TXSeries™ for Multiplatforms

Encina® Toolkit Programming Guide

Version 5.0

SC09-4484-01

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 283.

Second Edition (March 2001)

This edition replaces SC09-4484-00.

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

Contents

Figures

Tables

About this book

This document describes concepts, data types, and system calls for the Encina® Toolkit. The Encina Toolkit provides low-level services required for distributed transaction processing systems. The Encina Toolkit comprises several modules, which are grouped into two major components: the Toolkit Executive, which provides services that permit a process to initiate, participate in, and commit distributed transactions; and the Toolkit Server Core, which provides facilities for managing recoverable data (data that is accessed and updated transactionally).

Who should read this book

This document is oriented towards developers who may be new to transactional programming, new to distributed programming in general, or new to using the Encina Toolkit to develop these sorts of applications.

It is assumed that users of the Toolkit are familiar with program development in the C programming language and the Open Software Foundation (OSF®) Distributed Computing Environment (DCE).

Document organization

This document has the following organization:

Part 1: *Encina Toolkit*

- "Chapter 1. Introduction to the Encina Toolkit" on page 3 describes the Toolkit architecture and modules and provides an overview of Toolkit applications.

Part 2: *Encina Toolkit Executive*

- "Chapter 2. Transaction Service (TRAN)" on page 11 provides a guide to the functions and data types in the TRAN interface.
- "Chapter 3. Thread-to-Tid Mapping Service (ThreadTid)" on page 61 provides a guide to the functions and data types in the ThreadTid interface.
- "Chapter 4. Transactional RPC Service (TRPC)" on page 67 provides a guide to the functions and data types in the TRPC interface.
- "Chapter 5. Abort Facility" on page 91 provides a guide to the functions and data types in the interface.
- "Chapter 6. IBM Transarc/Encina DCE Utilities (TRDCE)" on page 95 provides a guide to the functions and data types in the TRDCE interface.

Part 3: *Encina Toolkit Server Core*

- "Chapter 7. Lock Service (LOCK)" on page 105 provides a guide to the functions and data types in the LOCK interface.
- "Chapter 8. Log Service (LOG)" on page 119 provides a guide to the functions and data types in the LOG interface.
- "Chapter 10. Recovery Service (REC)" on page 143 provides a guide to the functions and data types in the REC interface.
- "Chapter 11. Restart Service" on page 167 provides a guide to the functions and data types in the Restart interface.
- "Chapter 12. The Transaction Manager-XA (TM-XA) Service" on page 169 provides a guide to the functions and data types in the TM-XA interface.
- "Chapter 13. Volume Service (VOL)" on page 207 provides a guide to the functions and data types in the VOL interface.

Appendixes

- "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 documents the Encina Trace Facility.
- "Appendix B. Error messages from other components" on page 247 provides information about handling error messages from low-level Encina components and DCE.
- "Appendix C. Administrative RPC interfaces for the Encina Toolkit" on page 251 provides general information about RPC interfaces to administrative functions and contains the IDL files defining the administrative RPC interfaces provided with the Encina Toolkit Executive.
- "Appendix D. Using the Toolkit to write a recoverable server" on page 265 discusses how to develop recoverable servers using components of the Encina Toolkit and Transactional-C.

Related information

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

- *Writing Encina Applications*
- *Encina Transactional Programming Guide*
- *OSF DCE Application Development Reference* and *OSF DCE Application Development Guide* (for information about developing applications that use DCE RPC)

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.)
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.)

Table 1. Conventions used in this book (continued)

Convention	Meaning
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\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 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 WebSphere Application Server Enterprise Edition documentation, send your comments by e-mail to wasdoc@us.ibm.com. Be sure to include the name of the book, the document number of the book, the version of WebSphere Application Server Enterprise Edition, and, if applicable, the specific location of the information you are commenting on (for example, a page number or table number).

Part 1. Encina Toolkit

Chapter 1. Introduction to the Encina Toolkit

Encina Toolkit overview

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

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

The remainder of this chapter describes the design of the Encina Toolkit and introduces the functionality provided by the individual services. It also presents an overview of how the modules of the Toolkit are combined to create client and server applications.

The Encina Toolkit architecture

Encina is based on a modular, layered architecture that builds on services supplied by the Distributed Computing Environment (DCE). The lower layers of Encina extend DCE to form a set of core technologies that enable client/server transaction processing and the management of recoverable data.

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

- *Toolkit Executive*. The Executive provides services that permit a process to initiate, participate in, and commit distributed transactions. These services include transactional extensions to DCE remote procedure calls (RPCs) that ensure transactional integrity over distributed computations transparently. The Executive also supports nested transactions, a feature that provides failure containment and simplifies application development.
- *Toolkit Server Core*. Built upon the Executive, the Server Core provides facilities for managing recoverable data—data that is accessed and updated transactionally. These facilities include a locking library to serialize data

access, a recoverable storage system to allow transactions to roll back or roll forward after failures, and an X/Open XA interface to permit the use of XA-compliant resource managers.

Figure 1 illustrates the high-level architecture of Encina and the two major components of the Toolkit.

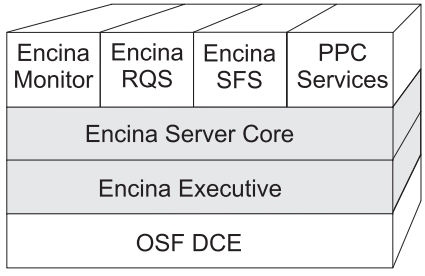

Figure 1. Architecture of the Encina system

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

The Encina Toolkit modules

Each module of the Toolkit provides a distinct service important to the functionality of a distributed transaction processing system. The Toolkit modules guarantee the basic functions of a transaction processing system.

The following services, which are part of the Encina Executive, provide support for writing client applications:

- The Distributed Transaction Service (TRAN) coordinates multiple transactions, guarantees that transactions either commit or abort consistently, and manages the delivery of information about transaction outcome to various participants.
- The Transactional Remote Procedure Call Service (TRPC) provides the underlying communications mechanisms used by the entire system, enabling transactions to be distributed to other programs and nodes.
- The Thread-to-Tid Mapping Service (ThreadTid) maintains the association between a thread and a transaction identifier (TID), allowing applications to determine which transaction is associated with a thread.

The following services, which are part of the Encina Server Core, provide support for writing server applications:

- The Lock Service (LOCK) permits synchronization of accesses to data. Transactions that run concurrently act as though each ran to completion before the next was begun.
- The Log Service (LOG) and Recovery Service (REC) help guarantee that changes made to data by a transaction are permanent if the transaction commits, regardless of system restarts or media failures, and that changes made to data by that transaction are undone if the transaction aborts.
- The Volume Service (VOL) provides a logical interface to underlying physical storage that enables volumes and files to span physical device boundaries.
- The Transaction Manager-XA Service (TM-XA) implements the transaction manager side of the X/Open XA interface for coordinating distributed transactions with XA-compliant resource managers.

The modules of the Toolkit can be invoked automatically through higher-level C-language interfaces provided by Encina. Transactional-C (Tran-C) is the suggested interface for the development of transactional applications based on the Encina Toolkit. Tran-C adds functions and constructs to the C programming language that simplify the creation of transactions and correctly handling whether those transactions abort or commit successfully. Tran-C, Encina's TX interface, and the Transactional Interface Definition Language (TIDL) are documented in the *Encina Transactional Programming Guide*.

For developing transactional applications based on the Encina Monitor, the Tran-C and Monitor interfaces are recommended. The Encina Monitor is documented in the *Encina Monitor Programming Guide*.

Toolkit applications

The modular construction of the Toolkit allows it to be used to develop different types of transactional applications. For this reason, it is possible to build a distributed transactional application using many different combinations of Toolkit components. The combination of components that you choose to use when developing an application is determined by the nature of the application and your preferences. Deciding which components to use also depends on whether the application integrates any existing, external applications or modules that perform some of the functions of a module of the Toolkit.

This section provides information about the organization of applications, discussing the interaction between the various modules of the Toolkit. This information is intended to help application developers select the modules of the Toolkit that are appropriate for their applications.

Toolkit clients

The modules that make up the Toolkit Executive provide the functionality necessary to write client applications. Client applications start transactions and make transactional RPCs to Encina server applications. The primary low-level modules of the Executive include TRAN, TRPC, and ThreadTid.

The easiest way to build a Toolkit client application is to use Tran-C. The use of low-level Toolkit modules is not usually necessary, as Tran-C provides most of the functionality offered by the low-level components of the Executive. There is little performance penalty for using Tran-C in preference to using the low-level modules directly.

