TXSeries™ for Multiplatforms

Encina Object-Oriented Programming Guide

Version 5.1

SC09-4478-05

TXSeries™ for Multiplatforms

Encina Object-Oriented Programming Guide

Version 5.1

SC09-4478-05

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 75.

Sixth Edition (March 2004)

This edition replaces SC09-4478-04.

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

Contents

Figures

Tables

About this book

This document explains how to use Encina++ to develop object-oriented distributed applications in the Distributed Computing Environment (DCE). Encina++ simplifies application development by providing high-level interfaces to the Encina® Toolkit development tools, the Encina Monitor, the Encina Recoverable Queueing Service (RQS), and the Encina Structured File Server (SFS). Each interface provides a specific type of service or performs a specific task required by distributed applications. Encina++ includes a set of extensions to the C++ language that provide direct access to low-level constructs, such as threading.

Who should read this book

This guide is intended for developers who are new to developing object-oriented distributed or transactional applications. Users of this document should be familiar with program development in the C++ programming language and the Encina Monitor.

Document organization

This document has the following organization:

- Chapter 1, "What is Encina++?," on page 1 describes the application programming interfaces and the support for client and server applications in the DCE environment.
- Chapter 2, "The Encina++ programming model," on page 5 introduces the concepts behind object-oriented transaction processing and distributed computing.
- Chapter 3, "Developing distributed applications," on page 9 provides general information on writing distributed client/server applications using Encina++.
- Chapter 4, "Developing Encina++/DCE applications," on page 17 provides an overview of the issues specific to developing applications in the DCE environment and describes how to write distributed client/server applications in the DCE environment.
- Chapter 5, "Transaction processing overview," on page 39 provides an overview of the transaction demarcation interfaces available in Encina++.
- Chapter 6, "Transaction processing with Transactional-C++," on page 41 describes how to use the Transactional-C++ interface to begin, end, and manage transactions in Encina++ applications.
- Chapter 7, "Transaction processing with OMG OTS for Encina++," on page 55 describes how to use the Encina++ Object Transaction Service (OTS) interface to begin, end, and manage transactions in Encina++ applications.
- Chapter 8, "Using threads," on page 63 provides information on using threads in Encina++ applications.
- Chapter 9, "Diagnostics," on page 69 provides information on the diagnostic facilities available for Encina++.
- "Compilation issues," on page 71 provides information on Encina++ header and library files, Java package and class library files, and some miscellaneous compilation issues.

Related information

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

- *Writing Encina Applications*
- *Encina Administration Guide Volume 1: Basic Administration*
- *Encina Monitor Programming Guide*
- *Encina RQS++ and SFS++ Programming Guide*
- *Encina Transactional Programming Guide*
- *Encina Toolkit Programming Guide*
- *OSF DCE Application Development Reference* and *OSF DCE Application Development Guide* (for information about developing applications that use DCE)

Conventions used in this book

TXSeries documentation uses the following typographical and keying conventions.

Table 1. Conventions used in this book

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

Table 1. Conventions used in this book (continued)

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

How to send your comments

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

Chapter 1. What is Encina++?

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

DCE support

Encina++ supports the development of object-oriented applications that are based on the Distributed Computing Environment (DCE).

Encina++ contains several different application programming interfaces.

Note: The Encina++ header files contain definitions of classes and member functions that are not documented. Undocumented classes and functions are not supported and, therefore should not be used. There is no guarantee that undocumented classes or functions will be available or will exhibit the same behavior in future releases.

Client and server support interfaces

The following Encina++ programming interfaces provide client and server support and transaction demarcation capabilities.

- The *Encina*++ 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 interface.
- The *Object Management Group Object Transaction Service* (OMG OTS) interface also defines C++ classes and member functions for distributed transactional processing. This interface implements the OMG *Object Transaction Service* specification as documented in *OMG document 94.8.4*.

To write Encina++/DCE applications, you must use Encina's Transactional Interface Definition Language (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.

Encina++/DCE clients bind to exported server objects by using constructors defined in the client stubs generated by the TIDL compiler.

Object-oriented access interfaces

The following Encina++ programming interfaces provide object-oriented access to two types of Encina servers offering specialized services:

- The Recoverable Queueing Service C++ interface (RQS++) defines C++ classes and functions for enqueueing and dequeueing 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 the classes used by RQS++ and SFS++ applications. For more information on these interfaces, see the *Encina RQS++ and SFS++ Programming Guide*.

Client and server support

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++ that access Encina++ servers.

C++ clients and servers

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++. See the *Encina Monitor Programming Guide* and *Concepts and Planning* for more information on the Encina 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.

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.

Encina++ terminology

The object-oriented programming interfaces for Encina can be grouped together in a variety of ways, depending on which components and environments are used. Table 2 describes the names used to refer to them in this manual. "DCE support" on page 1 provides more information about the individual interfaces.

Table 2. Description of Encina++ product names

Term	**Description**
Encina++	Used as a generic name for all the object-oriented interfaces in Encina, including client/server support, transaction processing support for C++, RQS++, and SFS++.
Encina++/DCE	Refers to the interfaces that can be used in a DCE-only environment, including client/server support, Tran-C++, and a subset of OMG OTS.

Table 2. Description of Encina++ product names (continued)

Term	Description
OTS for Encina++/DCE	Refers specifically to the OMG OTS interface for the DCE-only environment. This does *not* include client/server support or Tran-C++.
SFS++	Refers to the C++ interfaces to SFS.
RQS++	Refers to the C++ interfaces to RQS.

Chapter 2. The Encina++ programming model

Encina++ is an extension to Encina that is used to build object-oriented distributed computing systems. Encina++ supports a *client/object* programming model in which Encina++ clients can access objects exported by Encina++ application servers. Using objects to build distributed computing systems provides benefits such as faster application development, reduced complexity, and improved reuse of application code.

This chapter provides an overview of the Encina++ programming model. The topics introduced in this chapter are covered in greater detail in later chapters.

Object-oriented distributed computing

In distributed object computing, objects can be located across a variety of platforms and in different processes and can communicate transparently with each other (by issuing method requests) as if they were located on a single machine. As illustrated in Figure 1, client objects invoke methods on implementation objects exported by servers; implementation objects are typically located on different machines from the clients. Clients are unaware that the methods are being performed remotely—much like conventional clients that make remote procedure calls (RPCs).

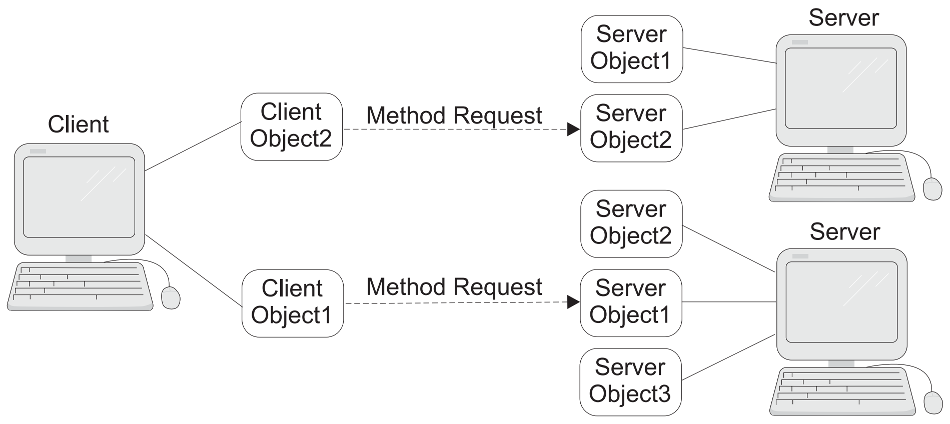

Figure 1. Distributed object computing

| The Distributed Computing Environment (DCE) includes features that support the
| use of distributed objects. For example, DCE provides an interface definition
| language (IDL) that provides a way to group related operations together in a
| logical fashion, much the way a class definition does. An interface definition can
| be mapped to a class definition in which class member functions represent the
| operations defined in the interface (as shown in Figure 2 on page 6).

```
interface account
{
    void credit([in] float amount);
    void debit([in] float amount);
}
```

```
class account
{
public:
   void credit(float amount);
   void debit(float amount);
}
```

Figure 2. Simple mapping of a DCE interface to a C++ class

The Encina++ programming model provides a distributed, object-oriented programming environment for developing Encina applications. Using the Transactional IDL (TIDL) compiler for Encina++/DCE, you can generate class definitions from interface definition files. The Encina++ application programming interfaces enable you to create and manage distributed objects based on those generated classes.

Client/object programming

The Encina++ classes support a client/object programming model in which clients access objects instead of servers. Servers make one or more interfaces (classes) available by exporting one or more instances of each class (objects). The client application can access objects exported by servers without the application developer 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 any appropriate object when the specific objects are not known or when all objects of a specific class provide the same capabilities. Typically, the application developer specifies a name for an object.

In Encina++, an interface definition language 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.

Communications

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 implementation class (also known as 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 an *implementation 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 implementation object (as shown in Figure 3 on page 7). After the client proxy object is bound to the implementation object, each member function call made on the client proxy object invokes an RPC to the implementation object, which executes the procedure and returns results to the client proxy object. The client communicates with the implementation object via the client proxy object.

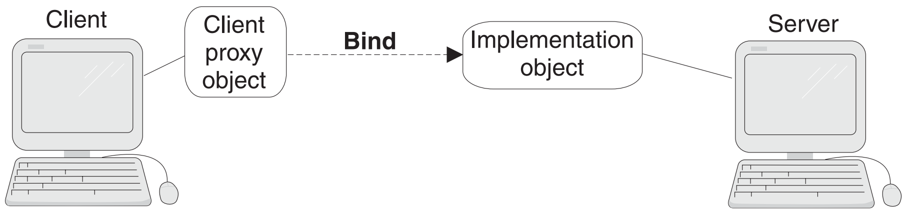

Figure 3. Binding to an implementation object

Application initialization and management

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. Operations available on an initialized server instance can be used to register XA resources, make implementation 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.

Other classes are provided for managing various aspects of an Encina++ application. For example, threads of control can be created in both transactional and nontransactional contexts (see “Transactional and nontransactional threads”).

Transaction processing

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. (Refer to the *Encina Transactional Programming Guide* for more information on transaction processing.) Encina++ supplies two different C++ interfaces for object-oriented transaction processing: Tran-C++ and the Object Management Group (OMG) OTS. These interfaces can be used either separately or together in an Encina++ application; issues regarding the compatibility of Tran-C++ and OMG OTS are covered in “Interactions between Tran-C++ and OTS interfaces” on page 61.

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.

Transactional and nontransactional threads

Threads can be used to improve the performance of an application. An application can perform a task (or set of tasks) in less time by executing operations concurrently rather than serially.

Encina++ provides classes that encapsulate Portable Operating System Interface (POSIX) threads. These classes allow you to create and manage two different types of threads: transactional threads and nontransactional threads. A *transactional thread* executes on behalf of the current transaction, if one exists. Using transactional threads, you can create multiple threads that execute concurrently within a single transaction or multiple transactions that execute concurrently. A *nontransactional thread* does not execute on behalf of a transaction even if created within the scope of a transaction.

Chapter 3. Developing distributed applications

This chapter describes the basic steps used to develop a distributed application with Encina++. The chapter also describes how to use features such as exceptions and threads that are common to all Encina++ applications.

Overview of application development

An Encina++ application is written in an object-oriented language (such as C++) and consists of calls on both local and remote objects. The interface to a remote object is defined by using an interface definition language (IDL). 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 beyond the scope of this document. Steps 2, 3, and 5 depend on the DCE environment.

Step 4 is to write the application code for your client and server. This step involves some tasks that are environment-specific and some that are not. Tasks such as initialization and termination of Encina++ clients and servers, are covered in this chapter.

The environment-specific tasks, such as generating client and server stubs and binding to remote objects, are covered in later chapters. Chapter 4, “Developing Encina++/DCE applications,” on page 17 provides information on developing Encina++ applications in a DCE environment.

Writing client applications

This section describes how to initialize and terminate an Encina++ client application. In addition to performing tasks that are specific to your application, Encina++ client applications must perform the following basic steps:

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

Each step is described in the following sections.

Initializing a client application

The **Encina::Client::Initialize** function is used to initialize a client application. This function initializes all of the necessary underlying Encina components and services. Figure 4 on page 10 shows a simple example of initializing a client application.

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

```
int main(int argc, char *argv[])
{
   // process command-line arguments
   ...
   // Initialize the client
   Encina::Client::Initialize();
   try {
      // Bind to a remote object
   }
   catch (...) {
      cerr << "An exception was raised." << endl;
      Encina::Client::Exit(1);
   }
   // Perform work
   ...
   Encina::Client::Exit(0);
}
```

Figure 4. Example of initializing an Encina client

The client program examples in this and other chapters use the **Encina::Client** class to initialize a client application. For Encina++/DCE clients that bind to Monitor application servers (MAS) only, you can also use the **EncinaMonitorClient** class to initialize a client. This class, which is publicly derived from the **Encina::Client** class, provides an overloaded **EncinaMonitorClient::Initialize** function, which initializes the client and associates it with a specific Encina Monitor cell. This function allows you to specify the following:

- The Monitor cell in which the client operates. If the function does not specify a particular Monitor cell, the value of the ENCINA_TPM_CELL environment variable is used, just as with the **Encina::Client::Initialize** function.
- Whether the client calls the **Encina::Client::Exit** function when it receives an interrupt. By default, it does not.

Binding to remote objects

The client stub class generated from the interface definition is used by the client application 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, called a *client proxy object*, to bind to a remote object. See Chapter 4, "Developing Encina++/DCE applications," on page 17 for information on binding in DCE.

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 makes a remote procedure call (RPC), invoking the corresponding method on the remote object.

Terminating a client application

To terminate a client, you must call the **Encina::Client::Exit** function. The client application must call the **Encina::Client::Exit** function as the last statement of the **main** routine, as shown in Figure 4. If you need to terminate the client at any other time, you also use the **Encina::Client::Exit** function. The example in Figure 4 shows the function being used to terminate a client application when binding to a remote object fails. 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 the underlying Encina services for the client application. All of the transactions in progress are aborted before the application exits.

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

Writing server applications

This section describes the basic functionality that is normally performed by an Encina++ server. Encina++ servers normally perform the following tasks:

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 as required.

Each step is described in the following sections. The typical steps used by a simple server are illustrated in the example in Figure 5.

```
int main(int argc, char *argv[])
{
  // Process command-line arguments
   ...
  // Create and initialize the server
  Encina::Server server;
  server.Initialize();
  // Initialize other Encina components
   ...
  // Create server objects
   ...

  try {
     // Listen for incoming RPCs
     server.Listen(Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS);
  }
  catch (...) {
     cerr << "An exception was raised." << endl;
     exit(1);
  }
  exit(0);
  }
```

Figure 5. Example of initializing an Encina server

The server program examples in this and other chapters use the **Encina::Server** class to initialize and terminate a server application, register resources, and listen for RPCs. For Encina++/DCE servers that operate in a Monitor environment only, you can also use the **EncinaMonitorServer** class to initialize and terminate a server, register resources for that server, and listen for RPCs. This class, which is publicly derived from the **Encina::Server** class, provides an overloaded **EncinaMonitorServer::RegisterResource** function registers and opens a resource manager and associates the name of the resource manager with its open and close strings and its thread-of-control agreement.

Creating a server instance

The first step for the server application is to create an instance of the **Encina::Server** class to represent the application server. The class constructor takes no arguments, as shown in Figure 5 on page 11.

