can determine the status of an Encina transaction that is executing by examining the Transaction Messages window under the Views main menu item of enconsole (see Figure 156 on page 355):


enconsole@alcatraz: /.:/orders/enc_prod
File Actions View Misc Help
Servers: /.:/orders/enc_prod
Server Transactions: /.:/orders/enc_prod /OrderProcServer
OK
DETAIL
UNFREEZE
HELP

```
TID                Locks
                   Held/Waiting   State            RPC in Progress
41                 0/0            Commit Complete
```

*Figure 156. Transaction Messages Window*

Alternatively, you can use the command line facilities provided with Encina to find out which transactions are running against a particular server, for example, server OrderProcServer:

```
enccp -c transaction list -server OrderProcServer
```

Furthermore you can determine the participants of a given transaction:

```
enccp -c transaction show <transaction id> -server <server name>
```

Once you have determined who the participants are, you can determine the status of their transactions, using the facilities provided by the database vendor. When all participating transactions have been cleared, you have to abort the Encina transaction itself:

```
enccp -c transaction abort <transaction id>
```

For more information about manipulating Encina transactions, consult the Encina manuals or run the following command:

```
enccp -c transaction help
```

## 13.5 Encina Trace Facility

Although the Encina message log files provide a lot of information about the events that occur within an Encina server, you may need to find out more about the server state during each of those events. For instance, you can find out that a server has terminated abnormally by looking at the Encina Node Manager's message log file, but you cannot determine why the server failed. The Encina Trace Facility maintains this kind of more detailed information. It provides tools that monitor error messages produced by the Encina servers.

When using the Encina Trace Facility, you have to decide which events you want to trace, in other words, the type of information you are looking for; select the output destination for the generated trace; and be able to use the tools provided for reading and interpreting the trace.

Always ensure that you are tracing all participants involved in the problem you are trying to resolve.

### 13.5.1 Selecting Trace Events

Encina supports several classes of events. Each event class is manipulated as a whole. When you turn on a given event class, all events of the class are recorded in the Encina trace. When you turn off an event class, information about all events of the disabled class are removed from the generated trace.

Serious (sometimes referred to as *critical*) events are always triggered and always appear in a trace. They are written to the server's message log file and displayed in the Serious Messages window by default (see Section 13.3, "Encina Message Log Files" on page 351).

The dump event class is used for low-level state information and is manipulated by using the tkadmin dump. More information about the dump event class can be found in the Encina support documentation. Always perform state dumps after collecting all other trace information from a running server because state dumps may sometimes result in a server failure.

All other event classes provide information about the execution path of the server. They are selectively triggered by using trace masks.

A trace mask is associated with every Encina component running in a server. It describes which event classes you want to trigger for that Encina component.

You specify a trace mask by listing the trace event classes that should be triggered for a particular component, for example:

```
trdce=basic+security
```

The above command indicates that all basic and security events for the trdce component should be triggered.

You can determine which components are running in a given server by using the following command:

```
tkadmin list trace -server <server name>
```

For example, you can find out the Encina components running in server OrderProcServer in this way:

```
tkadmin list trace -server OrderProcServer

Encina Executive:
       epm=0
       admin=default
       tran=trace_entry+trace_param
       trpc=default
       trdce=default
       threadTid=0
     Encina BDE:
       bde=0
     vendor_bpg:
       vendor_bpg=0
```

You can determine which classes are supported by a given component by using tkadmin, for example:

```
tkadmin query trace -server OrderProcServer trdce
```

For more information about the available components and their event classes, see the Encina documentation provided with the product, where you can also find a complete description of the trace masks. For our purposes, it is sufficient to say that you can describe the masks for the various components in two different ways, for example:

```
trdce=all,tran=basic
```

or

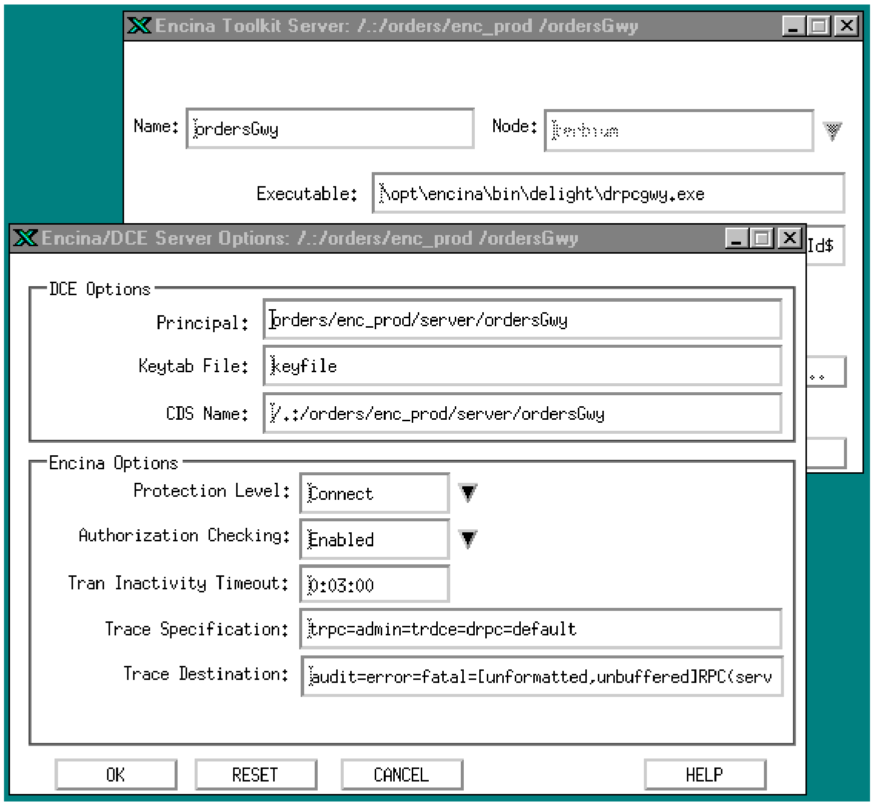
```
trdce=tran=xa=all
```

In the first example the masks of the components that we want to trace are presented as a comma separated list. In the second example we show how you can trigger the same event class for several components. Table 11 on page 358 presents the most common masks that we recommend for tracing Encina server problems.

*Table 11. Common Trace Masks*

| **Mask** | **Usage** |
|---|---|
| vol=log=all | Server startup problems |
| ots=all | Encina++ issues |
| tmxa=xa or<br>tmxa=all | XA-related issues;<br>Do not neglect the vendor’s logging facility. |
| trpc=trdce=all,tran=basic | The most generic mask one usually starts with;<br>good general starting point |
| all=all,bde=none | Entire trace; generates huge amounts of data |

You can set the trace masks in several different ways. We recommend using the enconsole tool to set the masks for the Encina server you want to monitor. Use the DCE Server Options window (Figure 157 on page 359) to alter the existing masks for a given server.


Encina Toolkit Server: /.:/orders/enc_prod /ordersGwy

Name: ordersGwy

Node: erbium

Executable: \opt\encina\bin\delight\drpcgwy.exe

Id$

Encina/DCE Server Options: /.:/orders/enc_prod /ordersGwy

DCE Options

Principal: orders/enc_prod/server/ordersGwy

Keytab File: keyfile

CDS Name: /.:/orders/enc_prod/server/ordersGwy

Encina Options

Protection Level: Connect

Authorization Checking: Enabled

Tran Inactivity Timeout: 0:03:00

Trace Specification: trpc=admin=trdce=drpc=default

Trace Destination: audit=error=fatal=[unformatted,unbuffered]RPC(serv

*Figure 157. DCE Server Options Window*

You can use the ENCINA_TRACE environment variable for setting the masks for a server started outside the Encina Monitor. The syntax for setting the variable in csh is:

```
setenv ENCINA_TRACE "trpc=tran=all,tmxa=xa"
```

If you are using ksh or Bourne sh, the correct syntax is:

```
ENCINA_TRACE="trpc=tran=all,tmxa=xa"
export ENCINA_TRACE
```

Regardless of how you change the trace masks, you need to restart the server for the change to take effect.

### 13.5.2 Selecting Output Destination

Trace data is always captured in the trace ringbuffer. The default size of the ringbuffer is 64 KB. We recommend changing it to a higher value (128 KB is usually sufficient) by using the ENCINA_TRACE_RING_SIZE variable. You have to restart the server for the new size to take effect.

The contents of the ringbuffer can be stored in a specified trace output destination. Because the contents are not immediately sent to the output destination, you have to examine them. Use the following command to examine the ringbuffers content:

```
tkadmin dump ringbuffer -server OrderProcServer dump_file
```

In this example the ringbuffer of server OrderProcServer is dumped into the dump_file file. You can also dump the ringbuffer automatically when exiting a particular server by setting the ENCINA_TRACE_BUFFER_DUMP_ON_EXIT environment variable to 1.

You can send the trace from the ringbuffer to any open file stream, such as stderr or stdout. Trace redirection, which you can do through enconsole, occurs for each trace event class (see Figure 157 on page 359).

The basix syntax for a trace output destination is described in Chapter 9, "Using the Trace Facility," in Transarc's *Encina Administration Guide Volume Two: Basic Administration*. It is fairly complicated and allows for a great deal of flexibility. The following trace output destination is used commonly:

```
all=[unformatted]FILE:trace.out
all=[unformatted]FILE:/home/tester/trace.out
```

These strings specify that all trace event classes are sent to a file named trace.out. The file resides in the server working directory unless it is fully qualified. The events are not formatted, which reduces the performance cost of having the event facility turned on. If you omit the [unformatted] string, the trace.out file would contain formatted trace output information.

The destination type must appear in uppercase. The most common destination types are:

- FILE destination, which refers to a local operating system file. The valid destinations for this class are complete or relative path names of a file.
- STREAM destination, which refers to well-defined standard I/O streams. The valid destinations are the standard output or standard error device (by default, the server.out file in the server's working directory).

**IMPORTANT**: On Windows NT, applications that do not have standard output or standard error devices cannot use the STREAM destination type. Such applications can use environment variable ENCINA_TRACE_REDIRECT to redirect trace output. The variable accepts any valid trace output destination.

- AIX destination, which refers to the local tracing and error logging facilities on the AIX operating system. The valid destinations for this type are trace and errlog.