In some cases, however, aspects of the high-level interface provided by Tran-C can conflict with the requirements of an application. For example, Tran-C's exception handling facility can cause a conflict if another language that has its own exception handling facility (such as C++) is used in the same application. Under such circumstances, you might decide to use the low-level modules directly instead of using Tran-C. Using the lower-level interfaces (such as TRAN) to begin and end transactions provides increased flexibility but can result in programs that are longer, less readable, and more difficult to maintain.

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

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

Toolkit servers

The modules that make up the Encina Server Core—in combination with the modules of the Executive—provide the functionality necessary to write recoverable server applications that manage persistent data and accept transactional RPCs to access that data. The primary low-level modules of the Server Core used to write server applications include REC, LOG, VOL, and LOCK. The Server Core also includes TM-XA, which is used to access and use XA-compliant resource managers in your transactions.

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

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

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

Part 2. Encina Toolkit Executive

Chapter 2. Transaction Service (TRAN)

Transaction Service overview

The Encina Distributed Transaction Service (TRAN) is a tool for building reliable distributed applications. The Transaction Service chapter is organized as follows:

- The remainder of this section explains the basic concepts behind the TRAN interface, such as the transaction model and other important abstractions. It describes how applications are normally structured and how they communicate. Read this overview before using TRAN calls.
- "Data types" on page 26 describes the TRAN data types that correspond to principal abstractions. It also describes the functions used to manipulate each data type.
- "Initialization and termination" on page 32 describes the initialization and termination functions.
- "Application interface" on page 34 describes the application interface, which begins and ends transactions, as well as providing more advanced functions for associating data with transactions, selecting the transaction coordinator, and advanced preparing functions.
- "Communications service interface" on page 41 describes the communications service interface.
- "Recovery Service interface" on page 46 describes the recovery service interface.
- "Heuristic outcomes" on page 53 describes heuristic outcomes.
- "Application environment specification" on page 53 describes the application environment TRAN requires. TRAN registers functions for the services such as synchronization, scheduling, memory management, time, and termination.
- "Properties" on page 56 provides a table listing the properties defined by TRAN.
- "Diagnostics" on page 57 describes the diagnostic facilities provided by TRAN. These diagnostic facilities include error messages, abort reasons, and the Encina Trace Facility.

Transaction model

The Transaction Service (TRAN) assists application programs in the definition, execution, and commitment of distributed transactions. A *transaction* is a set of operations that must be executed together to perform a consistent transformation of data. This set of operations may be distributed, meaning that the operations take place in more than one independent application

program. TRAN provides functions for grouping operations into transactions. TRAN supervises other modules to ensure that each transaction is executed atomically, independently of other transactions, and that once completed, its effects are permanent. The act of successfully completing a transaction is called *commitment* and is the primary function of TRAN.

For example, a banking system may distribute its account data over several databases, possibly on different machines. A transaction that transfers funds from one account to another would consist of two individual updates: one to debit the first account, and one to credit the second account. Both operations must complete and be successful to transfer funds accurately.

Transactions are defined to have the following properties:

- *Failure atomicity*: A transaction is either successful or unsuccessful. Either all of the operations that make up a transaction take effect, or none take effect. A successful transaction is said to *commit*. An unsuccessful transaction is said to *abort*. Any operations performed by an aborted transaction are undone so its effects are not visible.
- *Consistency*: A transaction transforms the distributed data from one consistent state to another. The application program is responsible for ensuring consistency.
- *Isolation*: Each transaction appears to execute independently of other transactions that may be running concurrently. The effects of a transaction are not visible to others until the transaction completes (commits or aborts). The transactions appear to be *serialized*, with two or more transactions acting as though one completed before the other was begun, even though they ran concurrently.
- *Durability*: Also known as *permanence*, this ensures that once completed, the effects of a transaction are permanent. A subsequent failure (such as abnormal program termination, communication failure, or hardware crash) will not cause the effects to be undone.

The transaction processing system is responsible for the atomicity, isolation, and durability properties. In doing so, it reduces all forms of failure during the execution of a transaction to one, consistency. A failed transaction aborts completely, with the appearance that nothing happened.

Nested transaction model

The Transaction Service allows transactions to be embedded within other transactions. Such transactions are called *nested* transactions or *subtransactions*. Standard tree terminology is used to describe relationships between nested transactions. A nested transaction is considered an indivisible operation within its enclosing transaction. Nested transactions have the same transactional properties, although with somewhat weaker guarantees. Subtransactions can be used to achieve safe parallelism and finer-granularity failure isolation.

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

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

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

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

Subtransactions isolate the effects of failures from the enclosing transaction and from other concurrent subtransactions. A subtransaction that has not completed may abort without causing its parent transaction to abort. Only the effects of the subtransaction are undone. More work, perhaps compensating for the failure, can be performed as part of the parent transaction, and the parent can be committed.

A committed subtransaction is permanent with respect to its parent: if the parent transaction aborts, the committed child is aborted. The child is said to be *committed with respect to* its parent. If one transaction is committed with respect to another, then all descendants of the first transaction are committed with respect to the second transaction and its descendants. Every transaction

is committed with respect to itself and its descendants. To abort a committed transaction, one must abort all of the transactions that are committed with respect to it.

Transactional application program components

The participants in a transaction are called *applications*. An application is any program that includes TRAN, either a client or a server program. An application defines a consistent set of updates that should be executed together. The implementation of transactions involves several distinct modules in application programs.

- The *application logic* (APPL) module defines the contents of the transaction, and is responsible for the consistency property.
- *Communications service* (COMM) modules provide mechanisms for the application program to make requests of other application programs.
- *Recovery service* (REC) modules manage modifications to permanent storage maintained by the application program.
- The *Transaction Service* (TRAN) module provides a basic interface that allows the application logic to begin, commit, and abort transactions. TRAN also provides an advanced interface for determining the state of transactions; one use of this interface is to implement the isolation property.

The terms *communications service* and *recovery service* refer to any modules that satisfy the functional descriptions in this chapter, not specific Toolkit components of similar names.

TRAN uses interfaces to the communications and recovery services to implement the atomicity and durability properties. These components are all bound into the application program. For example, APPL calls TRAN to define the transaction endpoints, COMM to invoke other server applications, and REC to update recoverable storage maintained with the application. Figure 2 on page 16 shows the interactions between the components in one application during a simple transaction. TRAN itself is a program library, not a separate service – calls to TRAN are local procedure calls.

Figure 2 on page 16 shows interactions between the components during a simple transaction. The interactions are described in the following steps:

Note: In all of the diagrams in this document, arrows denote the call to and return from a procedure.

1 The application calls **tran_Begin** to begin a new transaction.

2 The application performs work on behalf of the transaction locally (2A) and remotely (2B). This work can take place in parallel.

2A* The application calls its recovery service to modify recoverable storage.

2A1* A locking service provides synchronization to achieve the *serializability* property for transactions.

2A2* A log service provides permanent storage to achieve the *permanence* property for transactions.

2B* The application issues requests to other applications through its communications service.

2B1* The communications service calls TRAN to acquire transaction state data to deliver to TRAN in the remote application. The communications service transmits this data to the remote application along with the application data. This data permits TRAN to correlate all work done on behalf of the same transaction. Figure 5 on page 43 shows this process in greater detail.

2B2* The communications service sends messages to and receives messages from another application. TRAN places no constraints on the medium or protocol used for these interactions.

3 The application calls **tran_End** to indicate that work is finished. TRAN uses a commitment protocol to achieve the *atomicity* property for transactions – either all participants receive a *commit* outcome and the work is done in its entirety, or all participants receive an *abort* outcome and undo all work done on behalf of the transaction.

3A*, 3B* TRAN directs the recovery service to write a log record and the communications service to send and receive messages during the commitment protocol. Subsequent figures show these operations in greater detail.