Creating the server instance initializes the runtime attributes of the server. Your application uses this instance to manage the server; only one instance is permitted per server application.

We strongly recommend that you create an instance of the **Encina::Server** class inside **main**, as shown in Figure 5 on page 11. If you need to create a global instance of the **Encina::Server** class, use the **new** operator. These restrictions also apply to the **OtsServer** class (version 2.0 of Encina).

Registering resources

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. The example in Figure 6 illustrates the use of this function. The **Encina::Server::RegisterResource** function takes the following arguments:

- *xaSwitchP*—This argument identifies the XA switch structure used by the resource manager.
- *openString*—This argument specifies a character string of information that is specific to the resource manager and is passed in **xa_open** calls.
- *closeString*—This argument specifies a character string of information that is specific to the resource manager and is passed in **xa_close** calls.
- *isThreadAware*—This argument specifies whether the resource manager library can accommodate multithreaded applications. A nonzero value indicates that the library is thread aware; a value of `0` (zero) indicates that it is not thread aware.

```
int main(int argc, char **argv)
{
   const char* open;
   const char* close;
   struct xa_switch_t* xaSwitch;
   // Create the server
   Encina::Server server;
   // Perform database initialization and get the XA switch and
   // the open and close strings for the resource
   ...
   // Register a resource with the server
   server.RegisterResource(xaSwitch, open, close, 0);
   // Initialize the server
   server.Initialize();
   ...
}
```

Figure 6. Registering a resource with an Encina server

If you use the **EncinaMonitorServer** class instead of the **Encina::Server** class to instantiate a server instance, you can use the **EncinaMonitorServer::RegisterResource** function instead of the **Encina::Server::RegisterResource** function to register a resource. The **EncinaMonitorServer::RegisterResource** function requires the following information:

- The XA switch structure used by the resource manager being registered.
- The name associated with a resource manager's open and close strings and its threading agreement. This name is associated with the resource manager administratively.

If no XA-compliant resources are required for the application but you still want the server to be recoverable, you can use the **Encina::Server::RegisterRecoveryServices** function instead. Both of these functions are optional.

Accessing relational databases

Accessing a relational database management system (RDBMS) from an Encina++ application is no different from accessing an RDBMS from an Encina application. You can use programming statements in the Structured Query Language (SQL), or use the application programming interface (API) native to the relational database, or both. Modules containing SQL statements require precompiling before being compiled along with the other modules of the application. Because the SQL precompilers of many database systems do not support C++, modules that include SQL can be written in C and compiled separately before linking them in with the rest of the application.

Any database access must be done in the context of a transaction. Subtransactions are not supported by the XA interface. Therefore, work with a database is done following these steps:

1. Connect to the database.
2. Begin the transaction.
3. Access the data.
4. Commit, suspend, or abort the transaction.
5. Repeat Steps 2 to 4 as many times as you like.
6. Close the database connection.

See the *Encina Monitor Programming Guide* for further information about accessing relational databases.

Initializing a server

After you have created the server instance and registered any required resources, the next step is 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 the initialization of the 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 involves the use of transactions or RPCs before the server begins listening for RPCs.

| "Initializing a Monitor server" describes how to initialize a server.

Initializing a Monitor server

Encina++/DCE servers can be developed to run under the control of the Encina Monitor. The Monitor provides features that simplify the development and administration of servers. System administrators start and stop these servers by using Monitor administrative tools such as Enconsole.

Monitor servers obtain startup information automatically from the Monitor environment. This type of server can use the version of the **Encina::Server::Initialize** function that takes no arguments (as shown in Figure 5 on page 11).

Creating server objects

Before a server starts listening for RPCs, it 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. Details on how you create server objects are provided in Chapter 4, "Developing Encina++/DCE applications," on page 17 for Encina++/DCE applications.

Listening for RPCs

After server objects are created, start the server listening for incoming RPCs. Calling the **Encina::Server::Listen** function causes the server to start accepting RPCs from Encina++/DCE clients, 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 reference page for the **Encina::Server::ConcurrencyMode** type for descriptions of the available modes. By default, neither transactions nor transactional RPCs are serialized at the server when the **Encina::Server::Listen** function is called with no concurrency mode specified.

The example in Figure 5 on page 11 shows the function being used to listen for RPCs. The `Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS` concurrency mode specifies that all transactional RPCs (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, specify that the server serialize TRPCs and transactions.

Note: The thread-safety of a database is specified when the resource is registered. This threading agreement applies only to requests originating from clients. If the server starts local transactions that also issue XA calls, these XA calls are not governed by the threading agreement. Because the server-side calls are not govered by the threading agreement, under certain circumstances, mixing client- and server-side transactions can result in deadlocks in the database. Specifically, such deadlocks can arise in an application server that uses `SERIALIZE_TRPCS_AND_TRANSACTIONS` as the concurrency mode and uses FALSE (0) as the thread-awareness value. To avoid these deadlocks, do not mix client- and server-side transactions that access the same data within an application server that uses the concurrency mode `SERIALIZE_TRPCS_AND_TRANSACTIONS` and registers a database as non-thread-aware.

If the server application has not already called the **Encina::Server::Initialize** function, the **Encina::Server::Listen** function initializes Encina before the server begins listening. The function assumes that startup information is available from the environment.

Terminating a server

There are two ways to terminate an MAS:

- Calling the ANSI C **exit** function terminates the MAS but does not change the desired state of the MAS. Therefore, if an MAS terminates because of an error condition, the node manager can restart that server. Calling the ANSI C **exit** function affects only the processing agent (PA) in which it is called; all other PAs associated with the server are unaffected. Figure 5 on page 11 demonstrates use of this function to terminate a server.
- Calling the **Encina::Server::Exit** function terminates the MAS and also changes the desired state of the MAS to stopped. All PAs associated with the server are also stopped. If you need to forcibly terminate and stop the server application programmatically at any time, you can use the **Encina::Server::Exit** function. The **Encina::Server::Exit** function never returns. The function takes one argument, which is an integer status value that is returned to the calling environment.

If you used the **new** operator to create a global instance of the **Encina::Server** class (as described in "Creating a server instance" on page 12), you must use the **delete** operator to explicitly delete it when the server shuts down. Otherwise, the server possibly does not terminate properly. This also applies to global instances of the **OtsServer** class (version 2.0 of Encina).

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

Handling errors

Errors are handled in Encina++ by using exceptions. Exceptions provide a way of returning error information back through multiple levels of procedure or function calls, propagating this information until a function or procedure is reached that can respond appropriately to the error. Rather than testing status values to detect errors, Encina++ applications use the C++ exception-handling mechanism to throw (raise) and catch exceptions when error conditions occur.

| The exception class **OtsExceptions::Any** is the base class for all Encina++/DCE
| system exceptions that are thrown and caught when system errors occur. Exception
| classes enable application-specific exceptions to be thrown by servers and caught
| by clients. C++ exceptions can be used to handle application-specific errors local to
| the client or server. When using Transactional-C++ (see Chapter 6, "Transaction
| processing with Transactional-C++," on page 41), an uncaught exception aborts a
| transaction.

| The following sections describe how exceptions can be thrown and caught in
| Encina++ applications. "Using exceptions in Encina++/DCE" on page 33 describes
| how to use exceptions in a DCE environment.

Throwing exceptions

In Encina++, the C++ **throw** statement is used to throw exceptions. For example, if an exception named **insufficient_funds** is defined in the interface definition file for the **account** interface, a server manager function for the interface can throw the **insufficient_funds** exception as shown in Figure 7.

```
if (balance < amount)
    throw insufficient_funds();
```

Figure 7. Throwing a user-defined exception

Catching exceptions

In Encina++, C++ **try** and **catch** blocks are used to catch exception. For example, you can catch an exception named **insufficient_funds** defined in the interface definition file for the **account** interface as shown in Figure 8.

```
try {
    // call to one or more functions that can throw the
    // insufficient_funds exception
}
catch(account::insufficient_funds){
    // error handling for the named exception
}
catch(...){
    // error handling for uncaught exceptions
}
```

Figure 8. Catching exceptions in Encina++

Chapter 4. Developing Encina++/DCE applications

This chapter describes how to develop a distributed application by using Encina++ for the Distributed Computing Environment (DCE). It provides information about generating client and server stub files and writing client and server applications that is specific to Encina++/DCE. Refer to Chapter 3, "Developing distributed applications," on page 9 for more general information on developing distributed applications with Encina++.

Introduction to Encina++/DCE

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

In the DCE environment, the Transactional Interface Definition Language (TIDL) must be used to specify object interfaces in the form of remote procedures. The remote procedures are used for communications between the client and server applications. The interface definition in the TIDL file specifies whether each remote procedure for that interface is executed transactionally.

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 of the client stub class determines the binding method used. The instance is referred to as a *client proxy object*. The client proxy object can be bound to one of the following:

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

The C++ exception-handling mechanism is used to integrate DCE exceptions into Encina++/DCE. Encina++/DCE applications can throw and catch DCE and Encina system exceptions when error conditions occur. In addition, Encina++/DCE includes the ability to specify user-defined DCE exceptions. A *user-defined exception* is an exception defined by the application developer for handling application-specific errors. A user-defined exception can be thrown by the server and caught by the client to indicate that an error occurred during the execution of an RPC.

Defining the interface

This section briefly describes TIDL and the TIDL compiler. It also illustrates how to use the TIDL compiler to generate stub files for client and server applications. Note that this section documents only special requirements for using TIDL with Encina++/DCE; refer to the *Encina Transactional Programming Guide* for additional details on TIDL.

Using TIDL with Encina++

TIDL is used to define interfaces for the objects in an Encina++/DCE application. TIDL is an interface definition language, similar to the DCE IDL, that defines operations between a client and server. Unlike IDL, however, TIDL allows those operations to be specified as either transactional or nontransactional. When the TIDL compiler processes a TIDL file, the compiler produces code that includes transactional semantics.

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. The TIDL compiler is invoked with the **tidl** command; the **-ots** option must be specified on the command line so that C++ stub files, rather than C stub files, are generated for the interfaces described in the TIDL file.

On the server side, two server stub classes are generated as shown in Figure 9 on page 19. 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. For example, if you define an interface named **Order** in a TIDL file, the TIDL compiler generates the an abstract server stub class named **OrderMgrAbstract** and a concrete server stub class named **OrderMgr**.

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. However, the server application developer can also derive his own class from the abstract class and then implement the remote procedures for the interface as member functions of this derived class. The server application can then instantiate objects of the derived class and make those objects available to clients.

On the client side, a *client stub class* is generated (also shown in Figure 9 on page 19). The client stub class has the same name as the interface. For example, if you define an interface named **Order**, the TIDL compiler generates a client stub class also named **Order**.

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 implementation 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.

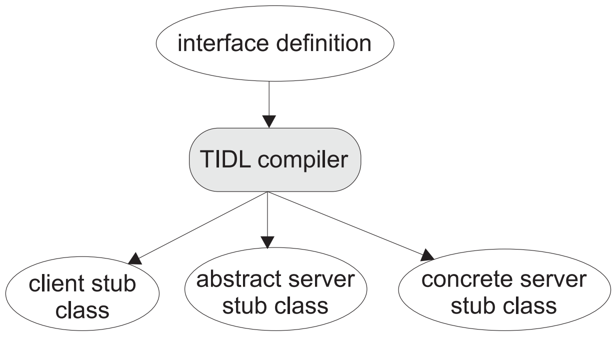

Figure 9. Encina++/DCE client and server stub classes

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

Making operations transactional

To define an operation as transactional for an Encina++/DCE application, you declare it in the TIDL file for an interface. In Figure 10, an example TIDL file for the **account** interface defines two operations as transactional and one as nontransactional.

```
[uuid(002068a4-f049-1b28-bdfc-c037cf6a0000), version(1.0)]
interface account{
    [transactional] void debit ([in] float amount);
    [transactional] void credit ([in] float amount);
    [nontransactional] float QueryBalance(void);
}
```

Figure 10. Sample TIDL declaration for transactional operations

The **transactional** keyword preceding the **credit** and **debit** function definitions specifies that the functions are executed transactionally. TIDL assumes that interface operations are transactional; therefore, the **transactional** keyword is optional. However, you must use the **nontransactional** keyword to specify that an operation is *not* transactional, as is the case with the **QueryBalance** function.

Generating stub files

This section describes how to generate the client and server stub files for Encina++/DCE client and server applications. The process is described using **account.tidl** as the name of an example TIDL interface definition file. Note that the files with **.C** and **.H** extensions denote C++ source and header files.

The following steps, illustrated in Figure 11 on page 20, are used to generate the client and server stub files:

1. Execute the **tidl** command. Use the **-ots** option and pass the **account.tidl** file as the argument to the command. The compiler produces the following files: **accountTC.C**, **accountTS.C**, **accountTC.H**, **accountTS.H**, and **account.idl**.
2. Run the **idl** compiler. Use the **-no_mepv** and **-cepv** options and pass the IDL interface definition file **account.idl** as the argument to the command. The compiler produces the following files: **account_cstub.c**, **account_sstub.c**, and **account.h**.

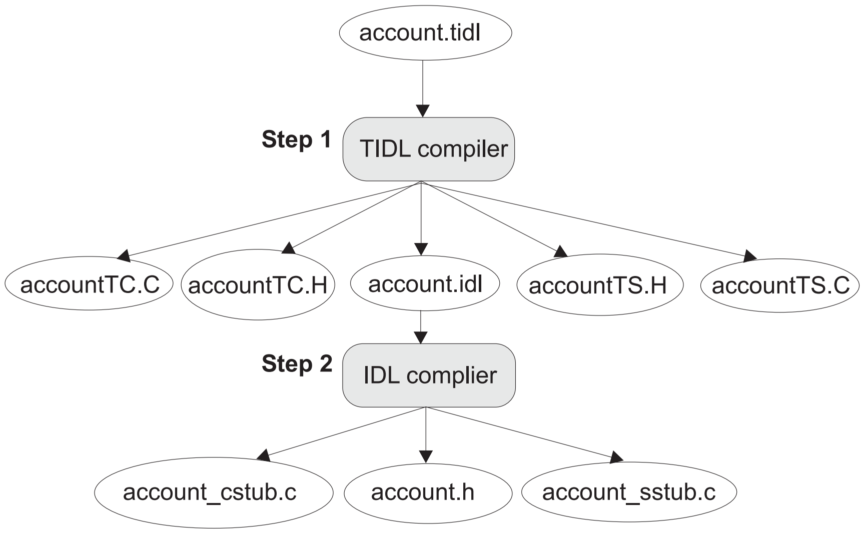

Figure 11. Generating Encina++/DCE client and server stub files

The TIDL compiler produces different header files for the client and server; the IDL compiler produces only one header file. While all of these files are required by the client and server, you need include only one file in your client and server source files. The source files for the client must include **accountTC.H**, and the source files for the server must include **accountTS.H**.

Binding to remote objects

The first time a member function call is made on the client proxy object, the proxy object is bound to an implementation object and the call is passed to that implementation object. The client then communicates with the implementation object by using the client proxy object.

The client stub class generated by the TIDL compiler is used by the client application to locate and bind to remote objects or servers that export the requested interface. The client stub class is derived from the **OtsBinding** class. The **OtsBinding** class is an abstract base class that provides binding functionality for all client stub classes. The member functions defined in the client stub class map to the remote procedures specified in the TIDL file. Creating a client proxy object causes the proxy object to bind to a remote object. When a member function call is made on the proxy object, the proxy object makes an RPC and invokes the corresponding method on the implementation object to which the proxy object is bound.