You can list the trace destinations for all trace classes by using the tkadmin list trace command. The command syntax is:

```
tkadmin list redirect -server servername
```

For example, enter the following command to list the trace destinations of all trace classes for a server:

```
tkadmin list redirect
     entry:
     event:
     param:
     audit: [formatted,unbuffered]STREAM:stderr
     dump:
     error: [formatted,unbuffered]STREAM:stderr
     fatal: [formatted,unbuffered]STREAM:stderr
```

Alternatively, you can use the ENCINA_TRACE_REDIRECT environment variable for servers running outside the Encina Monitor, for example:

```
setenv ENCINA_TRACE_REDIRECT all=[unformatted]FILE:trace.out
```

## 13.5.3 Reading Trace Output

Regardless of how you obtained the trace for a given server it is always stored in compressed format. Once you have the trace file, you can convert the trace into human-readable form and then format it.

You need to format a trace file in several situations: when you explicitly redirect the enabled trace event classes to a trace file (see Chapter 13.5.2, “Selecting Output Destination” on page 359); when an automatic ringbuffer dump is performed; and when you manually force a ringbuffer dump.

If you force a ringbuffer dump or redirect the trace to a file, you have to convert that file into human-readable form. If the dump file was automatically created, it is stored in the server’s working directory under the name EncinaTraceBuffer.<pid>, where <pid> is the process identifier of the server.

Use this command to translate the trace file into human-readable form:

```
interpretTrace <file name> | indentTrace > trace.out
```

Each line of the trace contains the following fields: thread_id, time_stamp, trace_unique_id, and class. To interpret the error codes that appear in the trace, use this command:

```
translateError <error code>
```

You can also locate the origin of a trace message by using this command:

```
translateTraceId trace_unique_id
```

In addition, you can use hex2binary, translateTrpcAddress, and translateTranMessage to obtain further information about the messages you are getting in the trace.

# Appendix A.  Encina Codes and Messages

## A.1  Error Codes

`ENC-adm-0001 (0x70084001) ADMIN_AUTH_FAILURE`

Authorization failure

Explanation:

Permission for an administrative command has been denied.  Check the appropriate administrative access control list (ACL) to ensure that the necessary permission has been granted to the principal. Ensure that the appropriate principal is logged in and that the login context is valid (e.g. has not expired).

`ENC-sfs-0076 (0x7715a04c) SFS_COMMUNICATION_ERROR`

A communication error occurred.

Explanation:

An Encina transactional RPC (TRPC) or DCE RPC from the sfs client to the sfs server has failed.  This could be caused by a wide range of environmental problems, from a transient communication error to the termination of the server process.  Use default trpc tracing to identify the DCE exception caught and verify the validity of the binding used for the RPC.  Ensure that the client process has a valid login context and check the viability of the server, using `rpcutil ping <server>`.

`ENC-trc-1031 (0x7796a407) TC_RPC_FAILURE_CODE`

RPC failure

Explanation:

An Encina transactional RPC (TRPC) failure has been caught by the Tran-C library. This could be caused by a wide range of environmental problems, from a transient communication error to the termination of the server process. Use default trpc tracing to identify the DCE exception caught and verify the validity of the binding used for the RPC. Ensure that the client process has a valid login context and check the viability of the server, using `rpcutil ping <server>`.

`ENC-mon-1036 (0x74d3d40c) MON_AUTHZ_VIOLATION`

RPC rejected: client unauthorized

Explanation:

Permission to access a server manager function of a Monitor Application Server (MAS) has been denied. Check the access control list (ACL) for the specified interface or function to ensure that the necessary permission (x) has been granted to the principal. Ensure that the appropriate principal is logged in and that the login context is valid (e.g. has not expired).

`ENC-ema-0017 (0x71ae0011) EMA_CHANGES_FAILED`

Requested changes have failed.

Explanation:

The requested Encina Monitor Administration (EMA) command failed. For server start requests, anything that prevents the specified server from starting and initializing successfully will result in this error. Check the cell's serious event log and the server's output file for messages.

`ENC-vol-0007 (0x7857b007) VOL_DISK_PROTECTED`

User is not permitted to access the disk.

Explanation:

The application was unable to read from or write to the specified disk device. Ensure that appropriate permissions have been granted for the device and that the application is running under the proper identity to access the device.

`ENC-trp-0010 (0x7797700a)` TRPC_UNBOUND_TRAN_HANDLE

Could not get a fully bound transactional handle

Explanation:

The transactional RPC (TRPC) has failed because a fully bound transactional handle could not be generated. This can be caused by a null or invalid TRPC handle, or the failure of the DCE RPC to the target server. Check the client's default trpc tracing for the underlying cause (usually, a DCE status or exception is identified as the root cause).

`ENC-tra-1135 (0x7796846f)` TRAN_ABORT_NO_SUITABLE_COORDINATOR

All acceptable applications refused to coordinate.

Explanation:

The transaction has been aborted because no suitable participant agreed to coordinate commitment of the transaction. A distributed transaction requires at least one recoverable participant to fill this role.

`ENC-bde-0019 (0x706bc013)` BDE_INVALID_PRIORITY

An invalid priority value was specified.

Explanation:

The requested priority could not be set when starting a process. This commonly occurs when the Encina Node Manager (enm) is not running as root and the server to be started has specified a higher priority than that under which the Node Manager process is running. The Node Manager must either be run as root or all processes started by it must specify a priority equal to or lower than the enm process.

`ENC-vol-0016 (0x7857b010) VOL_INVALID_NAME`

Client-provided volume or disk name is invalid.

Explanation:

An incorrect name has been specified to the Encina Volume Service (VOL). The most common cause is that the specified physical device or file volume does not exist, or that the specified logical volume name has not been created.

`ENC-sfs-0085 (0x7715a055) SFS_INSUFFICIENT_FILE_SYSTEM_ACCESS_RIGHTS`

Insufficient file system privilege for requested operation.

Explanation:

Permission to access the Encina Structured File Server (SFS), or one of its files, has been denied. Check the appropriate access control list (ACL) for the server or file to ensure that the necessary permission has been granted to the principal. Ensure that the appropriate principal is logged in and that the login context is valid (e.g. has not expired).

`ENC-sfs-0060 (0x7715a03c) SFS_OPERATION_TIMED_OUT`

Timeout expired before operation completed.

Explanation:

The requested Encina Structured File Server (SFS) operation timed out before completion.  This could be caused by a long-running operation or by a conflict (e.g. a transactional lock conflict) that prevents the operation from completing.  Increase the operation timeout for the request or resolve the conflict.

`ENC-log-0258 (0x746f6102) LOG_NO_SPACE`

Log volume is out of space.

Explanation:

An Encina Log Service (LOG) write request failed because the log is out of free space.  This can be caused by insufficient log space to support the necessary transaction rate, an insufficient checkpoint interval, or long-running transactions that prevent the log tail from advancing to free up log space. Before this error, the following warning is issued, indicating that log compression is occurring:

Compressing data on log volume <volume> for space reclamation.

Log compression is often accompanied by noticeably higher CPU utilization. If compression does not complete before the log is full, or if enough space is not reclaimed to support the current transaction rate, the above error will result.  Expand the log volume to provide enough log space to restart the server and, if necessary, take administrative action to force the resolution of any unresolved transactions.

`ENC-bde-0018 (0x706bc012) BDE_INVALID_PATH`

The specified file does not exist.

Explanation:

The specified path is invalid. This usually indicates that the path to a server executable has been incorrectly specified. Check the path specified for the server configuration and that the Encina Node Manager has permission to access the path.

`ENC-mon-0001 (0x74d3d001) MON_CELL_UNAVAILABLE`

Could not communicate with the Cell Manager.

Explanation:

An Encina Monitor application (usually, a client) failed to contact the Encina Cell Manager (ecm). This could be caused by a wide range of environmental problems, from a transient communication error to the termination of the server process. Use default trpc tracing to identify the DCE exception caught and verify the validity of the binding used for the RPC. Check the viability of the server using, `rpcutil ping <server>`.

`ENC-tra-1065 (0x77968429) TRAN_ABORT_COORDINATOR_MIGRATION_FAILURE`

Coordinator migration timed out.

Explanation:

The transaction has been aborted because coordinator migration timed out. By default, this message is attempted five times, delaying for 10 seconds between attempts. If all participants are believed to be running and reachable, use "trpc=all,tran=basic" tracing to determine the underlying cause of the RPC failure.

`ENC-ema-0014 (0x71ae000e) EMA_CALLER_NOT_AUTHORIZED`

Caller is unauthorized to perform the operation.

Explanation:

Permission for an Encina Monitor Administration (EMA) command has been denied. Check the administrative access control list (ACL) for the specified monitor cell to ensure that the necessary permission has been granted to the principal.  Ensure that the appropriate principal is logged in and that the login context is valid (e.g. has not expired).

ENC-trp-0029 (0x7797701d) TRPC_RPC_FAILED

RPC failed for unknown reasons (most likely that DCE cannot pass right status).

Explanation:

An Encina transactional RPC (TRPC) failed for unknown reasons. This is the generic "RPC failure" status code for TRPC. This status is returned when TRPC cannot successfully convert the exception caught into an appropriate corresponding status. It is also used by the Recoverable Queuing Service (RQS) when any exception is caught.  Use default trpc tracing to identify the exception caught.

ENC-tra-1064 (0x77968428) TRAN_ABORT_PREPARE_INFERIORS_TIMEOUT

The prepare phase timed out.

Explanation:

The transaction has been aborted because the prepare phase timed out. By default, this message is attempted five times, delaying for 10 seconds between attempts.  If all participants are believed to be running and reachable, use "trpc=all,tran=basic" tracing to determine the underlying cause of the RPC failure.

ENC-ppc-0016 (0x7601a010) PPC_CONN_FAILURE_NO_RETRY

Connection failure, no retry

Explanation:

The allocate request failed because Encina PPC was unable to establish a SNA connection. This is commonly caused by a mismatch between the "prof_name" entry of the side_info profile and the "local_lu_name" of the local_lu_lu6.2 profile. Refer to the appropriate SNA configuration documentation and the PPC section of the Encina Administration Guide Volume 2 (Server Administration) for more information.

`ENC-log-0256 (0x746f6100) LOG_VOL_ERROR`

Error encountered while operating on a log volume.