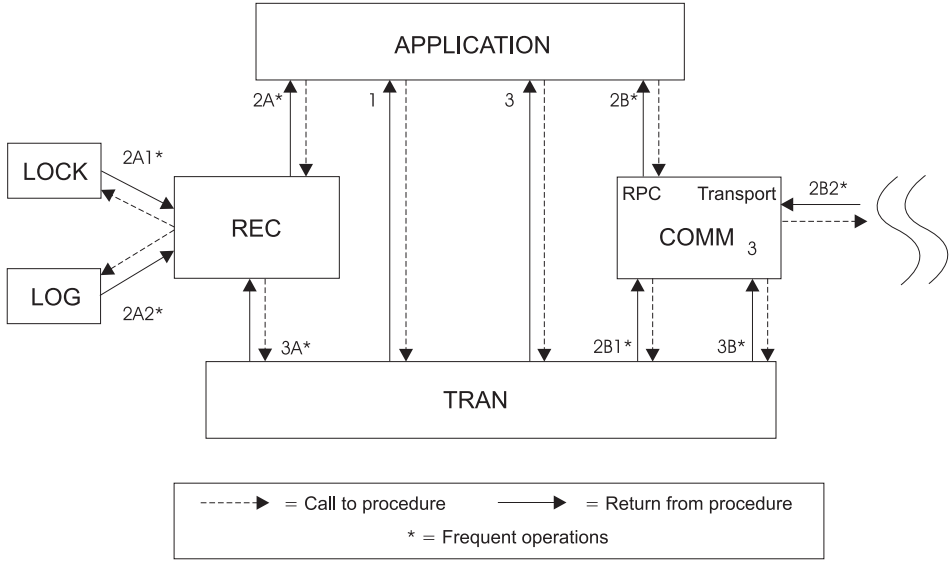

Figure 2. Application components: sample interactions

Application definition

Any program that participates in transactions is called an *application*. Application programs may manage permanent storage directly, such as a database; if so, the part of the application that maintains that storage can be considered a recovery service. Application programs may instead make requests to other programs to manipulate storage; if so, they are considered to use a communications service to make those requests. Applications are further classified according to their use of communications and recovery services.

Applications may be stand-alone programs, client programs, or server programs. An application is a program that executes independently of others. Typically, an application is a single operating system process. It may use multiple threads of control within a single address space, such as those provided by the Distributed Computing Environment (DCE), available from the OSF. An application may also be constructed using multiple operating system processes, provided that all application components and their data are shared; processes that do not share address memory must be distinct applications. TRAN treats applications as the unit of failure—an application can fail independently of others, but if any part of an application fails, the whole application fails.

CAUTION:
The DCE threading model permits the cancellation of threads. However, the TRAN Service assumes that its work will not be interrupted by such a cancellation. TRAN calls should only be made from threads with thread cancellation disabled. If a thread executing a TRAN Service function is cancelled, the results of that function are undefined.

Applications that have communications services may be *clients* or *servers*. A client is an application that requests work of another application. A server is an application that accepts requests from another. A single application can act as both a client and a server. An application that contains no communications service is called a stand-alone application.

Applications are also classified based on whether they contain a recovery service. Recoverable applications contain at least one recovery service. Ephemeral applications contain no recovery services. *Recoverable* applications are expected to be restarted after failure or termination. *Ephemeral* applications never restart once they terminate. Some of the functions of TRAN depend on being restarted after a failure; therefore, some of the interface function may not be available to ephemeral applications.

Note: The terms communications service and recovery service refer to modules within an application that perform the general functions defined in this chapter. They should not be confused with specific communications or recovery service implementations, such as other components in the Encina Toolkit. Many other communications and recovery service implementations are possible. The interfaces between these services and other application modules is not specified by TRAN; those interfaces are discussed as an example of how they may be structured.

Application logic component

The application logic (APPL) is the main program. Each application contains an application logic component; the remainder of this section uses the term application to refer to either the application logic component or the entire application program. In a distributed transaction involving several applications, the application logic component in one of the applications calls TRAN to begin a transaction. TRAN returns a new transaction identifier that can be used to correlate the work done as part of the transaction, and to make other Transaction Service calls. The application logic can make calls to communications services to make requests of other applications. It may make calls to recovery services to transactionally modify permanent storage maintained by the application.

Once all of the operations that make up a transaction have been completed, the application that began the transaction calls TRAN to end, asking that it be

committed. The application logic may call TRAN to abort the transaction, if for any reason the operations that make up a transaction cannot be completed correctly.

Communications service component

The *communications service* (COMM) component manages access to other applications. This component usually contains the run-time support for a *remote procedure call* (RPC) mechanism that permits an application to transparently invoke a procedure in another application. Although the remote procedure call is a common paradigm for performing work in another application, the communications service is free to use another paradigm.

TRAN cooperates with applications to implement *transactional remote procedure calls*. Every transactional RPC is called from within a transaction. If the RPC fails, the transaction is aborted. The remote function may still have been invoked, but TRAN informs the remote application so that updates can be undone. The remote function is effectively executed exactly once if the transaction commits and not at all if it aborts; from the transaction's point of view the function is guaranteed to execute exactly once.

This runtime support collects procedure arguments and sends a *message* to the remote application where the procedure is invoked; the run-time support in the remote application invokes the proper procedure, collects results, and sends a message back to the original application. The communications service must make calls to TRAN to ensure that remote work is done on behalf of the same distributed transaction. The communications service must also provide an interface that TRAN can call to send other messages to TRAN in another application. "Communications service interface" on page 41 describes the communications service interface.

Recovery Service component

The *Recovery Service* (REC) component provides access to persistent storage. The Recovery Service component is optional. This component usually provides an interface that allows the application to modify recoverable storage objects. It may use (or include) a *lock service* (LOCK) to coordinate shared access to these objects and a *log service* (LOG) to make changes to these objects permanent. A recovery service must also export an interface TRAN can call to access permanent storage and to direct recovery actions, described in "Recovery Service interface" on page 46.

Transaction Service component

TRAN provides an interface to the application logic for defining transactions, and uses interfaces to the communications and recovery services to implement the transactional properties. The application interface contains advanced functions that may be used to achieve the isolation property; some application logic module is responsible implementing this property. TRAN within one

application uses an interface to the communications services in that application to exchange information with TRAN in other applications. This permits TRAN to implement the atomicity property for distributed transactions. TRAN uses an interface to the recovery service to ensure that work done within an application is atomic and durable. "Transaction Service functions" goes into greater detail on the function of TRAN during the lifetime of a transaction and the process of committing a transaction.

Transaction Service functions

TRAN serves two basic purposes: it assists in the execution of a transaction; and, it coordinates the applications involved in a transaction to achieve the atomicity and durability properties. TRAN provides functions for beginning, ending, and aborting transactions, and for finding out when interesting events occur during the lifetime of a transaction. TRAN implements a distributed commitment protocol to achieve atomicity among all of the applications involved in a transaction. This commitment protocol can be viewed in two different ways: the global scheme used to achieve atomicity; and, the local actions that take place within each application. Figure 3 on page 20 depicts the message exchanges and local actions that take place between two participants using a standard two-phase commit protocol.

Transaction execution

TRAN passively assists the application while the transaction is active. It provides functions to begin, end, and abort a transaction. It provides functions that allow a communications service to spread a transaction to another application. It provides functions for finding out when transactions commit and abort; these can be used to implement isolation policies. TRAN only takes action as the result of calls made by the applications involved in the transaction.

Applications can only perform work on behalf of a transaction while it is *active*. A transaction is considered active in an application when one of the following conditions is true:

- The application began the transaction, but has not ended it.
- The application has accepted a request to do work from another application, and it has not returned from that request.
- The transaction is specifically permitted to do work by TRAN during the commitment of a transaction.

Many of the TRAN interface functions are only valid when a transaction is active.

Distributed commitment

TRAN achieves atomicity by ensuring that each application receives the same *outcome* (commit or abort) for a given transaction. TRAN gives each application an opportunity to *prepare* any work it has done. TRAN then

delivers the outcome to each participant. The TRAN at each application involved in a transaction exchanges information with the TRAN at each other application using the interface to the communications services. The exact nature of the message exchanges is not exposed from TRAN; however, some of the interface functions specifically constrains TRAN to using specifically the *two-phase commit* protocol.

TRAN must give each application an opportunity to prepare before it can commit the transaction. Until an application prepares, it is permitted to abort the transaction, even if it is not active. Once an application prepares, it must abide by the outcome provided by TRAN. Under certain circumstances, TRAN may ask an application to prepare before all other applications have stopped working on the transaction; if additional work takes place in an application that has prepared, TRAN will ask it to prepare again.