A client stub class is generated for each defined interface. The form and method of binding between client proxy objects and implementation objects are determined by the constructor used to create an instance of the client stub class. The generated client stub class defines several constructors. For example, given an interface named **account**, the following constructors are defined automatically for the client stub class **account**:

- **account**(void)—This constructor binds the client proxy object transparently to any implementation object that offers the interface.
- **account**(char **objectName*)—This constructor binds the client proxy object to the implementation object specified by the *objectName* parameter.

- **account**(OtsServerName &*server*)—This constructor binds the client proxy object to a Monitor application server that exports the interface; the name of the server must be specified. This binding method can be used to access non-Encina++ servers.
- **account**(ObjectRef **objectRef*)—This constructor binds the client proxy object to an implementation object by using an object reference. The object reference specified in the *objectRef* parameter identifies a particular object in a particular server. Binding by object reference is discussed in “Binding by object reference” on page 29.

For information on how the binding techniques are supported by DCE naming, see “Servers and objects in CDS” on page 27.

Note: Encina++ Monitor clients do not reuse the client proxy object’s binding handle by default. Instead, it uses a cache hierarchy to retrieve a new binding handle for each RPC. To ensure that Encina++ does reuse the proxy handle rather than obtaining a new handle, set the `ENCINA_OTS_USE_SAME_OBJECT` environment variable to TRUE.

Figure 12 shows an example client program that binds to any remote implementation object that exports the appropriate interface. The constructor has no arguments. Once the client proxy object binds to the implementation object, the proxy object can be used to invoke remote procedures, as the call to the example **debit** function demonstrates. (Note that because this function is transactional, it must be called within the scope of a transaction, as indicated by the comment lines. For more information on transactions see Chapter 5, “Transaction processing overview,” on page 39.)

```
int main(int argc, char *argv[])
{
   // Process command-line arguments and prompt user for instructions
   ...
   // Initialize the client
   Encina::Client::Initialize();
   // Create an account proxy object that binds to any remote
   // account object exported by a server
   account account1;
   // Begin transaction
      // Perform work
      ...
      try {
         // Call a member function of the account class to bind to
         // a remote account object
         account1.debit(amount);
      }
      catch (...) {
         cerr << "An exception was raised." << endl;
         Encina::Client::Exit(1);
      }
      ...
   // End transaction
   Encina::Client::Exit(0);
}
```

Figure 12. Example of binding a client to any remote implementation object that offers a particular interface

Figure 13 on page 22 shows an example of a client program that binds to a specific implementation object. The constructor takes the name of the implementation object, indicating that the client proxy object binds to any implementation object named **accountMgrObj** that exports the appropriate interface. Figure 18 on page 26 shows an example of the server program required by a client using this example client program.

```
int main(int argc, char *argv[])
{
   // Process command-line arguments, prompt the user for instructions,
   // and initialize the client
   ...
   // Create an account object that binds to a specific remote
   // account object exported by a server
   account account1("accountMgrObj");
   ...
}
```

Figure 13. Example of binding a client to a specific implementation object

Figure 14 shows an example of a client program that binds to any implementation object on the specified server that exports the appropriate interface. The constructor takes the full DCE pathname of the server containing the implementation object to which the client proxy object binds.

```
int main(int argc, char *argv[])
{
   // Process command-line arguments, prompt the user for instructions,
   // and initialize the client
   ...
   // Create an account object that binds to a specific server that
   // exports the account object
   account account1(OtsServerName("/.:/acme/server/AccountServer1"));
   ...
}
```

Figure 14. Example of binding a client to an implementation object on a specific server

The client stub class also defines a member function for each operation defined in the interface. The actual binding between a client proxy object and implementation object occurs the first time one of these member functions is invoked on the client proxy object. If the member function is defined as transactional in the TIDL file, the function must be called within the scope of a transaction. (See Chapter 5, "Transaction processing overview," on page 39 for more information on using transactions.)

Because binding occurs when a member function is called, the member functions for the client stub class are defined to throw the following exceptions if an attempt at binding fails:

- **OtsExceptions::ObjectNotFound—**The named object or an instance of the specified class was not found.
- **OtsExceptions::NameServiceError—**The name service is inaccessible or the principal does not have permission to query the name service.

After a client is bound to an implementation object, the following exceptions can be thrown when a member function defined in the proxy class is called:

- **OtsExceptions::TranAborted—**A transaction containing a transactional RPC aborted at the server.
- **OtsExceptions::ServerShutdown—**The server is being terminated when a request is made.

- **OtsExceptions::PermissionDenied—**The principal does not have permission to execute the requested operation.

If you specify any user-defined exceptions for the interface, they can be thrown for each member function as well.

Developing client applications

This section covers the DCE-specific issues involved in writing Encina++/DCE client applications. It describes how the client application developer can use the client stub class generated by the TIDL compiler to bind to objects exported by the server. It also describes how to build and run an Encina++/DCE client.

Building clients

A simple client application is made up of the following:

- A source file for the client that initiates remote requests; this file must include the TIDL-generated header file containing the client class definition corresponding to the interface name.
- A client stub file generated by the TIDL compiler.
- A client stub file generated by the DCE IDL compiler.

Using the example filenames shown in Figure 11 on page 20, Figure 15 illustrates the process used to build Encina++/DCE client applications. In the first part of the process, the following source files must be compiled:

- **client.C**, which is the client program that initiates RPCs. This file must include the **accountTC.H** header file (which in turn includes the **account.h** header file) and the appropriate Encina header files described in “Compilation issues,” on page 71.
- **accountTC.C**, which is the client stub file generated by the TIDL compiler.
- **account_cstub.c**, which is the client stub file generated by the DCE IDL compiler.

After compilation, the resulting object files must be linked with the appropriate Encina, DCE, and platform-specific library files described in “Compilation issues,” on page 71.

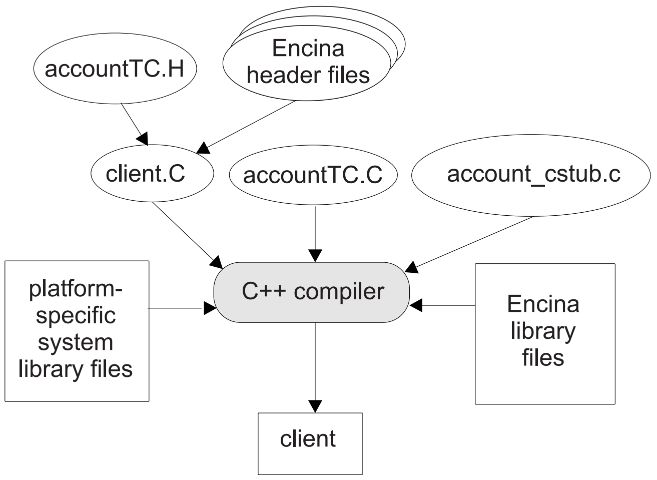

Figure 15. Building Encina++/DCE clients

Running clients

Before you run an Encina++/DCE client application, you must set the value of the ENCINA_TPM_CELL environment variable to specify the name of the Monitor cell containing the servers to which the client binds.

To set ENCINA_TPM_CELL in a C shell, specify the fully qualified DCE pathname for the Encina Monitor cell (for example, **/.:/mycell**) as follows:

```
setenv ENCINA_TPM_CELL /.:/mycell
```

Note: If you use the **EncinaMonitorClient** class and the **EncinaMonitorClient::Initialize** function (with the cell name specified), you do not need to set this environment variable.

Developing server applications

This section covers the DCE-specific issues involved in writing Encina++/DCE server applications. It documents the server stub classes generated by the TIDL compiler and describes how the server application developer can use the server stub class to implement a server interface and create implementation objects. It also describes how to build and run an Encina++/DCE server.

An Encina++/DCE server must do the following:

- Implement the manager functions; that is, it must contain the body of the functions defined in the interface. "Implementing manager functions" provides information on this task.
- Create an implementation object to which client proxy objects can bind to initiate RPCs. "Creating implementation objects" on page 26 provides information on this task.

Implementing manager functions

Manager functions contain the business logic that implement the procedures defined in an interface. When a call on a client proxy object results in an RPC, the corresponding manager function is executed. The server classes generated by the TIDL compiler are used by the server application developer as the framework for implementing the manager functions.

The server classes

TIDL generates two classes for each interface that you define: an abstract server class and a concrete server class. Use these classes as follows:

- If you do *not* have to create new functions or variables for your implementation class, you can use the concrete class directly and implement functions of this class.
- If you do need to create new functions or variables for your implementation class, you must derive your implementation class from the abstract class and implement the functions in this derived class.

For example, given an interface named **account**, TIDL generates an abstract server class named **accountMgrAbstract** and a concrete server class named **accountMgr**, as shown in Figure 16 on page 25.

The concrete server class is derived from the abstract server class. The concrete server class is provided to simplify the task of the server application developer. You can use the class to implement the manager functions for the server interface or to create classes that are derived from either the abstract or concrete stub class.

The abstract server class is the base server class for the interface. You cannot create an object of this class. However, you can derive a class from it and then implement manager functions using this derived class and create objects of this derived class. The abstract server class is itself derived from the **OtsInterfaceMgr** class, which provides functionality for creating implementation objects and exporting them to the namespace.

The **myAccountMgr** class derived from the **accountMgrAbstract** class in Figure 16 is not generated by TIDL. You typically define such a class in order to specialize the functionality defined in the generated abstract class from which it is derived.

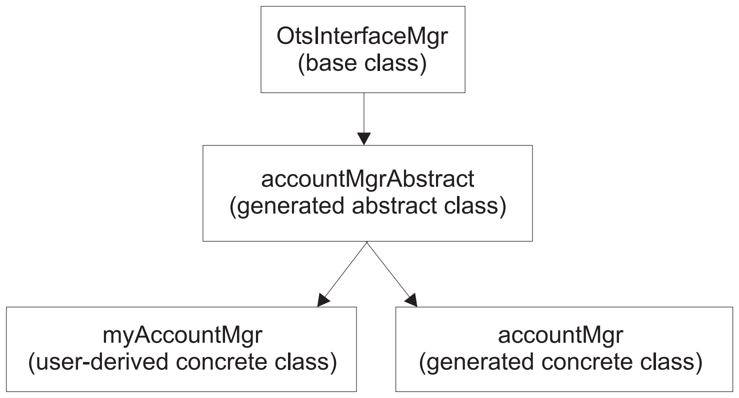
OtsInterfaceMgr
(base class)
accountMgrAbstract
(generated abstract class)
myAccountMgr
(user-derived concrete class)
accountMgr
(generated concrete class)

Figure 16. Server class hierarchy

The manager functions

If you want to implement only the interface functions as defined in the generated classes, use the concrete class. If you need to modify the class (to define additional member variables, for example) or implement an implementation object that exports multiple interfaces, you must define a class that is derived from the abstract class.

Figure 17 shows an example of using the concrete server class (the **accountMgr** class), to implement the manager functions for the **account** interface (**debit**, **credit**, and **QueryBalance** specified in Figure 10 on page 19). Note that parameters of the function are defined by using the TIDL data types.

In this example, the manager functions are implemented in a separate file named **manager.C**.

```
void accountMgr::credit(idl_short_float amount) {
   // Perform credit operations
   ...
}
void accountMgr::debit(idl_short_float amount) {
   // Perform debit operations
   ...
}
idl_short_float accountMgr::debit(void) {
   // Perform QueryBalance operations
   ...
}
```

Figure 17. Implementing the manager functions of the account interface

Creating implementation objects

In both the abstract and concrete stub classes that are generated by the TIDL compiler, two constructors are defined automatically. These constructors are used to create and register implementation objects to which client proxy objects can bind. The difference between the two constructors is the way in which a universal unique identifier (UUID) is assigned to the created implementation object:

- If the constructor does not take an object UUID as an argument, Encina++ creates a UUID and assigns it to the implementation object.
- If the constructor takes an object UUID as an argument, that UUID is assigned to the implementation object.

Note: The server-object constructors defined in the generated files include two additional parameters by default: the *defaultAcl* parameter and the *numThreads* parameter. These two parameters are not implemented for this release of Encina++; values specified for these parameters are ignored.

In general, most applications can create implementation objects without specifying object UUIDs for them. Situations in which you possibly need to assign a specific object UUID to an implementation object are rare.

In the example **account** interface, two identical constructors are defined for the **accountMgrAbstract** class and for the **accountMgr** class. Figure 18 provides an example of creating an implementation object of the **accountMgr** class. The object name, **accountMgrObj**, is the name used by the client application if the client needs to bind to a specific implementation object as described shown in Figure 13 on page 22. If no name is passed, an unnamed object is created.

```
int main(int argc, char *argv[])
{
   // Process command-line arguments
   ...
   // Create a server object
   Encina::Server server;
   // Initialize the server
   server.Initialize();
   // Create an accountMgr object to which client account objects can bind
   accountMgr account1("accountMgrObj");
   // Listen for incoming RPCs
   server.Listen(Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS);

   ...
}
```

Figure 18. Creating an implementation object for the accountMgr class

Figure 19 on page 27 shows creation of an implementation object of the **accountMgr** class where the UUID associated with the object is specified by the application.

```
int main(int argc, char *argv[])
{
   // Process command-line arguments, create a server object, initialize the server
   ...
   // Create an accountMgr object to which client account objects can bind
   accountMgr account1("cb5f8b98-ba8c-11d1-83f8-9e6204baa77", "accountMgrObj");
   ...
}
```

Figure 19. Creating an implementation object for the accountMgr class with UUID specified

Servers and objects in CDS

Encina++ clients can bind to servers in four different ways:

- By server name
- By object name
- By interface
- By object reference

This section describes the CDS entries used to support binding by server name, interface, and object name. See “Binding by object reference” on page 29 for more information on binding by object reference.

All Encina Monitor application servers register RPC binding entries with CDS. Monitor application servers consist of a group of processing agents (PAs), and each PA is a distinct process. Each PA registers its binding entry with CDS; the location of the entry is determined by the name of the PA, which has the following form:

`/.:/`*`monitorCellname`*`/server/`*`serverName`*`_pa`*`Number`*

The *monitorCellName* is the name of the Monitor cell in which the server runs. The *serverName* is the name configured for the group of PAs. *Number* is the number assigned to the PA when it starts; the numbering for PAs starts at 0.

The binding entry under a the name of a PA contains the information needed by a client to contact that PA. Each Monitor server also registers an RPC group under the name of the server, which has the following form:

`/.:/`*`monitorCellname`*`/server/`*`serverName`*

This group contains the names of each PA for the server. This group is used by clients that bind to a named server. A client looks up the name of the server and obtains the binding entry for a PA. Internally, the Monitor uses the name of the server to select a PA and uses the name of the PA to retrieve the binding entry for the PA.

Encina++ servers that create named objects also register information in the CDS `/.:/`*`monitorCellname`*`/objects` directory and with the endpoint mapper. Servers that create unnamed objects register them only with the endpoint mapper. For servers that create named objects, two distinct types of entries are stored in the CDS `/.:/`*`monitorCellname`*`/objects` directory:

- *Object groups:* RPC groups named by a combination of the UUID of the interface and the name of the object
- *Interface groups:* RPC groups named by the UUID of the interface

The name of an object group consists includes the interface UUID, a colon (:), and the name of the object, as follows:

`/.:/`*`monitorCellname`*`/objects/`*`interfaceUUID`*`:`*`objectName`*

This entry contains the names of the all the PAs that support the combination of interface and named object. The UUID of the object is also stored with the group. This group is used by clients that bind to a named object.