Explanation:

An error occurred while reading from or writing to a log volume. This typically indicates that the Encina Volume Service (VOL) encountered a hard I/O error. Use the default VOL tracing to identify the underlying error.

`ENC-ema-0016 (0x71ae0010) EMA_CELL_UNAVAILABLE`

Failed to communicate with the Cell Manager

Explanation:

An Encina Monitor Administration (EMA) command failed because it could not contact the Encina Cell Manager (ecm). This could be caused by a wide range of environmental problems, from a transient communication error to the termination of the server process. Use default trpc tracing to identify the DCE exception caught and verify the validity of the binding used for the RPC. Check the viability of the server, using `rpcutil ping <server>`.

`ENC-eai-0015 (0x7190800f) ENCONSOLE_SERVER_EXITED`

Server process exited prematurely

Explanation:

An attempt to verify that a process exists failed, indicating that the process terminated unexpectedly.  Check the cell's serious event log and the server's output file for messages indicating why the process terminated.

## A.2  Messages

### A.2.1  Monitor

*****************************************************************************

ID: 0x60e43c16

Component: Monitor

Message: "ecm: Failed to create cell object : %k."

Variables: %k - Error status code.

Explanation:

Based on the error code. If the following error statuses get returned during a cold start of the cell, they indicate an internal error and should be reported to Transarc as such. If they occur during a warm start, they indicate corruption of the data volume.

```
EMA_TYPE_DOES_NOT_EXIST
EMA_OPERATION_PROHIBITED
EMA_TYPE_SPEC_INCONSISTENT
EMA_VALTYPE_MISMATCH
EMA_VALTYPE_INVALID
```

The following error statuses are due to user errors:

`EMA_NAME_INVALID` - Either a space in the cell name or name was greater than 128 characters.

`EMA_OBJECT_ALREADY_EXISTS` - The user is trying to define another cell with the same name in the present cell's repository.

`EMA_ATTRIBUTE_DOES_NOT_EXIST` - This attribute does not exist i.e. is not an instance of ema_typeAttribute. Check the list of attribute names specified against the list specified in the manual.

EMA_ATTRIBUTE_NOT_APPLICABLE - The attribute in question does not exist for the object type (ema_typeCell) being created. Again go through the attribute list for the cell.

EMA_ATTRIBUTE_MISSING - A required but undefaulted attribute has not been specified. Again go through the manual to make sure that you have specified all required, undefaulted attributes.

******************************************************************************

ID: 0x60e43c26

Component: Monitor

Message: "ecm: Failed to retrieve cell object : %k."

Variables: %k - Error status code.

Explanation: Based on the error code. The following error statuses point to an Encina internal error and should be reported to Transarc as such:

EMA_LOCK_MODE_INVALID
EMA_LOCK_NOT_AVAILABLE

This public fatal is more likely to occur during a warm start when the cell attributes are being retrieved. It indicates a possible corruption of the data volume.

EMA_TYPE_DOES_NOT_EXIST
EMA_OPERATION_PROHIBITED
EMA_OBJECT_DOES_NOT_EXIST
EMA_ATTRIBUTE_NOT_APPLICABLE
EMA_ATTRIBUTE_NOT_PRESENT
EMA_VALTYPE_INVALID
EMA_ATTRIBUTE_DOES_NOT_EXIST

******************************************************************************

ID: 0x60e43c36

Component: Monitor

Message: "ecm: Cell name (%s) has been changed to (%s)"

Variables: %s - Old cell name, %s - New cell name

Explanation:

The cell name was changed before warm start, indicating corruption of the data volume.

*******************************************************************************

ID: 0x60e44416

Component: Monitor

Message: "ecm: Could not initialize data volume : %s."

Variable: %s - Reason for abort

Explanation:

Based on reason for abort. AREA_VOLUME_TOO_SMALL - Data volume size is too small. Increase volume size and retry. AREA_VOL_IN_USE - The volume is already in use. Use a new volume for this cold start.

*******************************************************************************

ID: 0x61140416

Component: Monitor

Message: "ecm: Failed to create attribute %s: %k"

Variables: %s - Name of erroneous attribute, %k - error status code.

Explanation:

If it returns an EMA error status, then it points to an internal error not a public fatal. If the error is reported from either ROS, AREA or REC then it points to a problem with the data volume. The data volume is either corrupt or out of space and needs to be enlarged.

*******************************************************************************

ID: 0x61140816

Component: Monitor

Message: "ecm: Failed to create type %s : %k \n"

Variables: %s - Name of type, %k - error status code

Explanation:

If it returns an EMA_ error status, it points to an internal error not a public fatal. If the error status is reported from either ROS, AREA or REC then it points to a problem with the data volume. The data volume is either corrupt or out of space and needs to be enlarged.

********************************************************************************

ID: 0x61141016

Component: Monitor

Message: "ecm: Failed to verify parent of type %s.\n"

Variables: %s - Name of the parent type

Explanation:

Errors occurs at warm start only and indicates corruption of a Cell Manager data volume.

********************************************************************************

ID: 0x61141026

Component: Monitor

Message: "ecm: Failed to find required attr %s in type %s.\n"

Variables: %s - Name of the required attribute, %s - Name of the type

Explanation:

Error occurs at warm start only and indicates corruption of a Cell Manager data volume.

*****************************************************************************

ID: 0x61141036

Component: Monitor

Message: "ecm: Failed to find optional attr %s in type %s.\n"

Variables: %s- Name of the optional attribute, %s -Name of the type

Explanation:

Error occurs at warm start only and indicates corruption of a Cell Manager data volume.

*****************************************************************************

ID: 0x61141046

Component: Monitor

Message: "ecm: Failed to verify type %s : %k \n"

Variables: %s - Type name, %k - error status code

Explanation:

Errors occurs at warm start only and indicates corruption of a Cell Manager data volume.

*****************************************************************************

ID: 0x61140c16

Component: Monitor

Message: "ecm: Failed to verify attribute %s (%d)\n"

Variables: %s - Name of erroneous attribute, %k - error status code

Explanation:

Error occurs at warm start only and indicates corruption of a Cell Manager data volume.

*******************************************************************************

ID: 0x61140816

Component: Monitor

Message: "ecm: Failed to create type %s: (%k)"

Variables: %s - type name, usually "ema_typeCustomType", %k - error status code.

Explanation:

If it returns an EMA_ error status, it points to an internal error not a public fatal. If the error is reported from either ROS, AREA or REC then it points to a problem with the data volume. The data volume is either corrupt or out of space and needs to be enlarged.

*******************************************************************************

ID: 0x60e4401c

Component: Monitor

Message: "ecm: Data volume name is required at cold start."

Variables: NONE

Explanation: The data volume name was not specified as a command line argument for ecm.

*******************************************************************************

ID: 0x61a01c2c

Component: Monitor

Message: "ecm: Failed to open serious event log file %s: %k"

Variables: %s - Serious event file name, %k - status code

Explanation:

Points to problems in opening a file such as insufficient privileges or insufficient disk space or network access problems.

******************************************************************************

ID: 0x61303c17

Component: Monitor

Message: "Cell name (%s) should be fully qualified."

Variables: %s - CDS name of cell

Explanation:

The cell name was not specified correctly as one of either `/.../<cell-name>` or `/.:/<cell-name>`

******************************************************************************

ID: 0x61481836

Component: Monitor, Node Manager

Message: "enm: Failed to secure handle to the ecm: %k."

Variables: %k - status code returned from DCE call.

Explanation:

The node manager tried to stamp authentication information into the binding handle used to communicate with the Cell Manager and failed.

User Response:

Could be due to a DCE problem or to the node not possessing a valid DCE login context. Check the error code and make sure that DCE is running properly and the Node Manager is running as a valid DCE principal.

*******************************************************************************

ID: 0x61481816

Component: Monitor, Node Manager

Message: "enm: Failed to get startup info from cell-mgr: %k"

Variables: %k - Status code returned from call to Cell Manager interface to retrieve node info.

Explanation:

The call to the Cell Manager interface to retrieve the node info failed.

User Response:

First check that the Node Manager is defined in the repository and then check that its desired state is set to running via an node start.

*******************************************************************************

ID: 0x61483436

Component: Monitor, Node Manager

Message: "enm: keyfile specified is invalid."

Variables: NONE

Explanation:

Could not validate the key file for the node principal after obtaining node startup options because:

1. The keyfile does not exist.

or

2. The Node Manager does not have permissions to open the file.

or

3. The requested key is not present for the Node Manager principal.

User Response:

Check that the startup options specifying the node principal name and keyfile are correct and that the keyfile has been created and has the correct permissions.

********************************************************************************

ID: 0x61484426

Component: Monitor, Node Manager

Message: "enm: Failed to notify cell manager of availability: %k."

Variables: %k - status code. Could be a DCE status code, indicating a problem with the RPC, or an EMA status code, indicating a problem with the Cell Manager.

Explanation:

1. The RPC to the Cell Manager notifying it of the node manager's availability did not succeed.

or

2. The node does not exist in the repository.

or

3. The node has not been started i.e. its desiredState is not RUNNING.

User Response:

Check that DCE and the Cell Manager are running, there is an entry for the Node Manager in the repository, and the node has been started.

********************************************************************************

ID: 0x61d40416

Component: Monitor, migrate

Message: "Usage: %s <aclAttrList>"

Variables: %s - name of the executable (addacl)

Explanation:

Incorrect number of arguments.

User Response:

Provide required attribute list argument.

*******************************************************************************

ID: 0x61d40426

Component: Monitor, migrate

Message: "Invalid attribute list format"

Variables: NONE

Explanation:

Argument is not a valid attribute list.

User Response:

Fix syntax errors in the input and retry. Attribute list syntax is specified in the *Encina Monitor Administrative Programmer's Guide and Reference*.

*******************************************************************************

ID: 0x61d40436

Component: Monitor, migrate

Message: "ENCINA_TPM_CELL environment variable must be set."

Variables: NONE

Explanation:

The ENCINA_TPM_CELL environment variable was not set.

User Response:

Rerun the program after setting the ENCINA_TPM_CELL environment variable.

******************************************************************************

ID: 0x61d40446

Component: Monitor, migrate

Message: "addacl failed: %k"

Variables: %k - DCE status code. Could indicate problems obtaining a handle to the Cell Manager, problems in obtaining the UUID of the ACL manager, or problems updating the ACL.