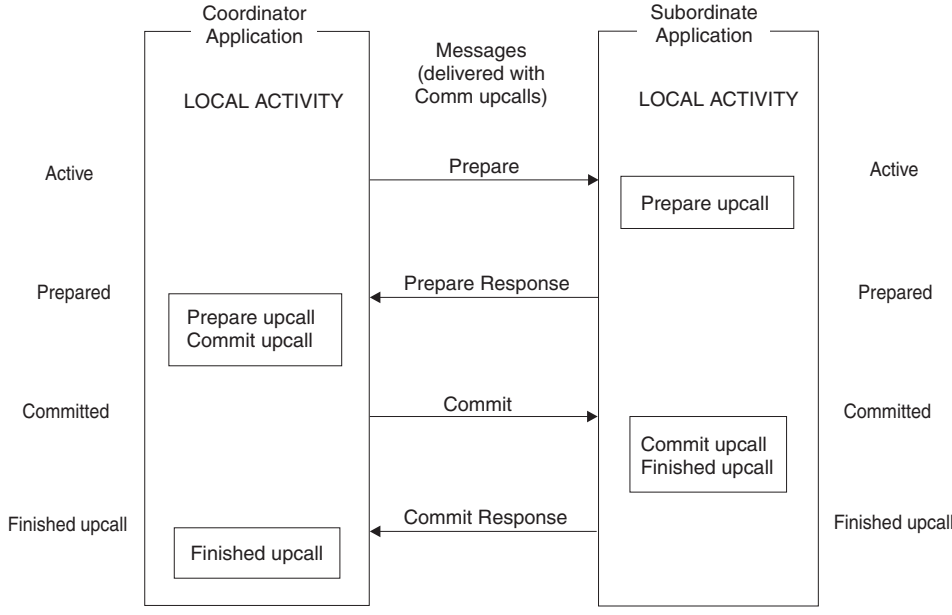

Figure 3. Two-phase commit message traffic

The two-phase commit protocol is one possible algorithm for arriving at a distributed outcome. The participants in the commit protocol are TRAN components in the applications that took part in the transaction. One distinguished participant called the *coordinator*, begins the protocol by sending a message to other *subordinate* participants asking them to prepare. Each subordinate application prepares, and then sends a response message to the coordinator to indicate that it has done so. Once all subordinates have prepared, the coordinator sends a message to each participant to indicate that the transaction has committed. The subordinates record the fact that the

transaction has committed, and send response messages to the coordinator. Once the coordinator has received commit responses from all subordinates, it forgets about the transaction. In order to use the same communication routes that the transaction did during its lifetime, the commit protocol may have some participants act both as a subordinate and as the coordinator for some other participants; a participant doesn't respond to a prepare or commit message until it has received prepare or commit responses, respectively, from its subordinates.

Unless specifically directed to use two-phase commit, TRAN may use other commit protocols. It may do so to achieve better performance or higher reliability. The only constraint is that all applications must be given an opportunity to prepare before the transaction can commit.

Local commitment sequence

Each application has certain rights and responsibilities in the commitment protocol. The application logic is given an opportunity to perform additional work on behalf of the transaction before the application can be considered prepared. Each recovery service in the application is then asked to prepare any work it has managed, and to store some data for TRAN. Once all recovery services have prepared, the application is no longer permitted to abort the transaction. TRAN promises to provide the distributed outcome regardless of intervening failure. If the application logic and the recovery services did not participate materially in the transaction, and they do not care about the outcome of the transaction, they may permit TRAN to deliver a local outcome that may not match the distributed outcome delivered to other applications. After TRAN delivers an outcome and it no longer needs to remember the transaction, it directs the recovery services to discard the data they wrote when preparing. The local events that take place during commit processing are reported by calling procedures registered by the recovery service during initialization, or by the application logic during the lifetime of the transaction.

The application logic is given an opportunity to perform last-minute work on behalf of a transaction before the transaction is committed. For example, an application may cache information in volatile storage during the lifetime of an application and only write it to permanent storage (either by calling a recovery service if it has one, or a communications service to call a server to do so) when the transaction is about to complete. Any form of work that would be legal during the lifetime of a transaction is legal at this time. This is also the application logic's last chance to abort the transaction should it so desire. This step consists of calling each before-prepare callback procedure registered by the application logic.

Each recovery service is asked to prepare its work and to write some data for TRAN. By preparing, the recovery service promises that it can make the

effects of the transaction durable if the transaction commits, or undo those effects if the transaction aborts, regardless of intervening failure. Typically, this requires that it force a description of changes made by the transaction to a log file on permanent storage, along with an indication that the transaction should not be undone after a failure. When it asks a recovery service to prepare, TRAN provides some data that the recovery service must also write to permanent storage. After a failure, the recovery service must return this data to TRAN so that TRAN can reconstruct the state of the transaction. The recovery service will normally incorporate TRAN data into its own prepare indication that it writes to its log. TRAN invokes the recovery-prepare upcall to ask a recovery service to prepare.

TRAN becomes responsible for delivering an outcome to the application once it has become prepared. It actively tries to determine the distributed outcome by exchanging messages with TRAN in other applications. If the application fails, the recovery service replays to TRAN the data it wrote as part of its prepare indication, permitting TRAN to resume the protocol. The outcome is delivered by calling the recovery-commit upcall (or the recovery-abort upcall) provided by each recovery service, each after-resolution callback registered by the application logic, and by returning from **tran_End** if the application began the transaction. Before calling the recovery-abort upcalls, TRAN invokes any before-abort callback functions registered by the application logic.

TRAN will deliver the *distributed outcome* to the application if the application participated materially in the transaction. A recovery service indicates that it has done work (such as modified permanent storage) by preparing normally. A recovery service alternatively may indicate that it participated in the transaction *read-only*, meaning that there would be no visible difference between committing and aborting. For example, the recovery service in a database server application might indicate that it was read-only if it responded to queries but performed no updates on behalf of a transaction. The application can also inform TRAN that it materially participated in the transaction, even if none of its recovery services did. If the application or any of its recovery service participated materially, TRAN will deliver to this application the same outcome that is delivered to other applications that participated.

TRAN will deliver a *local outcome* if the application did not participate materially in the transaction. The outcome delivered to the application may be different from that delivered to other applications. The benefit to accepting a local outcome is that it is more efficient—TRAN may be able to exchange fewer messages and may not need the recovery services to write data to permanent storage on its behalf. A disadvantage is that the application cannot truthfully report the outcome.

TRAN makes only limited outcome delivery guarantees to ephemeral applications. First, an ephemeral application contains no recovery services, so unless the application logic specifically states that it materially participated, TRAN will deliver a local outcome. Second, TRAN will only promise to deliver the distributed outcome for a reasonable period of time (as determined by the ephemeral application and others). After that time, a local outcome will be delivered.

When TRAN has fulfilled all of its commitments, it instructs the recovery services to finish the transaction. The recovery service must retain in permanent storage the prepare data provided by TRAN until this time. The instruction to finish requests that the recovery service discard that data. The finished indication occurs sometime after TRAN has provided an outcome to the application. TRAN invokes the recovery-finished upcall.

Important abstractions

This section describes the important abstractions exported by TRAN other than the transaction model itself.

Transaction identifier

A *transaction identifier* is a short temporary identifier for a transaction. It is represented by the exported type **tran_tid_t**. The application can obtain valid transaction identifiers only from TRAN. It must not obtain them directly from other applications, because two applications may be given different transaction identifiers for the same distributed transaction. Each transaction identifier is unique for a given application until that application terminates. Any transaction that is prepared when the application terminates is given the same identifier when the application restarts. Other transaction identifiers may be reused after restart.

Application identifier

Each application is permanently and uniquely named by an *application identifier*. This name can be used by the application to identify itself and by the Transaction Service to identify each participant in a transaction. The name must be permanent so that recoverable applications can be contacted after a failure. See Section “Naming policy for variable-sized objects” on page 27 for more information on the format of application identifiers.

The communications service is responsible for generating the application identifier the first time the application executes. An ephemeral application program is considered a new application every time it is executed and is called upon to generate an application identifier each time. A recoverable application is called upon to generate an application identifier exactly once; TRAN reconstructs the application identifier from its permanent storage when the recovery service for the application initializes.

Service identifier

A service identifier is a temporary short name for an application component: the application itself, a communications service, or recovery service. Each component is assigned its own name when it initializes. The value 0 (zero) is assigned to the application itself.

Abort descriptors

An application that initiates an abort describes its reason for doing so. The convention for describing abort reasons stipulates that the following information be provided:

- *abort data format*—A globally, permanently unique identifier to define how the abort reason is to be interpreted. Typically, this defines the type of application or the module that generates the reason, but one module may publish more than one format.
- *abort data*—The reason that the transaction was aborted. The abort data format identifier allows an application to determine how this data should be interpreted.
- *abort source*—The identifier of the application that aborted the transaction.

The abort information is stored in properties (see "Advanced functions" on page 35) that are propagated to other participants in the transaction; this information can be obtained in a before-abort callback (see "Abort callbacks" on page 34). Refer to "Abort data interpretation" on page 36 for more information on establishing and retrieving abort descriptions.

The Encina Abort Facility provides an alternative interface to setting, retrieving, and formatting abort reasons. Refer to Section "Chapter 5. Abort Facility" on page 91 for more information.

Transaction Service interface

An understanding of the structure of the TRAN interface is necessary to understand the remainder of the interface. An application is described as three logically independent components: the application logic, a communications component, and a recovery component. A separate interface section is provided for each component. TRAN uses three mechanisms to inform application components of interesting events: *return codes* from calls to TRAN; *upcall* procedures that TRAN can invoke for every transaction; and *callback* procedures that an application component can ask to be selectively invoked. Each time an application executes, each of its components must initialize its portion of the interface before new transactions can be created.