The name of an interface group includes the UUID followed by a colon (:), as follows:
`/.:/`*`monitorCellname`*`/objects/`*`interfaceUUID`*`:`

This entry contains the names of the object groups that share the interface UUID. This group is used by clients that bind by interface.

For example, suppose an application makes use of two interfaces, **merchandise** and **merchandise_admin**. The application provides two single-PA application servers, **server1** and **server 2**; **server1** exports the **merchandise** interface, and **server2** exports both interfaces. Specifically, **server1** creates a named object, **merchOjb1**, that supports the **merchandise** interface. The other server, **server2**, creates two named objects; **merchOjb2** supports the **merchandise** interface, and **adminObj** supports the **merchandise_admin** interface.

These servers create three object groups and two interface groups in the CDS `/.:/`*`monitorCellname`*`/objects` directory.

One object group is created for each combination of interface and object name, and each group contains the names of all PAs that support the interface-object combination. For **server1**, which supports a single interface, and **server2**, which supports two interfaces, a total of three object groups is created, as follows:
- For **server1**: *merchandiseUUID***:merchObj1**. This group contains the CDS name `/.:/`*`monitorCellName`*`/server/server1_pa0`
- For **server2**:
  - *merchandiseUUID***:merchObj2**, which contains the CDS name `/.:/`*`monitorCellName`*`/server/server2_pa0`
  - *merchandise_adminUUID***:adminObj**, which contains the CDS name `/.:/`*`monitorCellName`*`/server/server2_pa0`

One interface group is created for each interface exported in the application. For **server1**and **server2**, which collectively support the **merchandise** interface and the **merchandise_admin**, two interface groups are created, as follows::
- For the **merchandise** interface, the group *merchandiseUUID***:**. This group refers the following object groups:
  - *merchandiseUUID***:merchObj1**
  - *merchandiseUUID***:merchObj2**
- For the **merchandise_admin** interface, the group *merchandise_adminUUID***:**. This group refers to the object group *merchandise_adminUUID***:adminObj**

A client binding by interface to the **merchandise** interface uses that interface group to select an object group, from which an appropriate PA is selected. A client binding to the object named **adminObj** chooses that object group, from which an appropriate PA is selected.

Building servers

A simple server application is made up of the following:

- A source file for the server that listens for remote requests; this file must include the header file defining the implementation class for the server interface.
- A source file that implements the functions for the server interface.
- A server stub file generated by the TIDL compiler.
- A server stub file generated by the IDL compiler.

Using the example filenames shown in Figure 11 on page 20, Figure 20 illustrates the process used to build Encina++/DCE server applications. In the first part of the process, the following source files must be compiled:

- **server.C**, which is the server program. This file must include the **accountTS.H** header file (which in turn includes the **account.h** header file), the **manager.H**, and the appropriate Encina header files described in "Compilation issues," on page 71.
- **manager.C**, which is the file containing the source code that implements the manager functions.
- **accountTS.C**, which is the server stub file generated by the TIDL compiler.
- **account_sstub.c**, which is the server stub file generated by the DCE IDL compiler.

After compilation, the resulting object files must be linked with the appropriate Encina, DCE, and platform-specific library files described in "Compilation issues," on page 71.

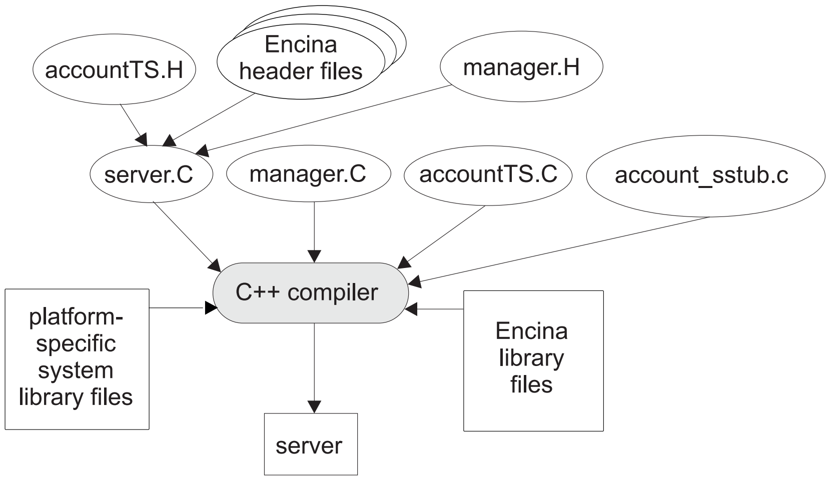

Figure 20. Building Encina++/DCE servers

Running servers

Encina++/DCE Monitor server applications are typically started by using Encina administrative tools such as Enconsole. Refer to the *Encina Administration Guide Volume 1: Basic Administration* for information on using Encina's administrative tools.

Binding by object reference

Implementation objects can be created dynamically, that is, on demand by a client. In this way, a server does not need to instantiate many different but infrequently used objects. For example, in a banking application where each account is represented by an account object, there are many accounts, but each individual account is seldom accessed. Binding by object reference enables the server to instantiate an object for an account as needed.

A *factory* is an interface whose manager functions create and delete objects of another interface class. The procedure for binding by object reference includes the following steps:

- The server exports a *factory object* rather than an implementation object.
- The client creates a factory proxy and binds to the server factory object.
- The client requests the creation of a server object.
- The server returns an object reference.
- The client creates another proxy and binds by reference to the server object.
- The client tells the factory to delete the server object.

Note: The following sections describe the mechanics of writing the factory interface and factory functions. The application must manage the state of these created objects and do garbage collection as required.

Writing the factory interface

To add factory support to an application, you must write an additional factory interface for each application interface. Your application, therefore, can require several additional factory interfaces. Like other interface definitions, a factory interface includes a UUID, a version number, a name, and a set of function prototypes. Apart from the UUID and the name, the factory interface definitions includes:

- A function to create an object and return its reference
- A function to destroy an object

The factory interface must also import the file **ots/dce/otsObjectRef.idl**. An example factory interface (**acctFactory.tidl**) is shown in Figure 21.

```
/* TIDL file for an Encina++ factory interface

[uuid(653853c4-3210-11d0-936205c9aa77),
 version(1.0)]
interface acctFactory
{
import "ots/dce/otsObjectRef.idl";
ObjectRef* Create(
   [in] handle_t handle,
   [in, out] uuid_t *createdObject);
void Delete(
   [in] handle_t handle,
   [in] uuid_t createdObject);
}
```

Figure 21. An example factory interface definition

As with any application interface, the client and the server programs must include header files to create instances of each factory object or client factory proxy object. The factory interface header files are generated by using the TIDL compiler with the **-ots** option, as is done with application interfaces, and by running the IDL compiler on the IDL file. In our example, the header file for the client is **acctFactoryTC.H**. It contains a class called **acctFactory**, comprising class constructors for creating client factory proxy objects and a member for each function in the factory interface (in our example, **Create** and **Delete**). The server-side header file is **acctFactoryTS.H**. It contains two classes (one abstract class and one concrete class), in our example, **acctFactoryMgrAbstract** and **acctFactoryMgr**. It also includes two class constructors (one by name only and one by name and UUID), and a member for each function in the factory interface.

Writing functions to create and delete objects

The next step is to implement the factory interface manager functions in the server program. Every interface derives from the class **otsInterfaceMgr**, which contains (among other methods) **GetObjectRef** and **GetUuid**. These functions are members of any interface and are used by a factory to identify the object and to associate the UUID with an **ObjectRef** structure. In our example, the **acctFactoryMgr::Create** function is implemented as shown in Figure 22. The function takes an optional UUID (or creates one), and returns a reference to an object of the type **accountMgr** (a server class derived from the **account** interface shown in “Making operations transactional” on page 19).

```
ObjectRef* accountFactoryMgr::Create(uuid_t* id)
{
   accountMgr  *acctObject;
   unsigned32  ignore;
   //If no UUID is passed in, create one.
   if(uuid_is_nil(id, & ignore)){
      acctObject = new accountMgr();
      Uuid uuid = acctObject->GetUuid();
      memmove(id, (uuid_t*) uuid, sizeof(uuid_t));
   }
   //Otherwise, use the UUID passed in.
   else {
      Uuid *objP = new Uuid(id);
      acctObject = new accountMgr(*objP);
   }
   return (&acctObject->GetObjectRef());
}
```

Figure 22. Example of a function to create an object

To delete an object, a factory retrieves the UUID of the object and then deletes the object, as shown in Figure 23.

```
void acctFactoryMgr::Delete(uuid_t id)
{
   accountMgr *acctObject =
       (accountMgr *) GetObject(id);
   delete acctObject;
}
```

Figure 23. Example of a function to delete an object

Supporting factories in the server program

To use a factory, a server must include the factory interface header file and declare the factory object during initialization (called **AccountFactory** in Figure 24 on page 32).

```
/* Header files ... */
#include <accountTS.H> /* application interface */
#include <acctFactoryTS.H> /* factory interface */;
[...]
int main(int argc, char **argv)
{
   [...]
   try{
      //Declare a factory object instead of an implementation object.
      acctFactoryMgr acctFactory("AccountFactory");
   [...]
   }
    catch{
   [...]
   }
   return 0;
}
```

Figure 24. Declaring the factory object in the server program

Supporting factories on the client

The client program must include the header files for any factory interface and use factory objects in the function prototypes. The client program then binds to the server's exported factory object (in our example, **AccountFactory**, shown in Figure 24), creating a client factory proxy object (**acctCreator**), as follows:

```
int main(int argc, char **argv)
{
   [...]

         //Declare a proxy factory object
             acctFactory acctCreator(
             "AccountFactory");

}
```

The example function definitions shown in Figure 25 are in the client program. The **CreateRemoteObject** function calls the factory's **Create** function and returns an object reference. The **DeleteRemoteObject** function calls the factory's **Delete** function and returns nothing.

```
ObjectRef  *CreateRemoteObject(
         acctFactory *factoryP,
         uuid_t *objectUuidP)
{
    ObjectRef *ref =
             factoryP->Create(objectUuidP);
    return(ref);
}
void DeleteRemoteObject(
         acctFactory *factoryP,
         uuid_t *objectUuid)
{
   factoryP->Delete(objectUuid);
}
```

Figure 25. Requesting and deleting server objects

Figure 26 on page 33 shows how these functions appear in the client program. Before an RPC is a made to a server manager function, the client program calls the

CreateRemoteObject function to request that the factory create an instance of the appropriate application interface class; the function returns an object reference. The client program uses the object reference to declare the client proxy object. The binding occurs when the server manager function is called. After doing any tasks required by the application, the client calls the **DeleteRemoteObject** function to destroy the object, and the client program frees the storage used by the object reference.

```
int main(int argc, char **argv)
{
    //Process command line arguments and prompt user for instructions
     ...
    try{

      // Declare a proxy factory object
      // called acctCreator
      acctFactory acctCreator(
          "AccountFactory");
      // Begin transaction.
      // Perform work
      // Call the factory to create an object and return its identifier
      uuid_t objectUuid;
      ObjectRef *ref = CreateRemoteObject(
          &acctCreator, &objectUuid);
      // Use the object's identifier to declare the client proxy object
     account account1(ref);
      // Call a member function of the account class to bind to a remote account
      // object
      account1.debit(amount);
      // Destroy the factory-created object and free the storage
      // for the reference
       DeleteRemoteObject (&acctCreator, objectUuid);
       rpc_ss_client_free(ref);

      //End transaction
   }
   catch{
        cerr << "An exception was raised." << end1'
        Encina::Client::Exit(1);
   }
   Encina::Client::Exit(0);
}
```

Figure 26. Binding by object reference

Using exceptions in Encina++/DCE

The Encina++ exception classes encapsulate DCE exceptions, enabling exceptions to be thrown by servers and caught by clients to indicate that an error occurred during the execution of an RPC. C++ exceptions can be used to handle application-specific errors local to the client or server. Encina++/DCE user-defined exceptions are implemented as DCE *named* exceptions, enabling the client to recognize an exception thrown by the server by name.

Defining exceptions

To define a named exception for an Encina++ application, declare it in the header of the TIDL file for an interface. When the TIDL file is compiled, it creates classes for the named exception; these classes are derived from the **OtsDceExceptions::UserException** class.

In Figure 27, an example header for the **account** interface declares an exception named **insufficient_funds**. Note that since the original version of the TIDL file did not include the exception, the version number has been increased to 2.0.

```
[uuid(002068a4-f049-1b28-bdfc-c037cf6a0000), version(2.0),
    exceptions(insufficient_funds)]
interface account{
...
}
```

Figure 27. Sample TIDL declaration for a user-defined exception

When the TIDL file for the **account** interface is compiled, it defines an exception class called **account::insufficient_funds** in both the client and server stubs generated by the TIDL compiler. The **account::insufficient_funds** exception can be thrown by the server and caught by the client. See "Handling errors" on page 15 for general information on throwing and catching exceptions in Encina++.

Note: If you use the Encina++/DCE classes to define an exception but do not specify it as a named exception in the TIDL file, the client cannot catch the exception by name if it is thrown by the server. The client can catch the exception by using the **OtsExceptions::Any** class, but named exceptions are usually preferred because they provide finer control over exception handling.

Throwing exceptions

You can throw Encina++ system exceptions in addition to exceptions that you define. For example, as shown in Figure 28, your manager function can throw the **OtsExceptions::TranAborted** exception as an alternative way to abort a transaction.

```
if (invalid_account)
    throw OtsExceptions::TranAborted();
```

Figure 28. Throwing Encina++ system exceptions

Catching exceptions

In Encina++, you can catch user-defined exceptions and system exceptions. Figure 29 shows an example of catching a user-defined exception named **insufficient_funds** or the **OtsExceptions::TranAborted** system exception.

```
try {
    // call to one or more functions that can throw the
    // insufficient_funds or OtsExceptions::TranAborted exception
}
catch(account::insufficient_funds){
    // error handling for the named exception
}
catch(OtsExceptions::TranAborted){
    // error handling for the system exception
}
```

Figure 29. Catching exceptions in Encina++

In addition to catching specific exceptions, the hierarchy of the Encina++ exception classes allows you to catch exceptions by class. You can catch *any* Encina++ system exceptions by specifying the class of the exception from which all Encina++ system exceptions are derived, **OtsExceptions::Any**. For example, you can use the series of **catch** statements shown in Figure 30 to catch a specific Encina++ exception or any Encina++ exception thrown by the server.

```
try {
    // call to one or more functions that can throw an Encina++
    // exception
}
catch(OtsExceptions::TranAborted){
    // error handling for the system exception
}
catch(OtsExceptions::Any){
    // error handling for any Encina++ exception
}
```

Figure 30. Catching Encina++ exceptions by class

If the server throws the **OtsExceptions::TranAborted** exception, it is caught by the first **catch** statement. Any other Encina++ exception is caught by the second **catch** statement. When Transactional-C++ is used (see Chapter 6, "Transaction processing with Transactional-C++," on page 41), an uncaught exception aborts a transaction.