Explanation:

The addition of the ACL failed due to any one of a variety of reasons (see the variable entry).

User Response:

Check that the Cell Manager is up and DCE is running correctly. If SEC_INSUFFICIENT_MEMORY is returned, try running the program after stopping other programs. For other problems notify Transarc.

******************************************************************************

ID: 0x60d01816

Component: Monitor, qrf

Message: "Cannot create element type (%k)"

Variable: %k - RQS status code.

Explanation:

Tried to create an RQS element and failed.

User Response:

Note the error code and inform Transarc.

*******************************************************************************

ID: 0x617c1c36

Component: Monitor, server run time

Message: "Server startup environment not present."

Variables: NONE

Explanation:

The ENCINA_TPM_STARTUP_DATA environment variable was not set. Normally it is set by the Node Manager process before the server process is forked off. If the server is being started in debug mode, the user must set the variable.

User Response:

Set the ENCINA_TPM_STARTUP_DATA environment variable and restart the server if it is being started in debug mode. Otherwise inform Transarc.

*******************************************************************************

ID: 0x617c1c2c

Component: Monitor, server run time

Message: "Server startup environment '%s' is invalid."

Variables: %s - the value of the ENCINA_TPM_STARTUP_DATA environment variable.

Explanation:

1. ENCINA_TPM_STARTUP_DATA environment variable is not set to a valid attribute list.

or

2. The variable does not specify the required cellName, nodeName, serverName or paNumber attributes.

User Response:

Ensure that the ENCINA_TPM_STARTUP_DATA environment variable is set to the correct value and restart the server only if server is being restarted in debug mode. Otherwise inform Transarc.

*******************************************************************************

ID: 0x617c201c

Component: Monitor, server run time

Message: "%s: Failed to get handle to enm: %k."

Variables: %s - trace id, %k - DCE status.

Explanation:

Server could not bind to Node Manager.

User Response:

Check that the Node Manager is up. If server is being started in debug mode, confirm whether the Node Name was specified correctly in ENCINA_TPM_STARTUP_DATA and restart the server.

*******************************************************************************

ID: 0x617c202c

Component: Monitor, server run time

Message: "Server was not started or PA 0 initialization has failed."

Variables: NONE

Explanation:

The server could not retrieve its startup information from the Node Manager. This could be due to a problem in making the RPC to the node manager, the node manager not being able to find the server object in the repository, or a problem with PA 0 initialization.

User Response:

To narrow the problem, retrieve the status code from the trace file from the trace entry with the "0x617c2027: Failed to get startup info from enm. (%k)" message.

******************************************************************************

ID: 0x617c281c

Component: Monitor, server run time

Message: "%s: Pings to the enm have failed."

Variable: %s - trace id.

Explanation:

The server failed to send an "I'm alive" message to the Node Manager, the server does not exist in the repository, or the server has been stopped.

User Response:

Ensure that the Node Manager is running and restart the server.

******************************************************************************

ID: 0x617c6c1c

Component: Monitor, server run time

Message: "%s: Failed to notify the node manager:%k".

Variables: %s - trace id, %k - status code.

Explanation:

The server was ready to start listening for RPCs but could not successfully notify the Node Manager. This could be due to either the node manager having terminated prematurely, the node manager being unable to find the server object in the repository, or the server has been stopped administratively.

User Response:

Examine the error code to determine whether the Node Manager needs to be restarted or the repository needs to be updated and restart server.

******************************************************************************

ID: 0x6050d81c

Component: Monitor, server run time

Message: "Cell name (%s) should be fully qualified."

Variable: %s - the encina cell name that was specified.

Explanation:

The name of the encina cell was incorrectly specified. it must be either `/.../<dceCellName>/<cellName>` or `/.:/<cellName>`

User Response:

Specify the encina cell name in the correct format.

******************************************************************************

ID: 0x61d00416

Component: Monitor, migrate

Message: "usage: %s <SFS server name> <cell name> [node name [del]]"

Variables: %s - the program name.

Explanation:

The program was run with an incorrect number of parameters.

User Response:

Rerun the program with the correct number of parameters.

******************************************************************************

ID: 0x61d00c16

Component: Monitor, migrate

Message: "Cell configuration record not found: %k"

Variables: %k - Status code.

Explanation:

Could not retrieve the cell configuration record from the SFS server.

User Response:

Make sure the cell name argument was specified correctly to the program. Check that the SFS server is up, there is a cell configuration record in the SFS file, and the SFS file has not been corrupted.

********************************************************************************

ID: 0x61d0241c

Component: Monitor, migrate

Message: "Couldn’t initiate ACL scan: %k"

Variables: %k - status code

Explanation:

The ACL scan could not be initiated at the SFS server.

User Response:

The error could be due to a DCE problem, the SFS server being down, the user principal not being authorized at the SFS server, or the SFS file being corrupted.  Examine the error code to take appropriate action.

********************************************************************************

ID: 0x61d02426

Component: Monitor, migrate

Message: "ACL Scan terminated abnormally: %k"

Variables: %k - status code

Explanation:

The program could not complete the ACL scan.

User Response:

The error could be due to a DCE problem, the SFS server terminating unexpectedly, or the SFS file being corrupted. Examine the error code to take appropriate action.

******************************************************************************

ID: 0x61d0281c

Component: Monitor, migrate

Message: "Node configuration record not found: %k"

Variables: %k - status code

Explanation:

Could not retrieve the node configuration record from the SFS server.

User Response:

Check that the node name argument was specified correctly to the program, the SFS server is up, and the SFS file has not been corrupted.

******************************************************************************

### A.2.2 OTS

ID: 0xd41fe8bc

Component: Encina++, DDL compiler

Message: "Could not open \"%s\"\n"

Variables: %s - The name of the output file

Explanation:

While processing the DDL file, an output file for the generated class could not be created or opened.

User Response:

Check that the user has permission to create new files or that there is enough free space in the destination file system.

******************************************************************************

ID: 0xd41fd03c

Component: Encina++, DDL compiler

Message: "The ddl must specify a primary index"

Variables: none

Explanation:

A primary index was not specified in the record being processed by the DDL compiler.

User Response:

Add a primary index to the record.

******************************************************************************

ID: 0xd41fdc1c

Component: Encina++, client

Message: "TPM cell name (ENCINA_TPM_CELL) must be set"

Variables: NONE

Explanation:

The name of the cell is required for correct operation of a client application but the name was not specified.

User Response:

Set the ENCINA_TPM_CELL environment variable to the name of the cell before executing the client application.

******************************************************************************

ID: 0xd41fe06c

Component: Encina++, Monitor client

Message: "mon_InitClient() failed: %k"

Variables: %k - The return code from `mon_InitClient` function converted into a string.

Explanation:

A call on the `mon_InitClient` function failed with the MON_PROTOCOL_ERROR return code. The reason was a duplicate attempt to initialize the client application.

User Response:

Correct the user-written code to ensure that the client is initialized only once.

******************************************************************************

ID: 0xd41fe0b6

Component: Encina++, Monitor client

Message: "Inter cell binding not supported when binding by interface"

Variables: NONE

Explanation:

An attempt was made to bind to another cell (by calling the Bind function explicitly with the cell name as an argument) but the binding mode was interface-based binding. This form of binding is not supported.

User Response:

Change the code to use another binding mode.

******************************************************************************

ID: 0xd41ff81c

Component: Encina++

Message: "Illegal call to join a thread that was not active"

Variables: NONE

Explanation:

A programming error has occurred. The user's code attempted to join a thread that was not active.

User Response:

Correct the code or debug the application as there may be a reason why the thread was not active when an attempt was made to join it.

******************************************************************************

ID: 0xd41fe03c

Component: Encina++, Orbix server

Message: "No server name specified for Persistent Orbix server, set the server name via ENCINA_OTS_TK_SERVER_ARGS env. variable or the -encina command line switch."

Variables: NONE

Explanation:

The server was registered as a "persistent" server with the Orbix daemon, but the server name was not specified in the server startup information.

User Response:

Specify the server name in the startup information. The startup information is a comma-separated string of name=value pairs that you can specify through the -encina <startup info> command-line switch and argument or the ENCINA_OTS_TK_SERVER_ARGS environment variable. The server name is specified within the string as "serverName=<name>".

******************************************************************************

ID: 0xd41fe46c

Component: Encina++, server

Message: "For a cold start the ENCINA_OTS_TK_SERVER_ARGS environment variable, or the \"-encina <encinaArgs>\" command line argument, must specify the name of a device for the log.");

Variables: NONE

Explanation:

The server was *cold started* but the name of the log device on which the log should be configured was not found in the server startup information.

User Response:

Specify the name of the log device in the startup information. The startup information is a comma-separated string of name=value pairs that you can specify through the -encina <startup info> command-line switch and argument or the ENCINA_OTS_TK_SERVER_ARGS environment variable. The server name is specified within the string as "logDevice=<name>".

******************************************************************************

ID: 0xd41fc81c

Component: Encina++, server

Message: "Invalid encina argument specification (incorrect form)."

Variables: NONE

Explanation:

The server startup information is a comma-separated string of name=value pairs that you can specify through the -encina <startup info> command-line switch and argument or the ENCINA_OTS_TK_SERVER_ARGS environment variable. While processing the string was found to be malformed.

User Response:

Correct the format of the startup information string so that it contains valid name=value pairs separated by commas.

****************************************************************************

ID: 0xd41fd02c

Component: Encina++, server

Message: "For a recoverable server the ENCINA_OTS_TK_SERVER_ARGS environment variable, or the "-encina <encinaArgs>" command line argument, must specify the name of the restart files."

Variables:NONE

Explanation:

The server was unable to configure the log service because the restart information for the volume on which the log service's volume is configured was not specified in the server startup information.

User Response:

Specify the restart file names in the startup information. The startup information is a comma-separated string of name=value pairs that you can specify through the -encina <startup info> command-line switch and argument or the ENCINA_OTS_TK_SERVER_ARGS environment variable. The restart files should be specified as a "restart string" that consists of a pair of colon-separated (semi-colon-separated on NT) names where the restart state will be (or has been) stored. An example is: "restartString=restart:restart.bak".