Return codes

Most interface functions return a value of type **tran_status_t** to indicate if they succeed. A function that completes successfully returns `TRAN_SUCCESS`. Any other return value indicates that some exceptional condition occurred and that out parameters are not defined unless the description of the call specifically

indicates otherwise. The description of each interface function includes a list of the status codes that it can return. The reference page for the **tran_status_t** data type describes all of the status codes defined for TRAN.

Upcalls

The application, the recovery service, and the communications service are each required to supply some procedures for TRAN to call. These procedures are referred to as upcalls. TRAN invokes upcalls only during calls into TRAN, from the threads that make those calls; therefore, an upcall procedure should never perform a non-local goto outside its scope. Only one upcall procedure is registered for a given purpose; this procedure is invoked on behalf of all transactions.

If an upcall blocks in application code, this may cause calls made into TRAN to block until the upcall completes. For example, the call into TRAN that ends a transaction typically does not return until an upcall to the recovery service successfully logs the transaction's commit record. This implies that the application should never hold a latch during a call to TRAN if it could be needed for completion of an upcall.

For convenience, this document uses the names of parameters that represent upcalls as names for the calls themselves. Application designers may use other names for these functions.

Callbacks

Several Transaction Service functions allow the application to specify procedures to be called when specific events occur. For example, the application can register a procedure to be called before a particular transaction prepares. Like upcalls, callback procedures are invoked only during calls into TRAN; the same cautions about blocking and non-local transfer of control apply. A call to any TRAN function can result in a callback; therefore a callback can occur from any thread that makes a call to a TRAN function.

Unlike upcalls, more than one callback may be registered for a given purpose. Each callback procedure is invoked once for each time it is registered. If multiple callback procedures are registered for the same event then they are all invoked in an unspecified order when the event occurs. The application can provide its own additional argument for each callback.

Transaction Service header files and libraries

Header files

Applications that link with TRAN must include the header file **tran/tran.h** in their C program.

libraries

The Encina TRAN functions are contained in the **Encina** library. See the *Writing Encina Applications* manual for further information on the libraries to use during compilation.

Data types

The TRAN interface defines two basic data types that are used throughout the interface: **tran_tid_t** and **tran_status_t**. Interface functions refer to transactions through identifiers of type **tran_tid_t** and typically return a status code of type **tran_status_t**. TRAN also defines a third type, **tran_abort_t**, for representing the reason for an aborted transaction in the form of an abort code.

In addition, a variety of more complex, special-purpose data types are provided to describe variable-length, application-defined data: applications, messages, log records, and force groups. TRAN provides functions for creating, manipulating, and interpreting all of these types.

Functions for transaction identification

The following functions are used to describe, identify, and compare the identifiers associated with specific transactions.

- **tran_TidEqual**
- **tran_TidHash**
- **tran_TidIsDescendent**
- **tran_TidIsRelated**
- **tran_TidIsTopLevel**
- **tran_TidKnownDescendents**
- **tran_TidParent**
- **tran_TidTopAncestor**
- **tran_TidToString**

Special-purpose data types

Special-purpose data types are provided to simplify the specification of TRAN interface functions. Several types are defined to contain data specified by the application:

- the application identifier (**tran_applId_t**)
- the property key (**tran_propertyKey_t**)
- the property value (**tran_propertyValue_t**)
- the address (**tran_address_t**)
- the address family (**tran_addressFamily_t**)
- the force-group identifier (**tran_forceGroupId_t**)

Other types are defined to contain data specified by the transaction service:

- the message (**tran_message_t**)
- the log record (**tran_logRecord_t**)
- the security key (**tran_securityKey_t**)

All of these types have exactly the same form: a variable-length sequence of bytes. TRAN provides special values and functions for creating, interpreting, and destroying objects of each of these types. All of these functions and values are usable as soon as the application has completed its call to the **tran_Init** function.

The application identifier, property key, address family, and force-group identifier are global names that must be globally unique. See "Naming policy for variable-sized objects" for more information on naming these four types of objects.

Naming policy for variable-sized objects

All variable-sized objects defined in an application share a global name space. Encina defines a policy for specifying subspaces within this global name space in order to prevent naming conflicts. The policy is that the value of the first byte specifies the naming subspace used for an object.

TRAN relies on the use of this naming policy in certain interface objects defined as variable-sized byte arrays. These objects include application identifiers, communications service address families, recovery service force groups, and transaction property keys. Applications that define values for these objects should always follow this naming policy to ensure that conflicts between naming subspaces do not occur. Table 2 shows the recommended first-byte values to use when naming variable-sized objects.

Table 2. First-byte values for naming variable-sized objects

First byte	Object
0	Objects defined by the TRAN component itself. This includes TRAN-defined property keys, application identifiers, communication service address families, and recovery service force groups.
1	Reserved.
2	DCE UUID subspaces. A DCE UUID follows the first byte. The *time_low*, *time_mid*, *time_hi_and_version*, *clock_seq_hi_and_reserved*, and *clock_seq_low* fields are encoded in that order, in most-significant-byte-first order. Customers may create their own subspaces using uuids that they generate.
3	Internet host name subspaces. A null-terminated, ASCII string follows the first byte. Customers may create their own subspaces using Internet host names that have been assigned to them.

Table 2. First-byte values for naming variable-sized objects (continued)

First byte	Object
4-31	Reserved for subspaces based on other prevalent naming standards.

Any first-byte values beyond 31 are defined by the respective subspace owners.

The TRAN interface defines several other variable-sized objects including addresses, property values, security keys, log records, and message contents. These objects have an implicit scope that prevents naming conflicts from occurring.

Special values

Only three special types have special values. TRAN defines an abort data format value for its own use. The identifier for this application is established when the application components are initialized; the **tran_ApplIdLocal** function returns the application's own identifier. TRAN defines the `TRAN_MESSAGE_NULL` value to indicate that no message should be sent on its behalf (see "Communications service interface" on page 41).

In addition, TRAN defines several special values for the property key. The property `TRAN_PROPERTY_KEY_ABORT_FORMAT` should be used to store the key for an application's abort data; TRAN uses the `TRAN_PROPERTY_KEY_ABORT_DATA` key for the abort data when it aborts a transaction.

Creation functions

Functions are provided to create a new object of each special-purpose data type. The application component provides a generic pointer and a data length to define the contents of the object. TRAN *copies* the data; the application may de-allocate its copy of the data as soon as the function completes. The application component must call the appropriate destruction function (see "Object destruction functions" on page 31) when it no longer needs the object.

TRAN defines the following creation functions:

- **tran_AddressCreate**
- **tran_AddressFamilyCreate**
- **tran_ApplIdCreate**
- **tran_ForceGroupIdCreate**
- **tran_LogRecordCreate**
- **tran_MessageCreate**
- **tran_PropertyKeyCreate**
- **tran_PropertyValueCreate**

- **tran_SecurityKeyCreate**

Construction functions

Construction functions are provided to create a new object of each type from existing data. The application component provides the data, defined by a generic pointer and a length, to be contained in the object. The construction functions store a reference to the original data rather than copying the data. The application component must not de-allocate or change the data it uses to construct an object until that object is destroyed; TRAN also promises not to change the data provided. The application component can specify that TRAN is responsible for freeing the data by calling a destructor function when it no longer needs the data; TRAN is permitted to free this memory whenever it deems appropriate.

TRAN defines the following construction functions:

- **tran_AddressCons**
- **tran_AddressFamilyCons**
- **tran_ApplIdCons**
- **tran_ForceGroupIdCons**
- **tran_LogRecordCons**
- **tran_MessageCons**
- **tran_PropertyKeyCons**
- **tran_PropertyValueCons**
- **tran_SecurityKeyCons**

Copy functions

Copy functions are provided to create a new object of each type from an existing object. The application component provides an object of the appropriate type to be copied. TRAN *copies* the data contained in the source object into a newly created object. The application component must call the appropriate destruction function (see "Object destruction functions" on page 31) when it no longer needs the object.

TRAN defines the following copy functions:

- **tran_AddressCopy**
- **tran_AddressFamilyCopy**
- **tran_ApplIdCopy**
- **tran_ForceGroupIdCopy**
- **tran_LogRecordCopy**
- **tran_MessageCopy**
- **tran_PropertyKeyCopy**
- **tran_PropertyValueCopy**

- **tran_SecurityKeyCopy**

Data access functions

Functions are provided to retrieve the data contained in an object. One function returns a generic pointer to the data; another function returns the valid length of that data in bytes. The data pointer only remains valid as long as the original object; if the data is required, the caller should copy the data before the object is destroyed.