You can get a reference to the exception object caught by specifying a variable name for it in the argument to **catch**. For example, you can get the exception object caught with the **OtsExceptions::Any** exception and print the string associated with the error code as shown in Figure 31.

```
try {
    // call to one or more functions that can throw an Encina++
    // exception
}
catch(OtsExceptions::Any &exception){
    // error handling for any Encina++ exception
    cerr << exception << endl;
}
```

Figure 31. Getting a reference to an exception object

If the server throws an Encina++ exception, the exception is caught by the **catch** statement, and the exception object is returned in the *exception* variable.

Signal handling

Encina has a built-in signal-handling capability. All Encina clients and Monitor-based servers automatically handle the following signals: SIGHUP, SIGINT, and SIGTERM. When an Encina++/DCE client receives any of these signals, it aborts any outstanding transactions and exits. When an Encina++/DCE server receives the SIGINT or SIGTERM signal, it aborts any outstanding transactions and exits; when an Encina++/DCE server receives the SIGHUP signal, it requests a Monitor state dump.

Although it is not recommended, you can create a customized signal handler for an Encina++ application. A customized signal handler can handle any signal. If you use a customized signal handler, the signal handler function must call any previously installed signal handler after the customized signal handler has performed any required work.

Naming in Encina++ Toolkit applications

Encina++/DCE applications typically run within the Monitor environment, but non-Monitor applications can be written. These are referred to as Toolkit applications, and they typically use DCE for naming and security. However, Encina++ Toolkit applications do not require the use of CDS to locate bindings to Encina++ objects. As an alternative to using CDS, clients and servers can share a binding file that defines the location of servers, server-side objects, and server-side interfaces, depending on the type of binding being used in the application. This allows Encina++ applications to be written without relying on the availability of CDS or DCE security.

Using Toolkit mode with CDS

In this mode, an Encina++ server registers itself and its objects with CDS. The following environment variables need to be set for the client and server:

```
setenv ENCINA_TK_MODE 1
setenv ENCINA_CDS_ROOT /.:/CdsRoot
```

where *CdsRoot* is the fully qualified path name of the server. These variables are used in addition to any other environment variables required by the client and server.

Using Toolkit mode without CDS

In this mode, Encina++ servers and clients share a binding file. The following environment variables need to be set for the client and server:

```
setenv ENCINA_BINDING_FILE bindingFile
setenv ENCINA_TK_MODE 1
setenv ENCINA_CDS_ROOT /.:/CdsRoot
```

where *bindingFile* is the path name of the binding file and *CdsRoot* is the fully qualified path name of the server. These variables are in addition to any other environment variables required by the client and server.

The contents of the binding file depend on the type of binding being used in the application, as shown in Table 3. Binding by object reference does not require an entry in the binding file because the application already possesses the required binding.

Table 3. Binding file entries

Type of binding	**Binding file entry**
By object reference	None
By interface	`$ENCINA_CDS_ROOT/interface/`*`interfaceName`* `ncadg_ip_udp:`*`server`*`[`*`port`*`]`
By server name	`$ENCINA_CDS_ROOT/server/`*`serverName`* `ncadg_ip_udp:`*`server`*`[`*`port`*`]`
By object name	`$ENCINA_CDS_ROOT/object/`*`objectName`* `ncadg_ip_udp:`*`server`*`[`*`port`*`]`

In Table 3, *interfaceName* is the name of the Encina++ interface (binding by interface); *objectName* is the name of the exported object (binding by object); *serverName* is the name of the Encina++ server (binding by server); and *server* is the name of the corresponding Encina++ server that exports the interface (this is applicable to all three binding methods).

Consider the following example:

Two servers, **S1** and **S2**, are running on machines **siam** and **kramer** respectively. Server **S1** exports interface **I1** and named objects **011** and **012**. Server **S2** exports interface **I2** and the named object **021**. The binding files must contain entries similar to the following:

```
/.:/cdsRoot/server/S1          ncadg_ip_udp : siam [2021]
/.:/cdsRoot/interface/I1         ncadg_ip_udp : siam [2021]
/.:/cdsRoot/object/011           ncadg_ip_udp : siam [2021]
/.:/cdsRoot/object/012           ncadg_ip_udp : siam [2021]
/.:/cdsRoot/server/S2          ncadg_ip_udp : kramer [2042]
/.:/cdsRoot/interface/I2         ncadg_ip_udp : kramer [2042]
/.:/cdsRoot/object/021           ncadg_ip_udp : kramer [2042]
```

Chapter 5. Transaction processing overview

This chapter introduces the two transaction-demarcation interfaces for Encina++: Transactional-C++ (Tran-C++) and Encina's implementation of the Object Management Group's (OMG) Object Transaction Service (OTS). It also provides some guidelines for using the two interfaces in the same application.

The details of using the two transaction interfaces are discussed in the following chapters:

- Chapter 6, "Transaction processing with Transactional-C++," on page 41 provides details on using Transactional-C++ (Tran-C++).
- Chapter 7, "Transaction processing with OMG OTS for Encina++," on page 55 provides details on using the Object Management Group's (OMG) Object Transaction Service (OTS).

More general information on transaction processing can be found in the *Encina Transactional Programming Guide*.

Introduction to Transactional-C++

Tran-C++ provides extensions to the C++ programming language that simplify the creation and management of distributed transactions in Encina++ applications. These extensions include constructs, macros, and classes.

- Tran-C++ provides constructs that you can use to create transactions, suspend and resume transactions, and catch exceptions thrown during the execution of transactions.
- Tran-C++ defines macros that you can use to abort transactions, retrieve transaction identifiers in various contexts, and specify and return abort information associated with a transaction.
- Tran-C++ defines classes that encapsulate objects used to represent transactions, abort reasons for transactions, and callback functions that can be registered for transaction-event notification.

The scope of a transaction is defined by the Tran-C++ construct that creates that transaction. Each Tran-C++ transaction is associated automatically with a single thread that executes on its behalf. Function calls made within the scope of a transaction are executed on the thread associated with the transaction. A thread can execute on behalf of only one transaction at a time.

In addition, each transaction is associated with an exception scope. When a transaction is aborted by a participant, the Encina++ runtime throws an Encina++ exception. Tran-C++ applications can throw an application-specific exception to abort a transaction explicitly. Exceptions that are thrown within the scope of the transaction but not caught within the scope of that transaction cause the transaction to abort automatically. The Tran-C++ transaction constructs integrate support for catching exceptions thrown within the scope of a transaction.

Chapter 6, "Transaction processing with Transactional-C++," on page 41 describes how to use the Tran-C++ interface to create and manage transactions.

Introduction to the Encina Object Transaction Service

The Encina Object Transaction Service (OTS) implements the Object Management Group's (OMG) OTS interface for transaction demarcation. The OMG OTS interface is described in detail in the specification provided by the OMG. Encina's OTS implementation is based on the OMG specification.

Encina++ provides a version of the OTS interface that supports the use of transactions in the Distributed Computing Environment (DCE). Chapter 7, "Transaction processing with OMG OTS for Encina++," on page 55 covers this OTS interface for Encina++.

The Encina OTS interfaces consist of C++ classes and data types that enable the creation and management of distributed transactions in Encina++ applications. The interfaces allow you to choose between two transaction-demarcation models and support the use of C++ exceptions.

Transaction-demarcation models

The OMG OTS interface supports two models for transaction demarcation. In the *implicit* model, each transaction is associated automatically with the current thread via a transaction context. A thread's *transaction context* specifies the transaction on behalf of which the thread is executing. Because the transaction context is associated with the calling thread, the transaction context that defines a transaction is passed implicitly to objects. In the *explicit* model, the programmer must specify that an object is part of a transaction by explicitly passing the transaction context as a parameter to a member function invoked on the object.

The implicit model provides a simpler interface for coding transactional applications than the explicit model. Because most applications use the implicit model, the primary focus of this document is on using the implicit model of transaction demarcation in Encina++ applications. Only a brief overview of the explicit model is provided in this document. For information on writing multithreaded applications using either model, see Chapter 3, "Developing distributed applications," on page 9.

Exceptions

OTS uses C++ exceptions to handle error conditions. Typically, a transaction is enclosed in a `try` block to handle exceptions thrown within the scope of the transaction; the transaction is begun at the start of the `try` block and committed at the end of the `try` block. Rollbacks and other error handling can be done in the `catch` blocks for exceptions thrown within the scope of the transaction. Exceptions defined within an Encina TIDL interface automatically abort an associated transaction.

Chapter 6. Transaction processing with Transactional-C++

This chapter introduces transaction processing with Tran-C++ and provides information on using the Tran-C++ classes, constructs, and macros in Encina++ client and server applications. Refer to Chapter 5, "Transaction processing overview," on page 39 for more general information on transaction processing in Encina++.

For information on interactions between Tran-C++ and the Object Transaction Service (OTS), see "Interactions between Tran-C++ and OTS interfaces" on page 61.

Creating transactions

You can use the **transaction** construct to begin a transaction, specify which operations are part of the transaction, and specify actions to take when the transaction completes. A **transaction** construct is made up of a **transaction** clause, an optional **onCommit** clause, optional **onException** clauses, and an **onAnyException** clause. (An **onAbort** clause can be used in place of the **onAnyException** clause.) Any number of **onException** clauses can be specified.

The **transaction** clause defines the scope of a transaction; all operations executed within that scope are part of the same transaction. The transaction ends when the closing brace of the **transaction** clause is reached or when the transaction is aborted.

Tran-C++ attempts to commit a transaction if all the statements in the **transaction** clause are executed to completion without aborting the transaction. A transaction is committed only after all the participants in the transaction agree to commit. If the transaction commits, any statements in the **onCommit** clause are executed. If the transaction aborts and there is an **onException** clause for the exception that caused the transaction to abort, the statements in that clause are executed. Otherwise, the statements in the **onAnyException** clause are executed. The example in Figure 32 illustrates the use of all clauses.

```
transaction{
    // operations that are part of the transaction
    account1.debit(amount);
}
onCommit{
    // operations executed after the transaction commits
    cout << "The transaction committed." << endl;
}
onException(account::invalid_amount, except){
    cout << "The amount was invalid."
    cout << "The exception was: " << exception << endl;
    cout << "Transaction aborted." << endl;
}
onAnyException{
    // operations executed only after the transaction aborts
    cout << "The transaction aborted." << endl;
}
```

Figure 32. Creating a transaction by using the transaction construct

In this example, the only operation that is part of the transaction is the **debit** function. If the **debit** function is executed without aborting the transaction and the transaction commits successfully, the statement in the **onCommit** clause is executed, printing a message to the standard output stream. If the **debit** function (or another function called by **debit**) aborts the transaction and throws the **account::invalid_amount** exception, the statements in the **onException** clause are executed, sending a different message to the standard output stream. If the transaction is aborted with any other exception, the statement in the **onAnyException** clause is executed.

Each Tran-C++ transaction is associated with a single thread that runss on its behalf. A thread can run on behalf of only one transaction at a time. The Encina C++ interface defines the **TranPthread** class, which you can use to manage transactional threads and create concurrent transactions. See Chapter 3, "Developing distributed applications," on page 9 for more information.

Note: It is illegal to transfer control (for example, using a **return** or **goto** statement) out of any of the transaction clauses.

Nesting transactions

Tran-C++ allows you to create nested transactions. A *nested transaction* is a transaction begun within the scope of another transaction. Nested transactions can be used to isolate the effects of failures from the enclosing transaction. Two types of nested transactions are supported: subtransactions and nested top-level transactions.

A *subtransaction* is executed within the scope of its parent transaction. A subtransaction commits with respect to its parent transaction. That is, if the subtransaction commits but the parent transaction aborts, the effects of the subtransaction are rolled back. The parent transaction, however, does not rely on the outcome of its subtransactions and can still commit even if the subtransactions abort. Note that within the **onException**, **onAnyException**, and **onCommit** clauses of the subtransaction, the subtransaction is complete, and thus the parent of the subtransaction is the current transaction. A subtransaction is created by nesting a **transaction** construct within the **transaction** clause of another transaction, as shown in Figure 33 on page 43.

```
transaction{
    // operations that are part of the parent transaction
    transaction{
        // operations that are part of the subtransaction
        account1.debit(amount);
    }
   onCommit{
       // operations executed after the subtransaction commits
       cout << "The subtransaction committed." << endl;
   }
    onException(account::invalid_amount, except){
        // operations executed after the subtransaction
        // aborts as a result of the specified exception
        cout << "The amount was invalid."
        cout << "The subtransaction aborted." << endl;
    }
    onAnyException{
        // operations executed after the subtransaction aborts
        cout << "The subtransaction aborted." << endl;
    }
    account1.QueryBalance();
}
onCommit{
    // operations executed after the parent transaction commits
    cout << "The transaction committed." << endl;
}
onAnyException{
    // operations executed after the parent transaction aborts
    cout << "The transaction aborted." << endl;
}
```

Figure 33. Creating a subtransaction

A *nested top-level transaction* is executed outside the scope of its parent transaction. A nested top-level transaction aborts or commits independently of its parent transaction. That is, if the nested top-level transaction commits but the parent transaction aborts, the nested top-level transaction is not rolled back. As with subtransactions, the parent transaction does not rely on the outcome of the nested top-level transaction. A nested top-level transaction is created by nesting a **topLevel** construct within the **transaction** clause of another transaction, as shown in Figure 34 on page 44.

```
transaction{
    // operations that are part of the parent transaction
    ...
    topLevel{
        // operations that are part of the topLevel transaction
        account1.debit(amount);
    }
    onException(account::invalid_amount, except){
        // operations that are executed after the topLevel transaction aborts
        // as a result of the specified exception
        cout << "The amount was invalid."
        cout << "The subtransaction aborted." << endl;
     }
    onAnyException{
        // operations that are executed after the topLevel transaction aborts
        cout << "The subtransaction aborted." << endl;
    }
    account1.QueryBalance();
}
onCommit{
    // operations that are executed only after the parent transaction commits
    cout << "The transaction committed." << endl;
}
onAnyException{
    // operations that are executed only after the parent transaction aborts
    cout << "The transaction aborted." << endl;
}
```

Figure 34. Creating a nested top-level transaction

Suspending and resuming transactions

Tran-C++ allows you to suspend and resume transactions. For example, when you want to suspend a transaction to do some nontransactional work. As another example, your server application can create a single transaction to handle requests from a nontransactional application; the application can suspend and resume the transaction to handle subsequent requests rather than create new transactions for each request. Note that the resources that a transaction is accessing remain locked while the transaction is suspended.