******************************************************************************

### A.2.3 PPC

ID: 0x74182c76

Component: PPC, ppcgwy

Message: "Unable to set principal (%s): %k"

Variables: %s - This string specifies principal Name. PPC gateway executes in the login context of this principal Name. %k - Results of an attempt to store principal name.

Explanation:

The PPC gateway was unable to allocate enough memory for storing the principal name.

User Response:

Make sure there is enough memory for the PPC gateway.

******************************************************************************

ID: 0x74182c86

Component: PPC, ppcgwy

Message: "Unable to set key file (%s): %k"

Variables: %s - This string specifies Key file name. Password for PPC gateway principal is obtained from this Key file. %k - Results of an attempt to store key file name.

Explanation:

The PPC gateway was unable to allocate enough memory for storing the key file name.

User Response:

Make sure there is enough memory for the PPC gateway.

******************************************************************************

ID: 0x74182c96

Component: PPC, ppcgwy

Message: "Unable to decode protection level (%s): %k"

Variable: %s - This is string specifying PPC gateway protection level. PPC gateway decodes the string to get the protection level value. %k - The result of an attempt to translate the protection level string to protection level Value.

Explanation:

Wrong protection level string was given.

User Response:

Check documentation to get the right protection level strings.

******************************************************************************

ID: 0x74182ca6

Component: PPC, ppcgwy

Message: "Invalid trace specification -- \%s\"

Variable: %s - This string indicates the direction in which all the PPC gateway trace messages are redirected.

Explanation: Format of the user-specified redirection specification is wrong.

User Response:

Check documentation to get the right redirection specification.

***************************************************************************

## A.2.4  Client Core

ID: 0x84048816

Component: Admin command line interface

Message: "Unrecognized argument name: \"%s\""

Variable: %s - The unrecognized argument.

Explanation:

A switch in the command line is not recognized.  An example is mistyping "-server" switch as "-sever".

User Response:

Check the command for typos.

*************************************************************************

ID: 0x84048836

Component: Admin commmand line interface

Message: "Ambiguous argument name: \"%s\""

Variable: %s - The ambiguous argument.

Explanation:

The command line parser cannot uniquely determine a partial match of a given switch.  For example, if "-logvolume" and "-logfile" switches were both valid for a command, a given switch "-log" would trigger this message.

User Response:

Use full name of the switch.

*************************************************************************

ID: 0xd8200816

Component: TRDCE

Message: "Key management error is not recoverable."

Explanation:

A DCE security key management function has returned an error that cannot be recovered with retry. For example, the application is not authorized to perform the operation.

User Response:

Make sure DCE and the environment are set for key management to work.

******************************************************************************

ID: 0x280c0836

Component: TRPC

Message: "The runtime library being used does not match the version of TIDL used.\nInterface name %s, function name %s, stubs version %d."

Variables: %s - Interface name of the RPC call. %s - Function name of the RPC call. %d - Version of TIDL used.

Explanation:

TRPC run time detected that a TRPC stub file was generated by an unmatching TIDL compiler.

User Response:

Make sure your TIDL files are compiled by a TIDL matching the TRPC run time.

******************************************************************************

ID: 0x280c2057

Component: TRPC

Message: "The TRPC runtime library does not match compiled manager stubs.\nInterface name %s, function name %s, stubs version %d.\n"

Variable: %s - Interface name of the RPC call. %s - Function name of the RPC call. %d - Version of stub.

Explanation:

TRPC run time detected that a manager stub file was generated by an unmatching TIDL compiler.

User Response:

Make sure your TIDL files are compiled by a TIDL matching the TRPC run time.

******************************************************************************

ID: 0x28106056

Component: Client TRPC

Message: "Runtime library could not acquire a string binding for a well-known endpoint (%k)."

Variable: %k - Error code returned by an RPC string binding function.

Explanation:

A well-known endpoint binding specified through trpc_UseWkEndpoints or trpc_BindWkEndpoints is invalid. Most likely, it has been incorrectly specified, it has been corrupted, or sufficient memory could not be allocated to return the string binding.

User Response:

Review the application’s use of these functions or ensure that the process has not exceeded its memory limit and that sufficient swap space exists on the machine, depending on the error returned.

******************************************************************************

ID: 0x28106066

Component: TRPC

Message: "Runtime library could not parse the string binding for a well-known endpoint (%k)."

Variable: %k - Error code returned by a RPC string binding function.

Explanation:

A well-known endpoint string binding could not be parsed, most likely because sufficient memory could not be allocated for the parsed strings.

User Response:

Ensure that the process has not exceeded its memory limit and that sufficient swap space exists on the machine.

**************************************************************************

ID: 0x28106076

Component: TRPC

Message: "Runtime library was provided a well-known endpoint that uses an invalid protocol sequence (%k)."

Variable: %k - Error code returned by an RPC binding function.

Explanation:

A well-known endpoint binding specified through trpc_UseWkEndpoints or trpc_BindWkEndpoints references an invalid or unsupported protocol sequence.

User Response:

Review the bindings specified by the application for correctness.

**************************************************************************

ID: 0x28106086

Component: TRPC

Message: "Runtime library was provided a well-known endpoint lacking endpoint information."

Variable: %k - Error code returned by an RPC binding function.

Explanation:

A well-known endpoint binding specified through trpc_UseWkEndpoints or trpc_BindWkEndpoints does not contain the required endpoint information.

User Response:

Review the bindings specified by the application for correctness.

****************************************************************************

ID: 0x28106c26

Component: TRPC

Message: "Runtime library could not create a dynamic endpoint for some protocol sequence (%k)."

Variable: %k - Error code returned by an RPC binding function.

Explanation:

The DCE rpc_server_use_protseq call has failed.  The DCE error returned should indicate the reason for the failure.  The most likely causes are that the maximum number of network descriptors has been reached or some other limitation has prevented socket creation.

User Response:

Find the cause indicated by the error code, such as exhaustion of the network descriptors, and raise the limit.

****************************************************************************

ID: 0x28107016

Component: TRPC

Message: "Runtime library has been given well-known endpoints that have not been bound by the application."

Explanation:

The application has called trpc_UseWkEndpoints, but at least one endpoint was specified for which no binding exists.

User Response:

Review the application for correctness and refer to the documentation for trpc_UseWkEndpoints; the use of trpc_BindWkEndpoints may have been intended.

************************************************************************

ID: 0x28107026

Component: TRPC

Message: "Runtime library needs either a well-known endpoint or the ability to use the name service because the application is recoverable."

Explanation:

A recoverable application must either permit the use of the name service or provide a well-known endpoint. This error indicates that trpc_SetEnvironment has been called to disable the use of the name service, and yet no well-known endpoints have been specified.

User Response:

Change the application to allow the use of the name service, or use trpc_UseWkEndpoints or trpc_BindWkEndpoints appropriately.

************************************************************************

ID: 0x14343c1c

Component: Trace listener

Message: "Unable to decode protection level(%s): %k"

Variable: %s - Protection level string specified in comamnd line. %k - Error code of decoding protection level.

Explanation:

An invalid protection level string is specified.

User Response:

Make sure to specify a valid protection level.

**************************************************************************

ID: 0x14343c2c

Component: Trace listener

Message: "Unable to establish security: %k"

Variable: %k - Error code set by security management.

Explanation:

Trace listener is unable to establish security.

User Response:

Make sure server principal, key file, and other security options are set correctly.

**************************************************************************

ID: 0x14343c4c

Component: Trace listener

Message: "Unable to register server with name %s (%k)."

Variable: %s - Server name. %k - Error code returned by DCE service.

Explanation:

Trace listener is unable to register the server with CDS.

User Response:

Check the CDS directory, ENCINA_CDS_ROOT environment variable, and server name for correctness.

**************************************************************************

ID: 0x14343c5c

Component: Trace listener

Message: "Unable to parse authorized principal (%s): %k"

Variable: %s - Server principal. %k - Error code.

Explanation:

Trace listener is unable parse the principal name.

User Response:

Make sure appropriate principal name is given.

**************************************************************************

### A.2.5 Server

ID: 0xac081c16

Component: Server restart

Message: "No restart data found\n"

Explanation:

The restart file given in the server command line is not found or contains invalid information.

User Response:

Make sure to supply valid restart files.

**************************************************************************

ID: 0x1c2c0416

Component: VOL

Message: "Release and version numbers in VOL restart data are invalid."

Explanation:

Release and version information on a volume does not match the Encina server version.

User Response:

Make sure the volume is not corrupted and was operated by a compatible Encina server version.

*************************************************************************

ID: 0x1c2c0426

Component: VOL

Message: "The VOL restart data is corrupted."

Explanation:

Restart data on a volume is invalid.

User Response:

Make sure the volume is not corrupted and was operated by a compatible Encina server.

*************************************************************************

ID: 0xc8340816

Component: Server command line parsing

Message: "Too many server-specific options."

Explanation:

Too many server command line arguments are specified.

User Response:

Make sure appropriate arguments are specified.

****************************************************************************

ID: 0xc8341416

Component: Server command line parsing

Message: "Unable to set key file (%s): %k"

Variables: %s - Key file name specified on command line. %k - Error code by a DCE key management function.

Explanation:

The server is unable to set the key file.

User Response:

Make sure the key file is valid and DCE security is running.

****************************************************************************

ID: 0xc8343816

Component: Server command line parsing

Message: "Unable to set authorized principal: %k"

Variables: %k - Error code by a DCE security acl function.

Explanation:

The server is unable to set authorization for specified server principal.

User Response:

Make sure the principal has been created and DCE security is running.

************************************************************************

ID: 0xc834141c

Component: Server command line parsing

Message: "Unable to decode protection level(%s): %k"

Variable: %s - Server principal. %k - Error code.

Explanation:

An invalid protection level string is specified.

User Response:

Make sure to specify a valid protection level.

************************************************************************

ID: 0xc834142c

Component: Server command line parsing

Message: "Unable to establish security: %k"

Variable: %k - Error code set by security management.

Explanation:

The server is unable to establish security.

User Response:

Make sure the server principal, key file, and other security options are set correctly.

************************************************************************

ID: 0xc834181c

Component: Server command line parsing

Message: "Illegal checkpoint interval specified: %s"

Variable: %s - Checkpoint interval specified in the command line.

Explanation:

A invalid checkpoint interval is specified.

User Response:

Examine the checkpoint interval for correctness.