TRAN defines the following data-access functions:

- **tran_AddressData**
- **tran_AddressLength**
- **tran_AddressFamilyData**
- **tran_AddressFamilyLength**
- **tran_ApplIdData**
- **tran_ApplIdLength**
- **tran_ForceGroupIdData**
- **tran_ForceGroupIdLength**
- **tran_LogRecordData**
- **tran_LogRecordLength**
- **tran_MessageData**
- **tran_MessageLength**
- **tran_PropertyKeyData**
- **tran_PropertyKeyLength**
- **tran_PropertyValueData**
- **tran_PropertyValueLength**
- **tran_SecurityKeyCons**
- **tran_SecurityKeyLength**

Comparison functions

TRAN provides functions to determine whether two objects are equal or identical. Functions to compare objects for equality are provided for those types where the contents are supplied by applications; two objects are considered equal if they contain the same number of data bytes and those bytes are equal. The **tran_MessageIdentical** function is provided to quickly test whether two messages created by TRAN are identical; two messages that are identical have the same contents. No comparison function is required or provided for the log record type. Each comparison function returns `TRUE` if the objects are equal or identical, and `FALSE` if the objects differ.

TRAN defines the following comparison functions:

- **tran_AddressEqual**

- **tran_AddressFamilyEqual**
- **tran_ApplIdEqual**
- **tran_ForceGroupIdEqual**
- **tran_MessageIdentical**
- **tran_PropertyKeyEqual**
- **tran_PropertyValueEqual**
- **tran_SecurityKeyEqual**

Object destruction functions

Functions are provided to destroy objects when they are no longer needed. Application components must use these functions for objects that they create or construct, and for objects that TRAN provides in interface calls, upcalls, and callbacks. TRAN never destroys objects provided by application components in interface calls. Once an object has been destroyed, it may not be used, nor may any pointers to its contents acquired using the data access functions.

TRAN defines the following object-destruction functions:

- **tran_AddressDestroy**
- **tran_AddressFamilyDestroy**
- **tran_ApplIdDestroy**
- **tran_ForceGroupIdDestroy**
- **tran_LogRecordDestroy**
- **tran_MessageDestroy**
- **tran_PropertyKeyDestroy**
- **tran_PropertyValueDestroy**
- **tran_SecurityKeyDestroy**

Array and string destruction functions

Some interface functions return strings or arrays of objects. These arrays may be indexed normally. The application must call the appropriate destruction function when the string or array is no longer needed. Destroying an array does not destroy its elements; they must be destroyed separately, as desired.

TRAN defines the following array- and string-destruction functions:

- **tran_PropertyValueArrayDestroy**
- **tran_StringDestroy**
- **tran_TidArrayDestroy**

Initialization and termination

An application must perform several steps each time it begins execution. First, the application must specify its environment. Second, it must initialize the application interface. Third, its communications and recovery services (if present) must initialize their portions of the interface. When all of these initialization operations have been completed, the application indicates that it is ready to begin work. Figure 4 on page 33 illustrates the initialization sequence for an application, which is described in the following steps. Arrows denote the call to and return from a procedure.

1 The application defines an environment. The **tran_StandardEnvironment** function specifies that the Encina Base Development Environment (BDE) should be used for basic services, such as memory allocation and synchronization. The **tran_SpecialEnvironment** function may be used by advanced applications to specify other functions that provide these services.

2 The application initializes TRAN interface. The **tran_Init** function indicates that the environment has been defined, and that other application components are ready to begin initialization.

3 The application calls its component services (recovery and communications) to perform their initialization. Each service is optional. If both services are present, they may be initialized in either order, or in parallel.

3A The application calls its recovery service to perform its initialization.

3A1 The recovery service calls **tran_RecInit** and replays transaction service log records. “Restart” on page 50 provides more detail on the recovery service initialization.

3B The application calls its communications service to perform its initialization.

3B1 The communication service calls **tran_CommInit**.

The application calls **tran_Ready** after its component services have finished their initialization.

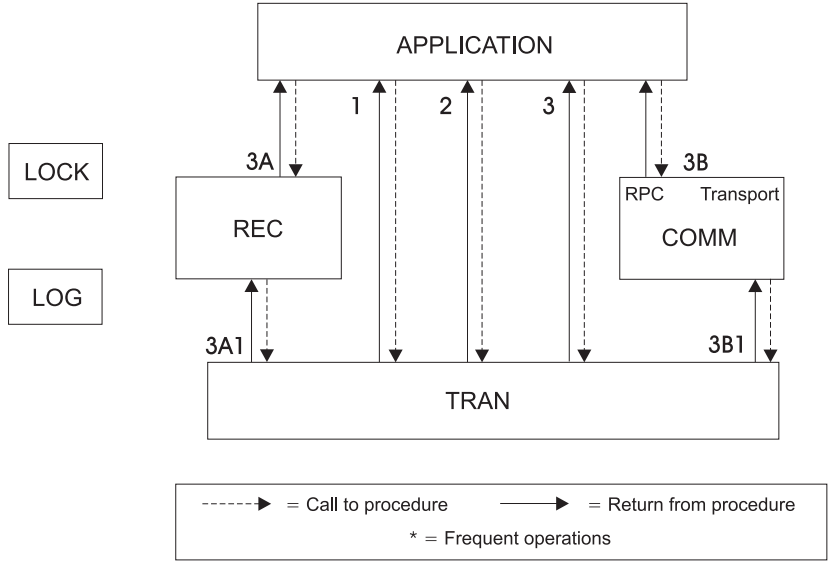

Figure 4. Initialization sequence

The application must specify its environment before making any other initialization calls. The application environment includes functions that TRAN can use to perform various system functions, to allocate memory, to synchronize, and to report fatal application errors. The **tran_StandardEnvironment** function specifies that the Transaction Service (TRAN) should use the standard Toolkit environment functions. The **tran_SpecialEnvironment** function permits sophisticated applications to specify one or more of their own environment functions. "Application environment specification" on page 53 provides more information on specifying the application environment.

The application must call **tran_Init** after configuring its environment. This function initializes the application interface. Once this step is complete, the application can use the data structure manipulation functions described in "Special-purpose data types" on page 26.

Other application components must be initialized next. The recovery service must call **tran_RecInit** and replay log records written on behalf of TRAN. The communications service must call **tran_CommInit**. These components are optional. If present, they may be initialized in either order or in parallel.

The application must call **tran_Ready** when all of its components have been completely initialized. Once this step is complete, the application may use the remainder of the application interface.

The application must call **tran_Terminate** before exiting. After calling **tran_Terminate**, the application must re-initialize the interface.

Application interface

This section describes the application interface to TRAN. This interface allows the application to begin, commit, and abort transactions and to determine the outcomes of transactions. It also allows the application to influence certain transaction management decisions, such as the choice of a coordinator.

Beginning and ending transactions

An application explicitly begins a transaction by calling **tran_Begin**. The same application may call **tran_End** to attempt to commit the transaction. Any participant may call **tran_Abort** to abort it. Any participant may call **tran_AbortFamily** to abort all members of a transaction family that have been active in that application.

Application status

An application can call **tran_ApplIsRecoverable** to determine whether the local application is recoverable and **tran_ListTransactions** to list the transactions in use by an application.

Transaction state

The following data types define the possible states of a transaction:

- **tran_globalState_t**
- **tran_localState_t**
- **tran_outcomeQuality_t**
- **tran_relativeCommitState_t**

The following functions are used in determining the state of a transaction:

- **tran_GetGlobalState**
- **tran_GetLocalState**
- **tran_GetRelativeCommitState**

Application callbacks

Restart callbacks

The application can call **tran_CallDuringRestart** to request that a procedure be invoked after TRAN has recovered its state during restart. The application can call **tran_CallAfterRestart** for a second restart pass, where calls affecting the outcome, such as **tran_Abort**, can be made.

Abort callbacks

The application can call **tran_CallBeforeAbort** during a transaction to request that a procedure be invoked before the transaction aborts, if it does abort. TRAN passes the callback procedure information about the cause of the abort.

This callback is useful, for example, when threads executing on behalf of the transaction must be stopped before the transaction's updates are undone. All before-abort callbacks for an aborted transaction are invoked before any after-resolution callbacks are invoked for any member of the same transaction family.

Prepare callbacks

The application can call **tran_CallBeforePrepare** during a transaction to request a callback before the transaction prepares. The application can call **tran_CallTransactionallyBeforePrepare** during a transaction to perform additional transactional work before the transaction prepares.

TRAN guarantees that transactional before-prepare callbacks are executed before nontransactional before-prepare callbacks. Ordering within each type of callback is undefined.

Relative commitment callbacks

If the application supports nested transactions, it can call **tran_CallAfterCWRT** to request that a procedure be invoked after a transaction commits with respect to another in the same family, or when either of the two transactions aborts. This callback is typically used for passing locks between family members.