You can suspend a transaction by using a **suspend** clause instead of an **onCommit** clause of a **transaction** or **topLevel** construct. If the transaction is not aborted, it is suspended when the end of the transaction is reached. The **suspend** clause has one parameter; an identifier for the transaction is returned in this parameter when the transaction is suspended. The returned value can be used to resume the transaction. Figure 35 on page 45 shows an example of a **suspend** clause used in a **transaction** construct.

```
void *id;
transaction{
    // operations that are part of the transaction
    account1.debit(amount);
}
suspend(id){
    // operations that are executed after the transaction is suspended
    cout << "The transaction is suspended." << endl;
}
onAnyException{
    // operations that are executed only after the transaction aborts
    cout << "The transaction aborted." << endl;
}
```

Figure 35. Suspending a transaction

You can resume a suspended transaction by using the **resumeTran** construct. Like any other transaction, a resumed transaction can be committed, suspended, or aborted. A pointer identifying the transaction to be resumed must be passed as an argument to the **resumeTran** clause. The argument you pass *must* be the value returned by the **suspend** clause that suspended the transaction you want to resume. Figure 36 shows an example of a **resumeTran** construct that resumes the transaction suspended in Figure 35.

```
resumeTran(id){
    // operations that are part of the resumed transaction
    account1.debit(amount);
}
onCommit{
    // operations that are executed only after the resumed transaction
    // commits
    cout << "The transaction committed." << endl;
}
onAnyException{
    // operations that are executed after the resumed transaction aborts
    cout << "The transaction aborted." << endl;
}
```

Figure 36. Resuming a suspended transaction

Getting the identity of a transaction

Some Tran-C++ macros and class functions require you to specify a transaction identifier as an argument. For example, you specify an identifier to abort a transaction other than the current transaction or to set a timeout for a transaction. Tran-C++ defines macros that allow you to get the identity of transactions in various scopes. Because transactions are managed as instances of the **Tran** class, the identity of a transaction is returned as a reference to a **Tran** object. Using Tran-C++, you can retrieve the identity of the current transaction, a parent transaction, or a completed transaction:

- To retrieve the identity of the current transaction, you can call the **getTran** macro from anywhere within the scope of the current transaction. Calling the **getTran** macro in the **onCommit**, **onException**, or **onAnyException** clause of a *nested* transaction returns the identity of the parent transaction because the **onCommit**, **onException**, and **onAnyException** clauses of a nested transaction are executed within the scope of the parent.

- To retrieve the identity of the parent of the current transaction (that is, the transaction that contains the current transaction), you can call the **getContainingTran** macro from anywhere within the scope of a nested transaction.
- To retrieve the identity of a completed transaction, you can call the **getCompletedTran** macro from the **onCommit**, **onException**, or **onAnyException** clause of the completed transaction.

The example in Figure 37 illustrates valid uses of the macros defined to retrieve the identity of a transaction.

```
transaction{
    // operations that are executed within the scope of the top-level
    // transaction
    cout << "Current transaction ID: " << getTran() << endl;
    transaction{
        // operations that are executed within the scope of the subtransaction
        cout << "Subtransaction ID: " << getTran() << endl;
        cout << "Parent ID: " << getContainingTran() << endl;
    }
    onAnyException{
        // nested onAnyException is executed within the scope of the parent
        // transaction
        cout << "Parent (now current) ID: " << getTran() << endl;
        cout << "Aborted transaction ID: " << getCompletedTran()
             << endl;
    }
}
onCommit{
    // top-level onCommit executes outside the scope of any
    // transaction
    cout << "Committed transaction ID: " << getCompletedTran()
         << endl;
}
onAnyException{
    // top-level onAnyException executes outside the scope of any
    // transaction
    cout << "Aborted transaction ID: " << getCompletedTran()
         << endl;
}
```

Figure 37. Getting the identity of a transaction in different contexts

"Aborting transactions" on page 47 describes the Tran-C++ macros that allow you to get abort information for transactions that have been aborted.

Checking transaction status

The Tran-C++ runtime system checks the status of a transaction every time an associated thread executes a Tran-C++ statement. Because the status of a transaction is not checked until execution of the next Tran-C++ statement, the transfer of control to the **onException** or **onAnyException** clause can be delayed if the transaction was not aborted by a Tran-C++ statement. To avoid potentially long delays in the transfer of control in applications that infrequently use Tran-C++ statements, you can call the **abortCheck** macro. Calling the **abortCheck** macro forces the runtime to check the status of the current transaction and transfer control to the **onAbort** clause if the transaction has been aborted.

The **abortCheck** macro takes no arguments and returns no value. Its only purpose is to provide a means for the Tran-C++ runtime system to signal a long-running application that a transaction has aborted. The macro has no effect if the current transaction has not been aborted.

Note: Exceptions thrown by the **abortCheck** macro should not be caught by the application.

Aborting transactions

A transaction can be aborted by the Tran-C++ runtime system, by any participant in the transaction, or by a Tran-C++ application. Communications or data access failures are the most common causes of runtime-system aborts. Your application can abort transactions explicitly by using the Tran-C++ **abortTran** macro or by throwing an exception that is not caught within the scope of the transaction.

Each Tran-C++ transaction is associated automatically with an exception scope. When a transaction is aborted, the **OtsExceptions::TranAborted** exception is thrown, causing an **onException** or **onAnyException** clause associated with the aborted transaction. When an abort is detected by a remote application that is performing work on behalf of a transaction, the remote application immediately stops performing work on behalf of the aborted transaction, the remote procedure call (RPC) returns to the calling application, and control is transferred to the **onException** or **onAnyException** clause of the transaction.

The **abortTran** macro must be called in a process operating within the scope of a transaction. Tran-C++ provides several definitions for the **abortTran** macro—each with default values for optional parameters—to enable you to abort transactions appropriately for your application. All definitions of the **abortTran** macro take an optional argument in which you can specify a particular transaction to abort. If another transaction is not specified, the **abortTran** macro aborts the current transaction.

Tran-C++ uses *abort reasons* to assign and retrieve information about why a transaction aborted. Any participant in the transaction can determine the abort reason for the aborted transaction. All definitions of the **abortTran** macro require that an abort reason be specified as the first argument. You can specify the reason for an abort in one of several ways: as an abort code, an abort string, an **AbortReason** object, or an abort reason data structure. The following sections describe the different ways to define abort reasons and how to use them when aborting transactions.

Using abort codes

An *abort code* is a signed integer constant that you define to indicate the reason why a transaction aborted. Using abort codes to specify abort reasons enables you to compare abort reasons and take actions based on the codes. If you use abort codes to identify abort reasons in your Tran-C++ application, define a different abort code for each condition under which the application aborts transactions. For example, an application can define abort codes to identify two conditions as shown in Figure 38 on page 48.

```
typedef enum {
    CANCELED_BY_USER,
    ACCESS_DENIED
} local_abort_t;
```

Figure 38. Example definition for abort codes

You can then specify one of the abort codes for the abort reason of an aborted transaction by passing the code as the first argument to the **abortTran** macro. For example, the statement shown in Figure 39 can be used to abort the current transaction and specify an abort code as its abort reason.

```
abortTran(CANCELED_BY_USER);
```

Figure 39. Example of aborting with an abort code

The abort reason for the aborted transaction contains an abort code of 0 (zero). The **abortCode** macro can be used to return the value of the abort code.

Although abort codes provide a way to compare and test abort reasons, they do not provide useful printable output. Calling the **abortReason** macro in an **onException** or **onAnyException** clause of a transaction aborted with an abort code simply returns a generic string generated by Tran-C++. Tran-C++, however, does allow you to associate an abort code with a function you define to translate codes to printable strings. The **abortTran** macro accepts an optional second argument if an abort code is specified for the abort reason. The value of the second argument must be a **Uuid** object that identifies a formatting function registered for the abort reason. (If a value is not specified for the second argument, Tran-C++ provides a default **Uuid** object that identifies a formatting function that is used internally to produce generic output.) If you use abort codes and want to produce printable strings for your abort reasons, you must provide a means for translating codes to a format appropriate for your application. See “Formatting abort reasons” on page 49 for a description of how to define and register a formatting function for your application.

Using abort strings

An *abort string* is a character string that you can use to specify the abort reason for a transaction. Though it is simpler to define abort reasons with strings than it is with codes, the usefulness of abort strings is limited. Because abort strings are variable-length strings that were possibly generated in a different National Language Support (NLS) locale, they cannot be compared as easily as abort codes. Also, programs that use abort strings are more difficult to internationalize than those that use codes.

You can specify a string as the abort reason by passing it as the first argument to the **abortTran** macro. For example, the statement in Figure 40 can be used to abort the current transaction and specify a string as its abort reason.

```
abortTran("User canceled the order.");
```

Figure 40. Example of aborting with a string

The abort reason for the aborted transaction contains the string “User cancelled the order.” and has an abort code of 0 (zero) by default. Calling the **abortReason** function in the **onException** or **onAnyException** clause of the aborted transaction returns the abort string as a null-terminated character string.

Formatting abort reasons

If you use abort codes to specify abort reasons for transactions, you can set up your application to convert abort codes to strings (such as NLS-compliant strings) for printing. Tran-C++ supplies a mechanism and defines conventions that enable you to do this. You can associate abort codes with a function you define to translate codes; Tran-C++ invokes this function (referred to as a *formatting function*) automatically when you display the abort reason for a transaction aborted with one of the associated abort codes.

In addition to defining the abort codes, you must take the following steps to set up your application to use a formatting function:

1. Define an abort format identifier to uniquely identify a formatting function.
2. Define a formatting function that formats abort codes for the abort reasons associated with the format identifier.
3. Register the format identifier and its associated formatting function with the application.

First, an abort format identifier must be defined so that Tran-C++ can associate a formatting function with the abort reason generated by an aborted transaction. The format identifier is a universal unique identifier (UUID) that uniquely identifies the formatting function; the format identifier is referred to as the *format UUID*. An example definition of two abort codes and a format UUID is shown in Figure 41. (In applications developed for the Distributed Computing Environment (DCE), the value for the format UUID can be created with the DCE **uuidgen** utility.) In the example, constants are used for the abort codes, but enumerated types can be used also.

```
/* Abort Code and Format */
const int CANCELED_BY_USER = 1;
const int ACCESS_DENIED = 2;
const char ABORT_FORMAT[] = "0014ad20-e154-1d68-85b0-9e62092caa77";
```

Figure 41. Defining an abort code and format UUID

Next, a formatting function must be defined for your abort reasons. The purpose of the formatting function is to take the information in an abort reason and use it to generate output appropriate to the application. The formatting function is invoked automatically and passed two arguments: a pointer to an abort reason data structure for the aborted transaction and a pointer to a buffer. By default, the buffer has a maximum size of `ENCINA_MAX_STATUS_STRING_SIZE` bytes. (See the reference page for the type for the abort reason data structure, **encina_abortReason_t**.)

Figure 42 on page 50 shows an example of a simple formatting function. The example function checks the abort code set for the abort reason and, based on the value of the abort code, returns a string describing the reason for the abort in the *bufferP* parameter. This example generates a printable string that is not NLS compliant. To be NLS compliant, a routine must consult a language catalog.

```
static void AbortFormatter(encina_abortReason_t *abortReasonP,
                           char *bufferP)
{
    char *abortString;
    switch(abortReasonP->code) {
      case CANCELED_BY_USER:
        abortString = "User canceled the order.";
        break;
      case ACCESS_DENIED:
        abortString = "Access denied!";
        break;
      default:
        abortString = "Unknown abort code.";
    }
    strcpy(bufferP, abortString);
}
```

Figure 42. Example function for formatting an abort reason

Finally, the formatting function must be registered with the application. Only one formatting function can be registered with the application at a time. The **registerAbortFormatter** macro associates the formatting function with the format UUID and registers it with the application. This macro takes two arguments: a **Uuid** object and the name of a function. The statement shown in Figure 43 registers the **AbortFormatter** function and associates the specified format UUID with it.

```
registerAbortFormatter(Uuid(ABORT_FORMAT), AbortFormatter);
```

Figure 43. Example of registering a formatting function

Once the function is registered, your application can call the **abortTran** macro to abort the transaction, using an abort code and a **Uuid** object for the format UUID associated with the formatting function. The statement in Figure 44 aborts the current transaction and specifies that the CANCELED_BY_USER abort code is formatted by using the **AbortFormatter** function.

```
abort(CANCELED_BY_USER, Uuid(ABORT_FORMAT));
```

Figure 44. Example of aborting with a formatted abort code

Calling the **abortReason** macro in the **onAbort** clause retrieves the abort reason string returned by the **AbortFormatter** function.

Using AbortReason objects

Tran-C++ uses the **AbortReason** class to manage abort reasons; the abort reason for an aborted transaction is represented as an instance of the **AbortReason** class. You can build on the functionality of the **AbortReason** class to control the way abort reasons are handled in your application. For example, you can derive your own abort reason class to specialize constructors.

An **AbortReason** object can be used to specify the abort reason for a transaction in the **abortTran** macro. Before you can use an **AbortReason** object as an abort reason, you must create an instance of the class. The class constructors require an argument that indicates the reason for the abort. Though you can specify the reason in a variety of ways, it is typically specified as an abort code or string. The remainder of this section describes one approach for using **AbortReason** objects in conjunction with abort codes. See "Using abort codes" on page 47 for more information on abort codes.

When abort codes are used to indicate the reason for an abort, you often need to be able to translate the codes into printable strings for use in error messages. If you derive your own abort reason class, you can define the routine that translates abort codes as a class member function rather than a function, thus limiting the scope of the routine to the class. You can also define constructors that automatically perform steps such as registering the formatting function. The example code in Figure 45 defines abort code and format UUID constants and a class derived from the **AbortReason** class. ("Formatting abort reasons" on page 49 provides general information on defining and registering formatting functions and format UUIDs for abort reasons.)

```
typedef enum {
    CANCELED_BY_USER,
    ACCESS_DENIED
} local_abort_t;
const char ABORT_FORMAT[] = "0014ad20-e154-1d68-85b0-9e62092caa77";
class LocalAbort : public AbortReason {
    private:
        static void LocalAbortFormatter(
                encina_abortReason_t *abortReasonP,
                char *bufferP)
        {
            char *abortString;
            AbortReason reason(abortReasonP);

            switch (reason.GetCode()) {
                case CANCELED_BY_USER:
                    abortString = "User canceled order.";
                    break;
                case ACCESS_DENIED:
                    abortString = "Access denied!";
                    break;
                default:
                    abortString = "Unknown abort code.";
            }
            strcpy(bufferP, abortString);
        }
    public:
        LocalAbort() {
            registerAbortFormatter(Uuid(ABORT_FORMAT),
                                   LocalAbortFormatter);
        }
        LocalAbort(long abortCode) :
            AbortReason(abortCode, Uuid(ABORT_FORMAT)) {}
};
```

Figure 45. Example class definition for specializing abort reasons

The **LocalAbort** class shown in Figure 45 defines one class member function and two constructors. The member function returns a printable string based on the abort code associated with the abort reason, which must be a **LocalAbort** object.

The application must register the formatting function before any transactions are aborted. The **LocalAbort** class defines a constructor that registers the member function with the Tran-C++ runtime and assigns a format UUID to uniquely identify the function. Using this constructor to create an instance of the **LocalAbort** class automatically registers the formatting function. The declaration shown in

Figure 46 creates an instance of the **LocalAbort** class named **aborter**, which invokes the constructor that registers the **LocalAbort::LocalAbortFormatter** function.

```
LocalAbort aborter;
```

Figure 46. Example of instantiating a specialized abort reason

Once the formatting function is registered, you can abort transactions by using an instance of the **LocalAbort** class as the abort reason. The other constructor defined for the **LocalAbort** class creates an abort reason by using an abort code and automatically sets the formatting function for the abort code. The statement shown in Figure 47 aborts the current transaction and specifies the reason for the abort as an instance of the **LocalAbort** class.

```
abortTran(LocalAbort(CANCELED_BY_USER));
```

Figure 47. Example of aborting with a specialized abort reason

This creates an abort reason that returns "User canceled order." as the abort reason string when the **abortReason** macro is called in the **onAnyException** clause for the transaction.

Using exceptions

You can abort a transaction by throwing an exception. Any exception that is not caught within the scope of the transaction aborts the transaction. For example, the statement in Figure 48 aborts the current transaction by throwing a user-defined exception named **insufficient_funds**.

```
throw insufficient_funds();
```

Figure 48. Example of aborting with an exception

The thrown exception is caught in the **onAnyException** or **onException** clause of the **transaction** construct.