************************************************************************

ID: 0xc834182c

Component: Server command line parsing

Message: "VOL service restart info missing. Use a ’%s’ switch"

Variable: %s - The missing switch "-v"

Explanation:

Volume restart device is not specified.

User Response:

Use the "-v" switch to specified the restart device.

************************************************************************

ID: 0xc834242c

Component: Server command line parsing

Message: "No server name provided.  Use a ’%s’ switch."

Variable: %s - The missing switch "-n".

Explanation:

Server name is not specified in the command line.

User Response:

Specify an appropriate server name with the "-n" switch.

****************************************************************************

ID: 0xc834281c

Component: Server command line parsing

Message: "Unable to register server with name %s (%k)."

Variable: %s - Server name. %k - Error code returned by DCE service.

Explanation:

The server failed to register the specified server name with CDS.

User Response:

Check the CDS directory, ENCINA_CDS_ROOT environment variable, and server name for correctness.

****************************************************************************

ID: 0xc834381c

Component: Server command line parsing

Message: "Unable to parse authorized principal (%s): %k"

Variable: %s - Server principal. %k - Error code.

Explanation:

The server principal name cannot be parsed by DCE security service.

User Response:

Make sure appropriate principal name is given.

****************************************************************************

### A.2.6 SFS

ID: 0x582c4486

Component: SFS EMS

Message: "ENCINA_PRINCIPAL environment variable must be set."

Explanation:

When DCE security is in use, an SFS application in the form of a COBOL program or through the ISAM interface must specify the DCE principal running the application.

User Response:

Make sure the evironment variable is set to the appropriate DCE principal.

*****************************************************************************

ID: 0x88042c26

Component: SFS EXTFH

Message: "extfh: Does NOT support interpretation of, %c, %c, %c , or %c.\n"

Variables: %c - Unsupported character ’&’. %c - Unsupported character ’>’. %c - Unsupported character ’<’. %c - Unsupported character ’|’

Explanation:

MicroFocus COBOL allows certain characters in file names that are stored in file name mapping environment variables. These characters tell the file control to redirect input and output to this file rather than just use it like a normal COBOL file. SFS does not support this feature.

User Response: Make sure those characters are not used.

*****************************************************************************

ID: 0x88043816

Component: SFS EXTFH

Message: "The %s environment variable must be set to use Encina EXTFH."

Variable: %s - The name of the missing environment variable.

Explanation:

When using SFS as a COBOL external file handler, the environment variables for server name, and volume name must be set.

User Response:

Set the environment variables.

****************************************************************************

ID: 0x881c1416

Component: SFS EXTFH

Message: "extfh:  Does NOT support NLS filename mapping.\n"

Explanation:

String "%NLS%" is specified in the file name mapping environment variable. EXTFH does not support NLS file name mapping.

User Response:  Make sure the string is not used.

****************************************************************************

ID: 0x88103c36

Component: SFS EXTFH

Message: "EXTFH: cannot do random read on sequential files"

Explanation:

Application found doing a random read on a sequential file.

User Response:

Use appropriate file type for random reads.

**************************************************************************

ID: 0x50249836

Component: SFS server

Message: "SFS Server Initialization Failure: Invalid SFS Server Version.\n"

Explanation:

The server version found in the restart area does not match the current server.

User Response:

Use appropriate server binary and make sure the volume is not corrupted.

**************************************************************************

ID: 0x502c781b

Component: SFS Server

Message: "sfs: Collating Language \"%s\" Could Not Be Found."

Variable: %s - Collating  language specified in the server start command line.

Explanation:

When SFS is restarted, the collating language specified does not match the one stored in the restart area.

User Response:

Do not specify a collating language for server restart. The server can find it from the restart data.

**************************************************************************

ID: 0x582c0846

Component: SFS TISAM

Message: "invalid %s environment value: %s.\n"

Variable: %s - Name of the environment variable ENCINA_TISAM_MODE. %s - Value of the environment variable.

Explanation:

Environment variable ENCINA_TISAM_MODE is not set to one of the two valid values: TISAM_STANDARD_MODE and TISAM_COMPATIBILITY_MODE.

User Response:

Check the setting of this evironment variable.

**************************************************************************

### A.2.7 RQS

ID: 0x8cc44416

Component: RQS client

Message: "rqs_Free: queue 0x%x belongs to too many queue sets (%d).\n"

Variable: %x - Pointer to the queue’s status structure. %d - Max number of queue sets to which a queue may belong.

Explanation:

The number of queue sets to which a queue belongs exceeds RQS_MAX_QSETS_PER_QUEUE.

User Response:

Observe the limit when inserting a queue into queue sets.

**************************************************************************

ID: 0x8c556826

Component: RQS client

Message: "PostLogFileInit: Collating Language \"%s\" Could Not Be Found."

Variable: %s - Collating language specified in the server start command line.

Explanation:

When RQS is restarted, the collating language specified does not match the one stored in the restart area.

User Response:

Do not specify a collating language for server restart. The server can find it from the restart data.

# Appendix B. Special Notices

This publication is intended to help application developers and application designers design, develop, and deploy their applications based on Encina technology. The information in this publication is not intended as the specification of any programming interfaces that are provided by Encina products. See the PUBLICATIONS section of the IBM Programming Announcement for TXSeries and Encina for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

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, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

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

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have

| | |
|---|---|
| IBM ® | ACF/VTAM |
| AIX | APPN |
| AS/400 | C Set ++ |
| CICS Transaction Server | CICS/MVS |
| CICS/ESA | CICS/VSE |
| CICS/400 | CICSPlex |
| Database 2 | DB2 |
| MQSeries | OS/2 |
| Presentation Manager | RACF |
| RISC System/6000 | RS/6000 |
| S/390 | VTAM |

| | |
|---|---|
| C++ | American Telephone and Telegraph Company, Incorporated |
| C-bus | Corollary, Inc. |
| CyberCash | CyberCash, Inc. |
| DCE, OSF | Open Software Foundation |
| DEC, VT100 | Digital Equipment Corporation |
| Encina | Transarc Corporation |
| Gopher | University of Minnesota |

# Appendix C. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## C.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 425.

- *Administering IBM DCE and DFS Version 2.1*, SG24-4714
- *DCE Cell Design Considerations*, SG24-4746
- *Accessing CICS Business Applications from the WWW*, SG24-4547
- *IBM Internet Connection Secure Server*, SG24-4805
- *Building a Firewall with the NetSP Secured Network Gateway*, SG24-2577
- *CICS Clients Unmasked*, SG24-2534
- *TCP/IP Tutorial and Technical Overview*, GG24-3376

## C.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

| CD-ROM Title | Subscription Number | Collection Kit Number |
|---|---|---|
| System/390 Redbooks Collection | SBOF-7201 | SK2T-2177 |
| Networking and Systems Management Redbooks Collection | SBOF-7370 | SK2T-6022 |
| Transaction Processing and Data Management Redbook | SBOF-7240 | SK2T-8038 |
| Lotus Redbooks Collection | SBOF-6899 | SK2T-8039 |
| Tivoli Redbooks Collection | SBOF-6898 | SK2T-8044 |
| AS/400 Redbooks Collection | SBOF-7270 | SK2T-2849 |
| RS/6000 Redbooks Collection (HTML, BkMgr) | SBOF-7230 | SK2T-8040 |
| RS/6000 Redbooks Collection (PostScript) | SBOF-7205 | SK2T-8041 |
| RS/6000 Redbooks Collection (PDF Format) | SBOF-8700 | SK2T-8043 |
| Application Development Redbooks Collection | SBOF-7290 | SK2T-8037 |

## C.3 Other Publications

The publications listed in Table 12 are also relevant as further information sources.

*Table 12. The Manuals for IBM Transaction Server for Windows NT*

| Order Number | Book Title | Description |
|---|---|---|
| GC33-1878 | Concepts and Facilities | Provides a technical overview of the concepts, services, and components of IBM Transaction Server for Windows NT |
| GC33-1879 | Quick Beginnings | Provides step-by-step instructions for getting a CICS server and an Encina Monitor running and for installing the CICS clients (including the CICS Client for AIX) that run with Transaction Server |
| GC33-1880 | Installation Guide | Provides system administrators with information about system requirements and installing and configuring IBM Transaction Server for Windows NT |
| SC33-1881 | Administration Guide (CICS) | Provides information for system administrators about all aspects of setting up and managing an operational CICS for Windows NT system |
| SC33-1885 | Administration Reference (CICS) | Provides reference information for system administrators on the commands and definitions required to administer a CICS system. Use in conjunction with the Administration Guide (CICS) |
| SC33-1795 | Basic Administration (Encina) | Provides information for system administrators about installing, configuring, and managing the Encina components of IBM Transaction Server for Windows NT |
| SC33-1948 | Server Administration (Encina) | Provides information for system administrators about adminstering the RQS, SFS, and PPC Gateway for Encina |
| SC33-1949 | Advanced Administration (Encina) | Provides information for system administrators about Encina maintenance, recovery, security, and performance |

| Order Number | Book Title | Description |
|---|---|---|
| SC33-1798 | Administration Reference (Encina) | Provides reference information for system administrators on the commands and definitions required to administer an Encina system.  Use in conjunction with Basic Administration (Encina) |
| SC33-1882 | Intercommunication Guide | Describes how CICS for Windows NT can communicate with other members of the CICS family and any other system that supports the LU 6.2 protocol |
| GC33-1883 | Problem Determination Guide | Provides guidance information to help in resolving CICS application and system problems and in using your support organization |
| GC33-1886 | Messages and Codes | Lists the messages and codes that CICS for Windows NT issues |
| SC33-1888 | Application Programming Guide (CICS) | Provides guidance information for application programmers about preparing applications in COBOL, C, or C++ using the CICS Application Programming Interface (API) and migrating CICS applications to and from CICS for Windows NT |
| SC33-1887 | Application Programming Reference (CICS) | Provides reference information for application programmers to prepare COBOL, C, or C++ applications using the CICS API |
| SC33-1914 | Encina Monitor Programming Guide | Describes how to develop Encina Monitor applications. |
| SC33-1915 | Encina PPC Services Programming Guide | Describes the programming environment and related utilities for the Encina PPC Executive |
| SC33-1916 | Encina RQS Programming Guide | Describes the RQS programming environment and how to develop Encina RQS applications |
| SC33-1917 | Encina SFS Programming Guide | Describes the SFS programming environment and how to develop Encina SFS applications |