Transaction resolution callbacks

The application can call **tran_CallAfterResolution** during a top-level transaction to request that a procedure be invoked after the transaction's outcome is determined. This callback is typically used for dropping locks or for deallocating data structures associated with a transaction.

Transaction completion callbacks

The application can call **tran_CallAfterFinished** to request a callback when all communications and recovery functions associated with the transaction have been completed.

Advanced functions

Transaction and application properties

TRAN provides a facility for associating property data with specific transactions or with the application. This data takes the form of a list of *(key, value)* pairs. TRAN provides the **tran_PropertyAdd** function for adding new properties and the **tran_PropertyRetrieve** function for retrieving properties that have been installed. Properties can be associated with a specific transaction or with the application. Properties associated with a transaction are propagated to other participants in the transaction along with transactional RPC messages and other Transaction Service messages.

Properties are recovered after a failure if the application is recoverable. The propagation and recovery aspects of the facility make properties expensive, so they should be used sparingly.

Coordinator selection

Normally TRAN chooses a coordinator for each transaction. Sometimes the participating applications should choose a coordinator based on application-specific considerations. They should base this decision on which application is mostly likely to remain available during the commit and which has the most to lose if the commit blocks. In many cases, the decision need not involve explicit negotiation. The choice of a coordinator may be implied by client-server relationships, levels of authentication, or some other standard acceptable to all participants. Ephemeral applications cannot be coordinators.

Any participant in a transaction can call **tran_SetCoordinator** to identify its preferred coordinator and indicate whether it insists on this coordinator or is willing to prepare with a different one. TRAN selects a coordinator according to the following rules:

- If participants insist on more than one coordinator, the transaction aborts.
- If participants insist on one coordinator, that one is chosen.
- If no participant insists on a coordinator, one is chosen arbitrarily from those suggested.

A participant can call **tran_GetCoordinator** during a before-prepare callback or recovery-prepare upcall to make sure an acceptable coordinator was chosen.

Coordinating a transaction is enough of a burden that some applications prefer to decide whether they are willing on a case-by-case basis. The application can call **tran_SelectivelyCoordinate** to indicate that it wants an upcall whenever it is asked to coordinate a transaction that it did not begin.

Abort data interpretation

TRAN generates an *abort reason* when it causes a transaction to abort. TRAN only aborts transactions when it must (for example, because a committed child or a parent transaction has been aborted) or when some failure (for example, message delivery, application failure) occurs. An application can retrieve a permanent form of the abort reason using the conventional abort properties (see **tran_Abort**). An application can call **tran_AbortDataToReason** to convert this permanent form, a property value, to a value of type **tran_abort_t** for comparison with known reasons. An application can call **tran_AbortReason** to get just the **tran_abort_t** form of the abort reason for a known transaction. The reference page for the **tran_abort_t** data type describes the reasons TRAN gives for aborting transactions.

Secure communications

TRAN provides an interface for securing an RPC and asynchronous messages it uses to transmit transaction state. Although a communications service is responsible for authenticating the source of transaction service messages, it cannot verify that the source is authorized to participate in the transaction represented by the message, because it cannot interpret the message. An application can call **tran_Secure** to provide data immune to forgery for secure communications.

Pre-prepare

A server can save a round of messages by unilaterally preparing before it returns from the last RPC in a transaction. The server calls **tran_PrePrepare** to give a hint to TRAN that it should prepare early. TRAN may or may not actually prepare early.

Application-controlled prepare

An application can call **tran_DeferCommit** to indicate that it wishes to explicitly provide an outcome in the event that it coordinates the transaction. Typically, a module that uses **tran_DeferCommit** also calls **tran_SetCoordinator** to insist that the application coordinate. When the beginner of the top-level transaction calls **tran_End** (or **tran_Prepare**, see below), each application that has participated in the transaction family is given an opportunity to prepare. Once all participants prepare, TRAN normally commits the transaction; however, if the coordinator application has elected to defer commitment (by calling **tran_DeferCommit**), TRAN instead invokes the callback specified in the **tran_DeferCommit** call, and waits for the application to direct an outcome. The application module that deferred commitment must eventually call **tran_ProvideOutcome** to indicate that the transaction should commit or abort; however, it need not do so within the scope of the callback. An application module that defers commitment of a transaction remains responsible for providing the outcome until the transaction finishes.

The beginner of a transaction can call **tran_Prepare** in place of **tran_End** when it does not want to wait for the transaction to be resolved. The **tran_Prepare** call, like **tran_End**, indicates that the transaction is complete and that TRAN should initiate commitment. The **tran_Prepare** call returns as soon as the transaction is prepared; the **tran_End** call does not return until the transaction is resolved. An application that calls **tran_Prepare** is expected to have called **tran_DeferCommit** to ensure that it controls the transaction outcome.

Application-controlled prepare is useful whenever the outcome of one transaction should be tied to the outcome of another. For example, consider an application that acts as a bridge between two transaction management protocols by forwarding requests from clients in one protocol domain to servers in another. When it first encounters a transaction in the client domain,

it can begin a proxy transaction in the server domain. When it receives a prepare for the original transaction, it can prepare the proxy. When it receives an outcome for the original transaction, it can call **tran_ProvideOutcome** for the proxy.

Outcome delivery requirements

TRAN provides functions to allow application modules to request special outcome delivery requirements. Normally, outcome delivery is dictated by the recovery services present in the application. If no recovery service demands the distributed outcome for a transaction, TRAN may deliver a local (read-only) outcome instead. Application modules may call **tran_RequireDistributedOutcome** to specify that they require the distributed outcome, regardless of the recovery services needs. Normally, after-resolution callbacks may be lost in the event of an application failure. Application modules may call **tran_RequireCompleteOutcome** to specify that they need to execute their after-resolution callbacks even after a failure; then TRAN will permit them to be reregistered during restart. Normally, ephemeral applications receive a local outcome. Ephemeral applications can call **tran_SetEphemeralOutcomeRequirementLimit** and recoverable applications can call **tran_SetEphemeralOutcomeDeliveryLimit** to specify how long TRAN should attempt to provide an ephemeral application with the distributed outcome.

Outcome delivery control

TRAN provides functions for controlling the progress of the callbacks and upcalls associated with transaction completion. An application can call any of the following functions:

- The **tran_DelayAbort** function delays the invocation of recovery service upcalls when a transaction aborts. This permits the application to operate on behalf of a transaction without concern for its local recoverable resources changing while it is still working.
- The **tran_ProlongResolution** function indicates that the after-resolution phase should not be considered complete. This allows an application module to return from its after-resolution callback (to avoid blocking the thread that is invoking those callbacks) while still preventing the transaction from being finished prematurely.
- The **tran_ProlongFinish** function indicates that the after-finished phase should not be considered complete. This allows an application module to return from its after-finished callback but preserve the transaction identifier (and access to the transaction state through that identifier), without prolonging the resolution phase.
- The **tran_RequestPromptFinish** function indicates that the application waits synchronously for a transaction to *finish*, not just for an outcome (commit or

abort) to be delivered. This permits the transaction to encourage local recovery services and other applications to complete their finish phases promptly, if necessary.

Declare last call

The beginner of a transaction can use the **tran_DeclareLastCall** function to declare that its next transactional RPC will be its last. This enables TRAN to combine the first message exchange for transaction commitment with the RPC. The **tran_DeclareLastCall** function can be called to enable and disable this optimization.

Last-call optimization can only be used in certain circumstances. If the following conditions are not met, or the transaction has been aborted, then the function returns `TRAN_TID_NOT_VALID`:

- Only the beginner of the transaction may use the **tran_DeclareLastCall** function. (The **tran_PrePrepare** function can be used to make a similar statement for servers.) The beginner must not have called either the **tran_End** or **tran_Prepare** function previously.
- The transaction must be top-level.
- The transaction must not have any outstanding subtransactions or RPCs.
- The next RPC *absolutely must* be the last. No transactional work is allowed between the next outgoing RPC and the call to the **tran_End** function. Any attempt to do work (for example, calling the **tran_CommReceivedRequest** function to begin an RPC) will result in an error. TRAN will not do any heuristic damage reporting automatically; if the application believes that it has damaged transactional consistency through misuse of the **tran_DeclareLastCall** function, then the application can use the **tran_RecordHeuristicOutcome** function to initiate reporting.

There are also limitations on the last call itself (in addition to any normal limitations on RPCs). If the following conditions are not met, the transaction is aborted with `TRAN_ABORT_ILLEGAL_LAST_CALL` as the abort reason:

- The last RPC must be made on behalf of the top-level transaction.
- There must be no other RPCs or subtransactions in that transaction family outstanding at the time of the last call.