See "Handling errors" on page 15 for more information on using exceptions.

Getting information about aborted transactions

When a transaction aborts, there are several Tran-C++ macros that can be called to retrieve information about the reason for the abort. These macros can be called *only* from within the **onException** or **onAnyException** clause of a transaction, and the information they return is valid only within the **onAbort** clause from which they are called.

Tran-C++ provides two macros that can be called to retrieve some indication of why the transaction aborted: the **abortReason** macro and the **abortCode** macro. Figure 49 on page 53 shows a simple example using these macros.

```
transaction{
    account1.debit(amount);
}
onCommit{
    cout << "The transaction committed." << endl;
}
onAnyException{
    // operations that are executed only after the transaction aborts
    cout << "The transaction aborted." << endl;
    cout << "    The reason is: " << abortReason() << endl;
    cout << "    The code is: " << abortCode() << endl;
}
```

Figure 49. Getting abort reason information

The **abortReason** macro returns a string that describes the reason why the current transaction aborted. If the transaction is aborted with a string used to specify the abort reason, the **abortReason** macro returns the string. If the transaction is aborted with an abort code used to specify the abort reason and a formatting function is registered for the code, the macro returns the string returned by the formatting function. Note that if the reason is defined as an abort code but no formatting function is registered for the code's format, the **abortReason** macro returns a string generated by Tran-C++. See "Using abort codes" on page 47 for more information on abort codes and formatting functions.

The **abortCode** macro returns an integer code that identifies the reason why the transaction aborted. You can use the **abortFormat** macro to ensure that the code returned by the **abortCode** macro is unique. The **abortFormat** macro returns a **Uuid** object that represents the format UUID for the abort reason. The format UUID is the identifier of the formatting function registered for the transaction. (See "Using abort codes" on page 47 for more information.) If your application performs actions based on the value of the abort code in an **onException** or **onAnyException** clause, first check the format UUID to ensure that the value corresponds to an abort code associated with the abort reason.

For example, your application can define two different sets of abort codes that have the same integer values; the abort codes in one set, however, are associated with a different format UUID than the abort codes in the other set. Your application must check the value returned by the **abortCode** macro *and* the value returned by the **abortFormat** macro to uniquely identify the abort reason.

The **getReason** macro returns the **AbortReason** object associated with an aborted transaction. You can use the **AbortReason** class member functions and operators to compare the **AbortReason** object against other abort reasons, get its abort code and format UUID, and so on. See the reference page for the **AbortReason** class for more information.

Chapter 7. Transaction processing with OMG OTS for Encina++

Encina++ provides an implementation of the Object Management Group's (OMG) Object Transaction Service (OTS) interface. The implementation supports the use of transactions in the Distributed Computing Environment (DCE). This chapter provides details about using the OTS interface in the DCE environment.

OTS provides the following models of transaction processing: implicit and explicit.

- Under the implicit model of transaction processing, the transaction is managed and propagated automatically, for the most part, by each transaction participant. This model is discussed in "Using the implicit model of transaction processing."
- Under the explicit model of transaction processing, the transaction is managed and propagated manually by each transaction participant. This model is discussed in "Using the explicit model of transaction processing" on page 59.

Using OTS in DCE

This section describes the DCE implementation of the OTS interface, which:

- Supports a subset of the OMG OTS specification; interfaces for defining recoverable resources are not included.
- Uses TIDL to define interface operations as transactional.
- Does not support the explicit creation or propagation of transactions, but does support the explicit management of implicitly created transactions.
- Uses Encina++ exceptions to report errors.

Examples and descriptions of OTS functionality that are used throughout this chapter are based on the DCE implementation.

Using the implicit model of transaction processing

The OTS interface defines the **Current** class to support the implicit model of transaction processing. The **Current** class defines class member functions to begin and end transactions, query the status of the current transaction, and set a timeout for a transaction. All member functions of the **Current** class are declared as static class functions; an instance of the class is not required to invoke a function.

In the implicit model, each thread being executed on behalf of a transaction has an associated transaction context. The transaction context specifies the transaction on behalf of which the thread is being executed. A thread can be executed on behalf of only one transaction at a time. Several threads can work on the same transaction; use the **TranPthread** class to create these threads.

For a function to propagate implicitly a transaction, that function must be marked as transactional within the Transactional Interface Definition Language (TIDL) file for an Encina++/DCE application.

Beginning and ending transactions

You can begin a transaction with the **Current::begin** function. Calling this function automatically associates the transaction context for the created transaction with the thread in which the function is executed. The transaction ends when one of the following occurs:

- The **Current::commit** function is invoked to commit the current transaction.
- The **Current::rollback** function is invoked to abort the current transaction.

The example in Figure 50 shows a typical use of the **Current** class member functions used to begin and end a transaction in an Encina++/DCE client.

```
int  SUCCESS = TRUE;
try {
    Current::begin();
    account1.debit(amount);
    account2.credit(amount);
    Current::commit();
}
catch(TransactionRolledBack) {
    cout << "A TransactionRolledBack exception was caught." << endl;
    SUCCESS = FALSE;
}
catch(...) {
    Current::rollback();
    cout << "An exception was caught." << endl;
    SUCCESS = FALSE;
}
if (SUCCESS)
    cout << "Transaction committed. " << endl;
else
    cout << "Transaction aborted. " << endl;
```

Figure 50. Beginning and ending an OTS transaction in Encina++/DCE

In Figure 50, the **debit** and **credit** functions are executed within the scope of the transaction. If the **debit** and **credit** functions are executed without aborting the transaction, the call to **Current::commit** ends the transaction and commit processing begins. The transaction is committed only if all of the participants in the transaction agree to commit; otherwise, the Encina++ runtime system throws an exception indicating that the transaction is rolled back (the **TransactionRolledBack** exception in Encina++/DCE). However, if the call to either the **debit** or **credit** function throws an exception (either explicitly or as a result of a communications failure, for example), the exception is caught at the client. In this case, the default **catch** statement catches the exception and calls the **Current::rollback** function to end the transaction.

Nesting transactions

OTS supports the use of subtransactions. A *subtransaction* is a nested transaction that is executed within the scope of its parent transaction and commits with respect to its parent. That is, if the subtransaction commits but the parent transaction aborts, the effects of the subtransaction are rolled back. The parent transaction, however, does not rely on the outcome of its subtransactions and can still commit even if the subtransactions abort. Figure 51 on page 57 shows an example that creates a subtransaction.

```
try {
    Current::begin();
    try {
        Current::begin();
        account1.debit(amount);
        Current::commit();
    }
    catch(...) {
        Current::rollback();
        cout << "An exception was caught." << endl;
        cout << "The subtransaction aborted." << endl;
    }
    account1.QueryBalance();
    Current::commit();
}
catch(...) {
    Current::rollback();
    cout << "An exception was caught." << endl;
    cout << "The transaction aborted." << endl;
}
```

Figure 51. Creating a nested OTS transaction

In Figure 51, the **debit** function is executed within the scope of a subtransaction, and the **QueryBalance** function is executed within the scope of the parent transaction. Even if the **debit** function is aborted (for example, due to insufficient funds), the **QueryBalance** function can still be executed without aborting the transaction, returning the correct account balance.

Aborting transactions

Transactions can be aborted by the runtime system or by any participant in a distributed transaction. Communications or data access failures are the most common cause of runtime-system aborts. Your application can abort transactions explicitly by calling the **Current::rollback** function. Typically, you define a remote procedure to throw an exception to transfer control out of the **try** block in which the transaction is begun, and the transaction is ended by a call to the **Current::rollback** function in the **catch** block. Figure 52 shows an example that explicitly aborts the transaction by throwing an exception.

```
void Account::debit(int amount)
{
    if ((balance - amount) < 0) {
        cout << "Insufficient funds." << endl;
        throw insufficient_funds{};
    }
    else {
        balance = balance - amount;
        cout << "New balance: " << balance << endl;
    }
}
```

Figure 52. Aborting an OTS transaction

The **debit** function aborts the transaction if the amount to be debited is greater than the account balance. The **insufficient_funds** exception is caught by the default **catch** statement (shown in Figure 50 on page 56), which calls the **Current::rollback** function to end the transaction.

Note: Make sure that your application ends each transaction once and only once by either committing or rolling back the transaction. This is particularly important if your application uses nested transactions. For example, if a manager function aborts a nested transaction instead of raising an exception, the current thread is disassociated from the nested transaction and associated with the parent transaction. In addition, execution of the statements in the `try` block enclosing the nested transaction continues until an exception is thrown. If no exception is thrown before the **Current::commit** function at the end of the `try` block is invoked, the function attempts to commit the *parent* transaction. To ensure that your application behaves as expected, you must manage the transaction context of the current thread carefully. (Refer to “Nesting transactions” on page 56 for more information on nested transactions.)

Suspending and resuming transactions

You can suspend a transaction by invoking the **Current::suspend** function in the context of the current transaction. The function returns a pointer of type **Control_ptr**, which is a pointer to a **Control** class instance. The **Control** instance represents the transaction context associated with the current thread. See “Using the explicit model of transaction processing” on page 59 for more information on the **Control** class. Note that the resources that a transaction is accessing remain locked while the transaction is suspended.

To resume the suspended transaction, call the **Current::resume** function, passing it the pointer returned when the transaction was suspended. Figure 53 shows an example of suspending and resuming a transaction.

```
Control_ptr controlP;
try {
    Current::begin();
    account1.debit(amount);
    controlP = Current::suspend();
    // do some nontransactional work
    ...
    Current::resume(controlP);
    Current::commit();
    }
catch(...) {
    Current::rollback();
    cout << "An exception was caught." << endl;
}
```

Figure 53. Suspending and resuming an OTS transaction

Checking transaction status

In a server, you can determine the status of the current transaction by invoking the **Current::get_status** function. The function returns a value of type **Status**, which is an enumerated type.

You can use the **Status** value to determine whether the current transaction is active, prepared, committed, marked for rollback, or already rolled back. The **Status** value can also indicate that the status is unknown or that there is no transaction. Refer to the reference page for the **Status** type for more information.

Using the explicit model of transaction processing

Under some circumstances, it is necessary to explicitly manage and propagate transactions. Explicit management (also known as direct management) and propagation of transactions can be difficult to program and is significantly less efficient than implicit management and propagation. Therefore, it is recommended that you use explicit transaction processing only when necessary.

Note: You cannot create or propagate transactions explicitly in Encina++/DCE applications using OTS; however, Encina++/DCE applications can explicitly manage transactions when they have been implicitly created.

The OTS interface defines several classes to encapsulate and manipulate transactions under the explicit model:

- The **Control** class contains member functions to get object references for the other two classes.
- The **Coordinator** class contains member functions for registering resources and gathering information on transactions.
- The **Terminator** class contains member functions for committing or rolling back a transaction.

The Control, Coordinator, and Terminator classes can be used to propagate a transaction between the participants. Using the Encina++/DCE framework, it is possible to commit or rollback a transaction. Only the transaction initiator can commit a transaction; any transaction participant can rollback a transaction. See "Committing or rolling back a transaction."

OTS also provides other functions for managing transactions explicitly. See "Other functions for explicitly managing transactions" on page 60 for a description of some of these functions.

Committing or rolling back a transaction

Only the transaction initiator can commit a transaction. Any transaction participant can roll back a transaction.

To commit or roll back a transaction, the transaction initiator must obtain a reference to a Terminator object. This can be done by calling the **Control::get_terminator** function. The transaction initiator calls the **Terminator::commit** function to commit the transaction, or the **Terminator::rollback** function to roll back the transaction

The **Terminator::commit** function has one parameter, *report_heuristics*, that specifies whether heuristic decisions are reported for the commit. This parameter can be set to either `TRUE` or `FALSE`. For more information on heuristics, see *Encina Toolkit Programming Guide*.

Figure 54 on page 60 shows an example of the code that is required to commit and roll back a transaction.

```
// we have a transaction
 Control_ptr txn;
//   we
 try { Current::begin();
  Account1.debit(amount);
  Account2.credit(amount);
  txn = Current::get_control();



//Call get_terminator to get a reference to the transaction's Terminator object
 Terminator_var txnTerm = txn->get_terminator();
...
//Perform transactional work
     ...
     //If transactional work is successfully completed, begin commit processing
     txnTerm->commit(FALSE);
     ...
     //If transactional work is not successfully completed, roll back
     txnTerm->rollback();
...
} // End Try block
```

Figure 54. Committing and rolling back a transaction in Encina++/DCE

Any transaction participant can roll back a transaction by calling the **Coordinator::rollback_only** function. The transaction is actually rolled back when the transaction initiator calls the **Terminator::commi**t function or the **Terminator::rollback** function.

To roll back a transaction, a transaction participant must obtain a reference to a **Coordinator** object. This can be done by calling the **Control::get_coordinator** function. The transaction participant can then call the **Coordinator::rollback_only** function.

Figure 55 shows an example of the code that is required to roll back a transaction.

```
//Obtain a transaction context
...
//Call get_coordinator to get a reference to the transaction's Coordinator object
 Terminator_var txnCoord = txn->get_coordinator();
...
//Perform transactional work
     ...
     //If transactional work is not successfully completed, mark the transaction for rollback
     txnCoord->rollback_only();
...
```

Figure 55. Rolling back a transaction in Encina++/DCE

Other functions for explicitly managing transactions

The **Control** and **Coordinator** classes contain a variety of additional functions and operators for explicitly managing transactions, including the following:

- The **Cooordinator::create_subtransaction** function creates a subtransaction.
- The **Cooordinator::get_parent_status**, **Cooordinator::get_status**, and **Cooordinator::get_top_level_status** functions return transaction status information.

For information on these and other **Control** and **Coordinator** member functions, see the reference pages for these classes and functions.

Interactions between Tran-C++ and OTS interfaces

This section documents general guidelines and limitations regarding the use of Tran-C++ and the Encina OTS interfaces in the same application. Though both interfaces are built on top of the same lower-level Encina components and provide similar functionality, the implementations differ in significant ways.

In general, using both interfaces in the same application is not recommended. However, there are possible cases in which you need to use one interface for a client application and the other for a server application. If you do this, you must be aware of the limitations imposed by the differences between the two interfaces. The main differences concern the use of abort reasons and transaction identity.

The OTS interface does not support the use of Tran-C++ abort reasons. For example, if a server manager function uses the Tran-C++ **abortTran** macro to abort a transaction created at the client with the OTS **Current** class, the abort reason specified as an argument to the **abortTran** macro has no meaning to the OTS transaction at the client. The OTS interface does not provide a way to retrieve the abort reason, abort code, or format identifier used when a Tran-C++ transaction is aborted.

Likewise, if a server manager function calls the **Current::rollback** function to abort a transaction created at the client with a Tran-C++ **transaction** construct, an abort reason cannot be specified in the call that aborts the transaction. If an abort reason is not specified, Tran-C++ creates a generic abort reason that contains default values for the abort code, format UUID, and so on. You can, however, use the functionality defined in the Encina Abort Facility to set and retrieve abort reasons in an OTS transaction. The Encina Abort Facility is documented in the *Encina Toolkit Programming Guide*.

Another difference between the Tran-C++ and OTS interfaces is the way they represent transaction identity. Tran-C++ uses **Tran** objects, and the OTS interface uses **Control** objects as transaction identifiers. You cannot cast a **Tran** object to a **Control** object, nor can you cast a **Control** object to a **Tran** object. You can, however, cast both **Tran** and **Control** objects to the Encina **tran_tid_t** type for the purpose of comparing transaction identifiers.