| Order Number | Book Title | Description |
|---|---|---|
| SC33-1918 | Encina Toolkit Programming Guide | Describes the organization of the Encina Toolkit, the interaction between the various modules of the Toolkit, and how to use specific Toolkit interfaces to develop distributed transactional applications |
| SC33-1919 | Encina Transactional Programming Guide | Describes how to develop distributed, transactional applications using the Encina Tran-C programming interface. It also describes the Encina TX interface and TIDL |
| SC33-1909 | Encina C Programming Reference Volume 1 | Provides reference information for the data types and functions of Encina Monitor, Transactional-C, Encina Monitor Administration (EMA), X/Open TX, and PPC Services interfaces |
| SC33-1910 | Encina C Programming Reference Volume 2 | Provides reference information for the data types and functions of Encina RQS and SFS interfaces |
| SC33-1911 | Encina C Programming Reference Volume 3 | Provides reference information for the data types and functions of Encina Distributed Transaction Service (TRAN), Thread-to-Tid Mapping Service (threadTid), and Transactional RPC Service (TRPC) components of the Encina Executive, and for the Encina Abort Facility, Trace Facility, and Transarc Encina DCE Utilities (TRDCE) |
| SC33-1912 | Encina C Programming Reference Volume 4 | Provides reference information for the data types and functions of the Encina Lock Service (LOCK), Log Service (LOG), Recovery Service (REC), Transaction Manager-XA Service (TM-XA), and Volume Service (VOL) components of the Encina Toolkit Server Core, and for the Encina Restart Service |
| SC33-1913 | Encina COBOL Programming Guide and Reference | Describes how to develop Encina Monitor applications using COBOL |

| Order Number | Book Title | Description |
|---|---|---|
| SC33-1760 | Writing Encina Applications | Provides guidance for application programmers who want to learn how to program in the Encina environment |
| SC33-1611 | Encina SFS Supplement to ISAM | Provides guidance about using the Encina Transactional Indexed Sequential Access Method (T-ISAM) interface |
| SC33-1759 | Encina++ Programming Guide and Reference | Explains how to develop object-oriented distributed applications using the Encina++ extensions to the C++ language. It also provides reference information for the Encina C++, Transactional C++, and OMG Object Transaction Service (OTS) C++ interfaces |
| | IBM CICS Client, Version 2.0.1 | |
| SC33-1792 | CICS Clients: Administration | Provides system administrators with information about installing, customizing, and controlling the IBM CICS Clients for DOS, OS/2, Windows, Windows NT, Windows 95, and Macintosh |
| SC33-1793 | CICS Clients: Messages | Lists user, error log, and trace log messages for the IBM CICS Clients for DOS, OS/2, Windows, Windows NT, Windows 95, and Macintosh |
| SC33-1821 | CICS Clients: Gateways | Describes how to set up and use the IBM CICS gateway for Lotus Notes and the IBM CICS Internet gateway, provided with the CICS Client for OS/2 or the CICS Client for Windows NT |
| | CICS Family | |
| SC33-1898 | Using IBM Communications Server for AIX with CICS | Describes how the IBM Communication Server for AIX can provide either local SNA CICS support to the CICS for AIX region or be used within an Encina PPC Gateway Server to provide gateway SNA support |

| Order Number | Book Title | Description |
|---|---|---|
| SC33-1899 | Using Microsoft SNA Server Version 2 with CICS | Describes how the Microsoft SNA Version 2 product can provide local SNA support to the CICS for Windows NT region |
| SC33-1715 | Using Microsoft SNA Server Version 3 with CICS | Describes how the Microsoft SNA Version 3 product can provide local SNA support to the CICS for Windows NT region |
| SC33-1900 | Using IBM Communication Server for Windows NT with CICS | Describes how the IBM Communication Server for Windows NT can provide local SNA support for a CICS for Windows NT region |
| SC33-0824 | CICS Family: Interproduct Communication | Introduces the CICS intercommunication functions for the CICS family of products |
| SC33-1435 | CICS Family: Client/Server Programming | Provides information on using the CICS External Call Interface (ECI) and CICS External Programming Interface (EPI) to develop client applications to use CICS systems as servers |
| SC33-1007 | CICS Family: API Structure | Provides a cross-reference to the level of support provided \| by each member of the CICS family for the CICS application programming interface (API) and the system programming INQUIRE and SET commands |
| SC33-1923 | CICS Family: OO Programming in C++ for CICS Clients | Describes object-oriented programming in C++ for the external call interface (ECI) and external presentation interface (EPI) using the classes and methods provided with CICS Clients Version 2.0.1 |
| SC33-1924 | CICS Family: OO Programming in BASIC for CICS Clients | Describes object-oriented programming in Visual BASIC for the external call interface (ECI) and external presentation interface (EPI) using the classes and methods provided with CICS Clients Version 2.0.1 |

All TXSeries documentation and manuals can be accessed online at http://www.transarc.com.

# How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

## How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** – to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BokkManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

**Redpieces**

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

## How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:

| | **IBMMAIL** | **Internet** |
|---|---|---|
| In United States | usib6fpl at ibmmail | usib6fpl@ibmmail.com |
| In Canada | caibmbkz at ibmmail | lmannix@vnet.ibm.com |
| Outside North America | dkibmbsh at ibmmail | bookshop@dk.ibm.com |

- **Telephone Orders**

| | |
|---|---|
| United States (toll free) | 1-800-879-2755 |
| Canada (toll free) | 1-800-IBM-4YOU |
| | |
| Outside North America | (long distance charges apply) |
| (+45) 4810-1320 - Danish | (+45) 4810-1020 - German |
| (+45) 4810-1420 - Dutch | (+45) 4810-1620 - Italian |
| (+45) 4810-1540 - English | (+45) 4810-1270 - Norwegian |
| (+45) 4810-1670 - Finnish | (+45) 4810-1120 - Spanish |
| (+45) 4810-1220 - French | (+45) 4810-1170 - Swedish |

- **Mail Orders** – send orders to:

| | | |
|---|---|---|
| IBM Publications | IBM Publications | IBM Direct Services |
| Publications Customer Support | 144-4th Avenue, S.W. | Sortemosevej 21 |
| P.O. Box 29570 | Calgary, Alberta T2P 3N5 | DK-3450 Allerød |
| Raleigh, NC 27626-0570 | Canada | Denmark |
| USA | | |

- **Fax** – send orders to:

| | |
|---|---|
| United States (toll free) | 1-800-445-9269 |
| Canada | 1-800-267-4455 |
| Outside North America | (+45) 48 14 2207 (long distance charge) |

- **1-800-IBM-4FAX (United States)** or **(+1) 408 256 5422 (Outside USA)** – ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

| | |
|---|---|
| Redbooks Web Site | http://www.redbooks.ibm.com |
| IBM Direct Publications Catalog | http://www.elink.ibmlink.ibm.com/pbl/pbl |

**Redpieces**

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

## IBM Redbook Order Form

**Please send me the following:**

| Title | Order Number | Quantity |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

| | | |
|---|---|---|
| First name | Last name | |
| Company | | |
| Address | | |
| City | Postal code | Country |
| Telephone number | Telefax number | VAT number |

☐ Invoice to customer number

☐ Credit card number

| | | |
|---|---|---|
| Credit card expiration date | Card issued to | Signature |

**We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.**

# Glossary

An excellent glossary of Internet and Internet related terms is available at:

http://www.matisse.net/files/glossary.html

Other terms not covered in the above-mentioned Web document or clarified in this document are listed below.

**anchor.** An HTML element that defines a link between Internet resources.

**abend.** Abnormal end of task.

**API.** Application programming interface. A set of calling conventions defining how a service is invoked through a software package.

**APPC.** Advanced program-to-program communication. An implementation of the SNA LU 6.2 protocol that allows interconnected systems to communicate and share the processing of programs.

**asynchronous.** Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions. See *synchronous*.

**browser.** An application that displays World Wide Web documents.

**CERN.** The Conseil Europeen pour la Recherche Nucleaire (European Particle Physics Laboratory), which developed hypertext technologies.

**distributed program link (DPL).** Enables an application program executing in one CICS system to link (pass control) to a program in a different CICS system. The linked-to program executes and returns a result to the linking program. This process is equivalent to remote procedure calls (RPCs). You can write applications that issue RPCs that can be received by members of the CICS family.

**distributed transaction processing (DTP).** Enables a transaction running in one CICS system to communicate synchronously with transactions running in other systems. The transactions are designed and coded specifically to communicate with each other. This method is typically used by banks, for example, in "just-in-time" stock replacement.

**Customer Information Control System (CICS).** A distributed online transaction processing system designed to support a network of many terminals. The CICS family of products is available for a variety of platforms ranging from a single workstation to the largest mainframe.

**client.** As in client/server computing, the application that makes requests to the server and, often, handles the interaction necessary with the user.

**client/server computing.** A form of distributed processing, in which the task required to be processed is accomplished by a client portion that requests services and a server portion that fulfills those requests. The client and server remain transparent to each other in terms of location and platform. See *client* and *server*.

**commit.** An action that an application takes to make permanent the changes it has made to CICS resources.

**Common Gateway Interface (CGI).** The defined standard for the communications between HTTP servers and external executable programs.

**conversational.** A communication model where two distributed applications exchange information by way of a conversation; typically one application starts (or allocates) the conversation, sends some data, and allows the other application to send some data. Both applications continue in turn until one decides to finish (or deallocate). The conversational model is a synchronous form of communication.

**database.** (1) A collection of interrelated data stored together with controlled redundancy according to a scheme to serve one or more applications. (2) All data files stored in the system. (3) A set of data stored together and managed by a database management system.

**Distributed Computing Environment (DCE).** Adopted by the computer industry as a de facto standard for distributed computing. DCE allows computers from a variety of vendors to communicate transparently and share resources such as computing power, files, printers, and other objects in the network.

**delimiter.** A character or sequence of characters used as a separator in text or data files.

**distributed processing.** An application or systems model in which function and data can be distributed across multiple computing resources connected on a LAN or WAN. See *client/server computing*.