Chapter 8. Using threads

Threads provide a way of improving the performance of an application by making it possible to execute multiple operations in parallel. Encina++ servers can create threads automatically, depending on which concurrency mode is set when the server starts listening (see "Listening for RPCs" on page 14). The Encina++ thread classes encapsulate Portable Operating System Interface (POSIX) threads, enabling you to create threads explicitly; you can create threads that are executed either outside the context of a transaction (nontransactional threads) or within the context of a transaction (transactional threads). The following sections describe how these two different types of threads can be used in Encina++ applications.

Using nontransactional threads

Encina++ defines the **Pthread** class for creating and controlling threads. A thread created by using the **Pthread** class is referred to as a *nontransactional thread* because the function executed by the thread does no work on behalf of a transaction, even if the thread is created within the scope of a transaction.

The example code in Figure 56 illustrates the use of the **Pthread** class. The first **for** loop calls the **Pthread::Create** function to execute the same function in five new threads. The second **for** loop calls the **Pthread::Join** function to join the five threads to the main thread.

```
// instantiate 5 threads
Pthread threads[5];
int i;

// create 5 threads that execute the threadWork function and then
// join the threads
for (i=0;i<5;i++)
    threads[i].Create(threadWork, (void *)i);
for (i=0;i<5;i++)
    threads[i].Join();
```

Figure 56. Creating and joining nontransactional threads

Each new thread executes a function specified as the first argument to the **Pthread::Create** function. The function executed by the threads created in Figure 56 is named **threadWork**. Figure 57 on page 64 shows a sample definition for the **threadWork** function.

```
void* threadWork(void *argP)
{
    int arg = (int)argP;
    void *result = (void *) -1;

    cout << "   Executing thread #" << (int)argP << endl;
    if (arg % 2)
        ThisPthread::Exit(result);
    else
        // do work here...

    cout << "\t Thread #" << arg << " completed." << endl;
    return(result);
}
```

Figure 57. Sample function executed on a nontransactional thread

The function executed by the thread can use the **ThisPthread** class to control the thread from within the thread itself. For example, the **threadWork** function in Figure 57 calls **ThisPthread::Exit** to terminate the execution of odd-numbered threads before the remaining statements are executed. You can also call the **ThisPthread::Delay** function to suspend the execution of the thread for a specified period of time or the **ThisPthread::Yield** function to yield the processor to another thread.

Using transactional threads

Encina++ defines the **TranPthread** class, which is derived from the **Pthread** class, for creating and controlling threads that are executed transactionally. When a *transactional thread* is created within the scope of a transaction, the thread inherits the transaction's environment; the operation being executed by the thread is executed on behalf of the transaction. When a transactional thread is created outside the scope of a transaction, the thread behaves like a nontransactional thread (unless you specify explicitly that the thread create a new transaction). Using the **TranPthread** class, you can create either of the following:

- Concurrent transactional threads
- Concurrent transactions

If *concurrent transactional threads* are created, each thread is executed on behalf of the same transaction (see "Creating concurrent transactional threads"). If *concurrent transactions* are created, each transaction is executed on its own thread (see "Creating concurrent transactions" on page 65).

Because the **TranPthread** class is defined as part of the Encina C++ interface, you can create transactional threads in applications that use Tran-C++, the OMG OTS interface, or neither. For applications that use the OMG OTS interface, only those applications that conform to the implicit model of transaction demarcation (using the **Current** class) can use the **TranPthread** class to create transactional threads. Applications that use the explicit model can use the **Pthread** class, passing the explicit (**Control**) object as an argument to the newly created thread to achieve the same effect. See Chapter 5, "Transaction processing overview," on page 39 for more information on the two transaction-demarcation models for the OMG OTS interface.

Creating concurrent transactional threads

Transactional threads are typically used to create concurrent threads that do transactional work. Figure 58 on page 65 shows a Tran-C++ example that uses the

TranPthread::Create function to create five threads; the threads execute the function named **tranThreadWork** concurrently.

```
TranPthread threads[5];

// creating threads within the scope of a transaction
transaction {
    for (i=0;i<5;i++)
        threads[i].Create(tranThreadWork, (void *)i);
    for (i=0;i<5;i++)
        threads[i].Join();
}
onAnyException {
    cout << "  Transaction " << getCompletedTran() << " aborted: "
         << abortReason() << endl;
}
```

Figure 58. Creating concurrent transactional threads

Because the transactional threads are created within the scope of the transaction construct, the **tranThreadWork** function is executed as part of the transaction. Figure 59 shows a simple example function that displays the transaction identifier and causes the odd-numbered threads (in this case, threads 1 and 3) to exit. As with nontransactional threads, you can use the **ThisPthread** class to control a thread from within the thread itself. The call to the **ThisPthread::Exit** function causes the **tranThreadWork** function to exit before the remaining statements are executed.

```
void* tranThreadWork(void *argP)
{
    int arg = (int)argP;
    void *result = (void *) -1;

    cout << "\t Transaction ID: " << getTran() << endl;
    if (arg % 2)
        ThisPthread::Exit(result);

    cout << "\t Thread #" << arg << " completed." << endl;
    return(result);
}
```

Figure 59. Sample function executed on a transactional thread

Creating concurrent transactions

You can also use transactional threads to create concurrent transactions or concurrent subtransactions. The **TranPthread::Create** function allows you to specify that the function executed on the thread be executed within the scope of a new transaction by passing TRUE as the third argument. If TRUE is *not* passed as the third argument, either concurrent transactional threads or concurrent nontransactional threads are created, depending on whether the **TranPthread::Create** function is called within the scope of a transaction.

For example, you can create concurrent, top-level transactions by passing TRUE as the third argument to the **TranPthread::Create** function as shown in Figure 60 on page 66. Each transaction completes when the thread on which it is executed completes.

```
TranPthread threads[5];
AbortReason *reason;

for (i=0;i<5;i++)
    threads[i].Create(tranThreadWork, (void *)i, TRUE);
for (i=0;i<5;i++) {
    threads[i].Join();
    if (reason = threads[i].GetReason())
        cout << "  Transaction aborted in thread #" << i << ": "
             << *reason << endl;
}
```

Figure 60. Creating concurrent transactions

Note the use of the **TranPthread::GetReason** function in the second **for** loop. You can use this function to get the abort reason for transactions that you do not create explicitly (for example, by using the Tran-C++ **transaction** construct). Figure 61 shows a simple example function that displays the transaction identifier and aborts the transactions created on odd-numbered threads (in this case, threads 1 and 3).

```
void* tranThreadWork(void *argP)
{
    int arg = (int)argP;
    void *result = (void *) -1;

    cout << "\t Transaction ID: " << getTran() << endl;
    if (arg % 2)
        abortTran("*** Odd-numbered thread ***");

    cout << "\t Thread # " << arg << " completed." << endl;
    return(result);
}
```

Figure 61. Sample function executed on a concurrent transactional thread

There are a variety of ways to create concurrent subtransactions in Encina++ applications. One way that you can create concurrent subtransactions is by calling the **TranPthread::Create** function (with the `TRUE` argument) within the scope of a **transaction** construct as shown in Figure 62.

```
transaction {
    for (i=0;i<5;i++)
        threads[i].Create(tranThreadWork, (void *)i, TRUE);
    for (i=0;i<5;i++) {
        threads[i].Join();
        if (reason = threads[i].GetReason())
            cout << "  Subtransaction of transaction " << getTran()
                 << " aborted in thread #" << i << ": " << *reason
                 << endl;
    }
}
onAnyException {
    cout << "  Transaction " << getCompletedTran() << " aborted: "
         << abortReason() << endl;
}
```

Figure 62. Creating concurrent subtransactions

You can also create subtransactions by explicitly creating transactions rather than by passing `TRUE` to the **TranPthread::Create** function. For example, you can use the Tran-C++ **transaction** construct in the function executed by the transactional thread. Refer to Chapter 5, “Transaction processing overview,” on page 39 for more information on using the Tran-C++ or OMG OTS interfaces.

Chapter 9. Diagnostics

This chapter describes the diagnostic support provided for Encina++. Encina++ uses the standard tracing mechanism used by all Encina components to generate output to aid in diagnosing problems. The following sections describe the programmatic controls for diagnostic output and provide lists of the error and warning messages defined for the Encina++ interfaces.

Tracing applications

The Encina Trace Facility enables you to follow the execution path of Encina applications, tracing various events such as the entry and exit of functions. It also enables you to specify the destination for trace output. This section documents trace information specific to Encina++. See the documentation for the Encina Trace Facility in the *Encina Toolkit Programming Guide* for detailed information about tracing Encina applications.

Tracing can be controlled through Encina's administrative facilities. Some administrative facilities require that you specify a name for the component to be traced. The component name defined for Encina++ is **ots** (object transaction service).

Tracing can be controlled programmatically as well. To enable or disable tracing, you can assign a value for the Encina++ trace mask in your application. This also sets the trace level for the Encina++ component. The trace mask for Encina++ is an exported global variable named *ots_traceMask*. The *ots_traceMask* variable is interpreted as a bit mask. It is defined as follows:

```
unsigned long ots_traceMask;
```

The Encina Trace Facility defines bit constants that can be used in specifying the value of a trace mask. Encina++ supports the following bit constants for the *ots_traceMask* variable:

- `TRACE_ENTRY` enables tracing of the entry and exit of Encina++ functions.
- `TRACE_EVENT` enables tracing of significant events in Encina++.
- `TRACE_PARAM` enables tracing of the parameters passed to Encina++ functions.
- `TRACE_GLOBAL` enables all tracing.
- `TRACE_NONE` disables all tracing.

The value specified for the trace mask variable takes effect immediately. By default, trace output is sent to an internal buffer unless you redirect the output to another destination.

Dumping the application state

The Encina C++ interface defines a state dump function named **ots_DumpState**, which generates output describing the internal state of an Encina++ application at the time the function is called. You can write your application to call the function directly or use debugger commands to call the function if you are running the application within a debugger. See the reference l page for the **ots_DumpState** function for additional information.

State dumps are managed by the Encina Trace Facility. By default, output from a state dump is sent to an internal buffer unless you redirect the output to another destination. See the *Encina Toolkit Programming Guide* for more information.

Error and warning messages

Encina++ error and warning messages are documented in *Encina Messages and Codes*.

Appendix. Compilation issues

This appendix describes the header files and library files required for compiling C++ applications. It also includes information on compiler and linker options and other compilation issues.

TIDL and IDL compilation

This section describes the tasks required to compile the interfaces needed for Encina++ client and server interaction.

Follow these steps when defining an interface for use in an Encina++/DCE application:

1. Run the **tidl** compiler. Use the **-ots** option and pass the transactional interface definition language (TIDL) file as the argument to the command.
2. Run the **idl** compiler. Use the **-no_mepv** and **-cepv** options and pass the interface definition language (IDL) file generated by the **tidl** command.

For more information on the compilation process and the files produced, see "Generating stub files" on page 19.

C++ header and library files

Encina++ provides header files and libraries for use with C++ applications. The header files and libraries that you use depend on the requirements of your application. You can develop applications only for an environment that uses the Distributed Computing Environment (DCE)

Header files

Encina++ header files

Encina++ applications must include the appropriate header files to define the data types, classes, functions, macros, and constructs that are used in Encina++ and the runtime environment.

In a DCE-only environment:

- Client and server applications must each include the file **ots/dce/encina_dce.H**. This header file automatically includes the Tran-C++ classes, macros, and constructs.
- Applications that use OTS must include the file **ots/omg/ots.H**.
- SFS++ applications must include the file **ots/pos/sfs++.H**.
- RQS++ applications must include the file **ots/pos/rqs++.H**.

C header files

C++ uses a technique called *name mangling* to make the names of overloaded member functions unique. To prevent name mangling of C function names, C language header files must be declared as `extern "C"` when included in an Encina++ application. The `extern "C"` declaration is required only for non-Encina C header files; the C header files that are provided with Encina use declarations that are compatible with Encina++.

Library files

Encina++ Libraries

Encina++ applications must link with the appropriate Encina library files.

Note: Servers that also act as clients to other servers must link with the libraries that are listed for servers. They must not link with the libraries that are listed for clients. Linking with both client and server libraries can result in unpredictable behavior.

In a DCE-only environment

- Client applications must link with the following libraries:
  - **EncPlusCli**
  - **EncMonCli**
  - **Encina**
- Server applications must link with the following libraries:
  - **EncPlusServ**
  - **EncMonServ**
  - **EncServer**
  - **EncClient**
  - **Encina**
- SFS++ server applications must also link with the **EncPlusSfs** and **EncSfs** libraries. For more information on SFS++, see *Encina RQS++ and SFS++ Programming Guide*.
- RQS++ server applications must also link with the **EncPlusRqs** and **EncRqs** libraries. For more information on RQS++, see *Encina RQS++ and SFS++ Programming Guide*.
- No additional libraries are required for the OTS or Tran-C++ interfaces.

DCE libraries

On most platforms, the DCE library (**dce**) must be explicitly linked with Encina++/DCE applications. The libraries that are needed for various platforms are shown in Table 4.

Table 4. DCE libraries by platform

Platform	**Encina++ / DCE**
Solaris 2.x	dce, m, dl
HP-UX	dcekt
AIX	dce
Windows	libdce, pthreads

Platform-specific libraries

Additional libraries must be linked with the application code. The additional libraries depend on the operating system and machine type. The libraries that are needed for various platforms are shown in Table 5.

Table 5. Platform-specific libraries

Platform	**System libraries**
Solaris 2.x	nsl, socket, thread
HP-UX	ndbm, M, dld, c_r

Table 5. Platform-specific libraries (continued)

Platform	System libraries
AIX	C_r, c

Note: Information on system libraries can change with time. For the most current information about system libraries for the platform that you are using, see the release notes for that platform. Also refer to the Makefile that is installed with the sample Encina applications.

Compiler and linker options

Table 6 and Table 7 list platform-specific compiler and linker options for UNIX platforms. Table 8 lists compiler and linker flags that are required on Windows platforms.

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

Table 6. Platform-specific compiler options for UNIX

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

Table 7. Platform-specific linker options for UNIX (HP-UX only)

Platform	Additional linker options
HP-UX	*+eh*

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

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

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

Other compilation issues

This section describes issues related to compilation for consideration when developing your Encina++ applications.

Renaming the abort macro

The Tran-C++ **abort** macro (supported for backwards compatibility only) cannot be used in an application that also uses the C library **abort** function. You can avoid a name collision between these two functions by using the **abortTran** macro instead of **abort**.

Checking for runtime errors

By default, the Encina++ runtime does not check whether an application uses Tran-C++ constructs and macros correctly; checking for runtime errors of this type degrades the performance of the application. If a construct or macro is used incorrectly (for example, the **abortTran** macro is called outside the scope of a transaction), the behavior is undefined, and execution errors typically occur.

Runtime checking, however, is useful during the development of an application. You can enable Encina++ to perform runtime checking by defining the flag `OTS_CHECK_MACRO_USAGE` when compiling your application. Defining this flag causes the runtime to generate a fatal error when an incorrectly used construct or macro is encountered.

Notices

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

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

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

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

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

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

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

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

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

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

Trademarks and service marks

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

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

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

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

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

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

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

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

This software contains RSA encryption code.

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

Index

Printed in USA

SC09-4478-05

Spine information:

IBM TXSeries™ Encina Object-Oriented Programming Guide Version 5.1 SC09-4478-05