**external call interface (ECI).** An application programming interface (API) that enables a non-CICS client application to call a CICS program as a subroutine. The client application communicates with the server CICS program using a data area called a COMMAREA.

**external presentation interface (EPI).** An application programming interface (API) that allows a non-CICS application program to appear to the CICS system as one or more standard 3270 terminals. The non-CICS application can start CICS transactions and send and receive standard 3270 data streams to those transactions.

**environment.** The collective hardware and software configuration of a system.

**File Transfer Protocol (FTP).** A protocol that defines how to transfer files from one computer to another.

**forms.** Parts of HTML documents that allow users to enter data.

**function shipping.** Enables an application program running in one CICS system to access resources owned by another CICS system. In the resource-owning system, a transaction is initiated to perform the necessary operation; for example, to access CICS files or temporary storage, and to reply to the requester. The user is unaware of these "behind-the-scenes" activities and need not know where the resource actually exists.

**gateway.** Software that transfers data between normally incompatible applications or between networks.

**gopher.** Menu-based software for exploring Internet resources.

**Graphic Interchange Format (GIF).** 256-color graphic format.

**graphical user interface (GUI).** A style of user interface that replaces the character-based screen with an all-points-addressable, high-resolution graphics screen. Windows display multiple applications at the same time and allow user input by means of a keyboard or a pointing device such as a mouse, pen, or trackball.

**home page.** The default page shown at the first connection to an HTTP server.

**host.** (1) In a computer network, a computer providing services such as computation, database access, and network control functions. (2) In a multiple computer installation, the primary or controlling computer.

**hypertext.** Text that activates connection to other documents when selected.

**Hypertext Markup Language (HTML).** Standard language used to create hypertext documents.

**Hypertext Transmission Protocol (HTTP).** Standard WWW client/server communications protocol.

**Internet Keyed Payment Protocol (iKP).** Proposed protocol for conducting secure commercial financial transactions on the Internet.

**intercommunication.** Communication between separate systems by means of Systems Network Architecture (SNA), Transmission Control Protocol/Internet Protocol (TCP/IP), and Network Basic Input/Output System (NetBIOS) networking facilities.

**Internet.** A collection of networks.

**LU type 6.2 (LU 6.2).** A type of logical unit used for CICS intersystem communication (ISC). LU 6.2 architecture supports CICS host-to-system-level products and CICS host-to-device-level products. APPC is the protocol boundary of the LU 6.2 architecture.

**logical unit of work (LUW).** An update that durably transforms a resource from one consistent state to another consistent state. A sequence of processing actions (for example, database changes) that must be completed before any of the individual actions can be regarded as committed. When changes are committed (by successful completion of the LUW and recording of the synch point on the system log), they do not need to be backed out after a subsequent error within the task or region. The end of an LUW is marked in a transaction by a synch point that is issued by either the user program or the CICS server, at the end of task. If there are no user synch points, the entire task is an LUW.

**markup tag.** Special character sequences put in text used to pass information to a tool, such as a document formatter.

**NCSA Mosaic.** A Web browser available on multiple platforms.

**Multipurpose Internet Mail Extension (MIME).** The Internet standard for mail that supports text, images, audio, and video.

**online transaction processing (OLTP).** A style of computing that supports interactive applications in which requests submitted by terminal users are processed as soon as they are received. Results are returned to the requester in a relatively short period of time. An online transaction processing system supervises the sharing of resources to allow efficient processing of multiple transactions at the same time.

**PostScript.** The standard for presenting text and graphics in a device-independent format.

**protocol.** (1) A formal set of conventions governing the format and control of data. (2) A set of procedures or rules for establishing and controlling transmissions from a source device or process to a target device or process.

**proxy.** A gateway that allows Web browsers to pass on a network request (a URL) to an outside agent.

**pseudoconversational.** A type of CICS application design that appears to the user as a continuous conversation but consists internally of multiple tasks.

**recovery.** The use of archived copies to reconstruct files, databases, or complete disk images after they are lost or destroyed.

**recoverable resources.** Items whose integrity CICS maintains in the event of a system error. These include individual files and queues.

**script.** An executable program invoked by HTTP servers.

**server.** Any computing resource dedicated to responding to client requests. Servers can be linked to clients through LANs or WANs to perform services, such as printing, database access, fax, and image processing, on behalf of multiple clients at the same time.

**Socket Secure (SOCKS).** The gateway that allows compliant client code (client code made socket secure) to establish a session with a remote host.

**Standard Generalized Markup Language (SGML).** The standard that defines several markup languages, HTML included.

**synchronous.** (1) Pertaining to two or more processes that depend on the occurrence of a specific event such as a common timing signal. (2) Occurring with a regular or predictable time relationship.

**synchpoint.** A logical point in execution of an application program where the changes made to the databases by the program are consistent and complete and can be committed to the database. The output, which has been held up to that point, is sent to its destination, the input is removed from the message queues, and the database updates are made available to other applications. When a program terminates abnormally, CICS recovery and restart facilities do not back out updates prior to the last completed synchpoint.

**transaction.** A unit of processing (consisting of one or more application programs) initiated by a single request. A transaction can require the initiation of one or more tasks for its execution.

**transaction processing.** A style of computing that supports interactive applications in which requests submitted by users are processed as soon as they are received. Results are returned to the requester in a relatively short period of time. A transaction processing system supervises the sharing of resources for processing multiple transactions at the same time.

**transaction routing.** Enables a terminal connected to one CICS system to run a transaction in another CICS system. It is common for CICS/ESA, CICS/VSE, and CICS/MVS users to have a terminal-owning region (TOR) that "owns" end-user network resources.

# List of Abbreviations

| | |
|---|---|
| ***ACF*** | access control file |
| ***ACL*** | access control list |
| ***AIX*** | Advanced Interactive eXecutive |
| ***APA*** | all points addressable |
| ***API*** | application programming interface |
| ***APPC*** | Advanced Program-to-Program Communication |
| ***AS*** | Application Support |
| ***ASCII*** | American National Standard Code for Information Interchange |
| ***BMS*** | basic mapping support |
| ***CERN*** | Conseil Europeen pour la Recherche Nucleaire (European Laboratory for Particle Physics) |
| ***CGI*** | Common Gateway Interface |
| ***CICS*** | Customer Information Control System |
| ***CM/2*** | Communications Manager/2 |
| ***COMMAREA*** | communication area |
| ***CRP*** | current record pointer |
| ***CSD*** | CICS system definition |
| ***DCE*** | Distributed Computing Environment |
| ***DEC*** | Digital Equipment Corporation |
| ***DNS*** | Domain Name Server |
| ***DOS*** | Disk Operating System |
| ***DPL*** | distributed program link |
| ***DTP*** | distributed transaction processing |
| ***ECI*** | external call interface |
| ***EPI*** | external presentation interface |
| ***ESA*** | Enterprise Systems Architecture |
| ***EXCI*** | external CICS interface |
| ***FAT*** | file allocation table |
| ***FTP*** | File Transfer Protocol |
| ***GIF*** | graphic interchange format |
| ***HPFS*** | High Performance File System |
| ***HTML*** | Hypertext Markup Language |
| ***HTTP*** | Hypertext Transfer Protocol |
| ***IBM*** | International Business Machines Corporation |
| ***IDL*** | Interface Definition Language (see also TIDL) |
| ***IETF*** | Internet Engineering Task Force |
| ***iKP*** | Internet Keyed Payment Protocol |
| ***IMS*** | Information Management System |
| ***IP*** | Internet Protocol |
| ***ISC*** | intersystem communication |
| ***ITSO*** | International Technical Support Organization |
| ***LAN*** | local area network |
| ***LUW*** | logical unit of work |

| | |
|---|---|
| ***MIME*** | Multipurpose Internet Mail Extension |
| ***NCSA*** | National Center of Supercomputing Applications |
| ***OCCS*** | Object Concurrency Control Service |
| ***OLTP*** | online transaction processing |
| ***OMG*** | Object Management Group |
| ***ONC RPC*** | Open Network Computing Remote Procedure Call |
| ***ORB*** | Object Request Broker |
| ***OS/2*** | Operating System/2 |
| ***OSF*** | Open Software Foundation Inc. |
| ***OTMA*** | Open Transaction Manager Access |
| ***OTS*** | Object Transaction Service |
| ***PGP*** | pretty good privacy |
| ***PIN*** | personal identification number |
| ***PM*** | Presentation Manager |
| ***POP*** | Post Office Protocol |
| ***PPC*** | peer to peer communication |
| ***RACF*** | Resource Access Control Facility |
| ***RPC*** | remote procedure call |
| ***RQS*** | Recoverable Queuing System |
| ***SET*** | Secure Electronic Transaction |
| ***SFS*** | Structured File Server |
| ***S-HTTP*** | Secure Hypertext Transfer Protocol |
| ***SGML*** | Standard Generalized Markup Language |
| ***SMTP*** | Simple Mail Transfer Protocol |
| ***SNA*** | Systems Network Architecture |
| ***SNT*** | signon table |
| ***SOCKS*** | socket secure |
| ***SSL*** | Secure Sockets Layer |
| ***TCP/IP*** | Transmission Control Protocol/Internet Protocol |
| ***TIDL*** | Transactional Interface Definition Language |
| ***TOR*** | terminal owning region |
| ***TRUE*** | task-related user exit |
| ***URI*** | uniform resource identifier |
| ***URL*** | uniform resource locator or universal resource locator |
| ***WWW*** | World Wide Web |
| ***WYSIWYG*** | What you see is what you get |
| ***W3*** | World Wide Web (mostly used for Intranet declaration) |

# Index

# ITSO Redbook Evaluation

Developing Distributed Transaction Applications with Encina
SG24-5241-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
**_ Customer  _ Business Partner  _ Solution Developer  _ IBM employee**
**_ None of the above**

**Please rate your overall satisfaction** with this book using the scale:
**(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)**

Overall Satisfaction __________

**Please answer the following questions:**

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

What other redbooks would you like to see published?

_______________________________________________

_______________________________________________

_______________________________________________

**Comments/Suggestions:  (THANK YOU FOR YOUR FEEDBACK!)**

_______________________________________________

_______________________________________________

_______________________________________________

_______________________________________________

**Developing Distributed Transaction Applications with Encina** **SG24-5241-00**

**SG24-5241-00**
**Printed in the U.S.A.**