Once the optimization is enabled and the last call made, the beginner can enter the prepare phase of the commit process. This means that the beginner cannot abort the transaction unilaterally—it must wait for the outcome from the coordinator (or force a heuristic outcome, risking inconsistency). It also means that the beginner must not depend on the local state being `TRAN_LOCAL_STATE_ACTIVE` or the before-prepare callbacks not being fired before the **tran_End** function is called.

The Transaction Service does not deliver a commit result to the application until the **tran_End** function is called. This ensures that resolution callbacks (and consequently, all finished phase actions) will be held even though the global state reflects commitment.

Tuning parameters

TRAN allows the specification of internal *tuning parameters* for optimizing the performance of an application. These parameters define timing specifications and other implementation options that TRAN uses in delivering messages, such as commit and abort messages.

TRAN defines two functions for setting and retrieving internal tuning parameters. These two functions are defined as follows:

```
tran_status_t tran_GetTuningParameters(OUT char **stringP)
tran_status_t tran_SetTuningParameters(IN char *string)
```

For more information on tuning parameters and the use of these functions, contact your support representative.

Transaction identifier reservation

An application module might need to perform recovery on behalf of transactions for which TRAN has no responsibility, that is, before commitment of those transactions begins. To do so, it is useful for the application module to be able to reserve the original identifier for the transaction, so that the same identifier can be used after restart without concern for it also being used to identify a new transaction.

An application module can call the **tran_Reserve** function to reserve a transaction identifier during the restart phase (that is, between the **tran_Init** function and the **tran_Ready** function). TRAN allows any number of application modules to reserve a transaction identifier. Reserving an identifier does not affect the recovery of that transaction if TRAN has any recovery responsibility (through replay of log records by recovery services). If TRAN has no responsibility, the transaction will appear to be finished.

Recovery services can reserve an identifier for a transaction that they know to be finished by replaying an empty (“finished”) log record for that identifier; this facility, however, must be used very carefully. Instead of replaying empty transaction records, recovery services can use the **tran_Reserve** function to reserve transaction identifiers. Recovery services can use this function more safely in applications with multiple recovery services or other pseudo-recoverable modules.

Communications service interface

A *communications service* is a module within the application that manages transactional communications with other applications. A communications service provides the mechanism for invoking services in other applications; remote procedure call (RPC) is one such mechanism. TRAN provides an interface to the communications service for making remote invocations transactional; the communications service makes calls to acquire transaction state to be passed along with the invocation. A communications service must export an interface that TRAN can call to deliver transaction state information to other applications during commit processing. TRAN also exports an interface for the communications service to provide feedback about its communications abilities.

Each communications service must call the **tran_CommInit** function to initialize its interface to TRAN each time the application executes. It must make this call after the application calls the **tran_Init** function, but before it calls the **tran_Ready** function. The communications service must provide an *address family* to identify itself. This identifier must be unique among all communications services, but a given communications service must use the same identifier in all applications.

A communications service must call TRAN to acquire transaction state information to be transmitted along with remote invocations. The communications service in the client application must call the **tran_CommSendingRequest** function to inform TRAN that remote work is being done on behalf of a transaction. TRAN returns data to be transmitted along with the request. The communications service in the server application must call the **tran_CommReceivedRequest** function, providing this Transaction Service data; TRAN returns an identifier for the transaction in the server.

An analogous sequence takes place when the remote work is done: the communications service in the server calls the **tran_CommSendingReply** function to pick up Transaction Service data to be returned to the client; the communications service in the client calls the **tran_CommReceivedReply** function to provide the information to its Transaction Service. TRAN does not restrict the mechanism used for transmitting its data or the amount of communication that takes place between the **tran_CommReceivedRequest** and **tran_CommSendingReply** calls. TRAN does require that once the remote invocation is begun, all four calls must be completed; the communications service is responsible for aborting the transaction if it cannot complete a remote invocation or deliver TRAN data.

Figure 5 on page 43 illustrates the sequence of calls for a simple RPC that is described in the following steps. Arrows denote the call to and return from a procedure.

1 The client application initiates a remote procedure call. It actually invokes a local *stub* procedure that collects procedure arguments into a *request* message for transmission to the remote application.

2 The client stub procedure calls **tran_CommSendingRequest** to pick up data from TRAN. It incorporates this data in the message to be transmitted.

3 The request message is transmitted to the server application. The transport medium (such as network shared memory) and protocol used to transmit the message are not constrained by TRAN. Any number of actual messages may be used to transmit the remote procedure call arguments and TRAN data.

4 Another stub procedure in the server application receives the message and calls **tran_CommReceivedRequest**, which supplies TRAN data from the message and acquires a transaction identifier in return.

5 The server stub procedure calls the actual server procedure with the arguments from the message. Any results from the server procedure are collected into a *reply* message.

6 The server stub procedure calls **tran_CommSendingReply** to pick up Transaction Service data to be incorporated into the reply message.

7 The reply message is transmitted back to the client application.

8 The client stub procedure calls **tran_CommReceivedReply** with TRAN data from the reply message.

9 The client stub procedure returns the remote procedure call results to the application.

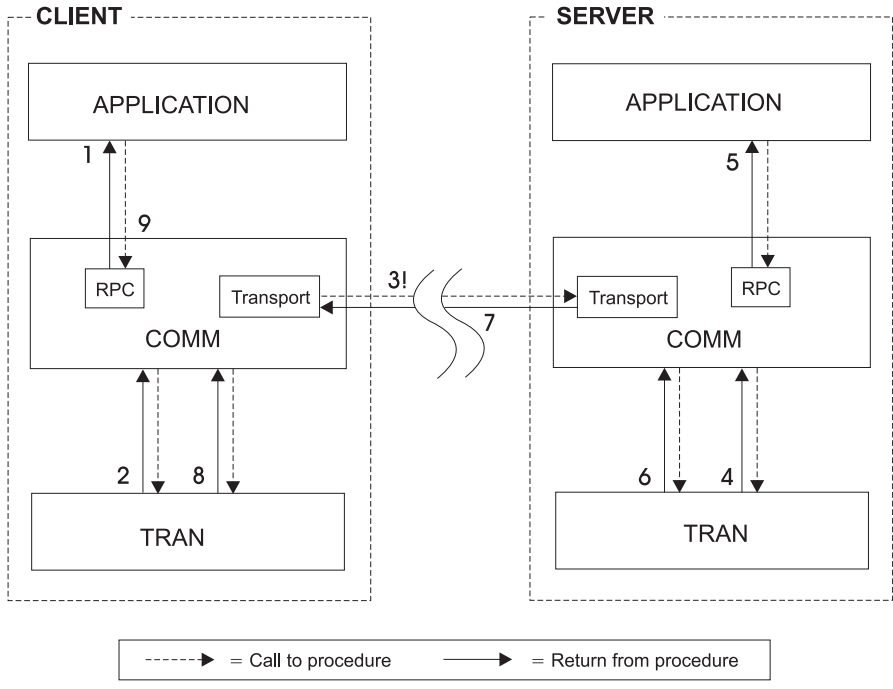

Figure 5. Transactional communication example: simple RPC

Each communications service also provides an interface to deliver information for TRAN during commit processing (see Figure 6 on page 44). The communications service provides an *address* for the other application in each of the remote invocation calls (**tran_CommSendingRequest**, for example). The address must contain sufficient information for the communications service to deliver additional Transaction Service data to the same application, even after recovering from an application or communications failure. A Transaction Service invokes the comm-send upcall procedure to ask the communications service to deliver Transaction Service data to TRAN in another application, providing the application identifier and address for the destination. The communications service in the destination application must call the **tran_CommReceived** function to provide the data to its Transaction Service.

Figure 6 on page 44 illustrates the interactions of client-server applications using asynchronous communications that are described in the following steps. Arrows denote the call to and return from a procedure. Operations that may take place several times are denoted with an asterisk.

1 The application makes a call to TRAN. Any call to TRAN can generate asynchronous communication.

2 TRAN calls the comm-send upcall function provided by the communications service in **tran_CommInit**. The upcall contains one or more addresses (an application identifier) and a message to be delivered to each addressee.

3 The communications service translates the application identifier into an appropriate communication address. The addressee is usually an application that has been contacted before using the transactional remote procedure call functions (**tran_CommSendingRequest**, **tran_CommReceivedRequest**, **tran_CommSendingReply**, and **tran_CommReceivedReply**).

4* The communications service transmits the messages. The communications service at each addressee that receives a message is expected to call **tran_CommReceived**.

5=2 The communications service returns from the upcall. The transmission need not be reliable; no acknowledgment is required. The upcall function can queue messages for transmission and return before they are actually delivered.

6=1 TRAN returns from the application call when the communications needs of TRAN have been met.

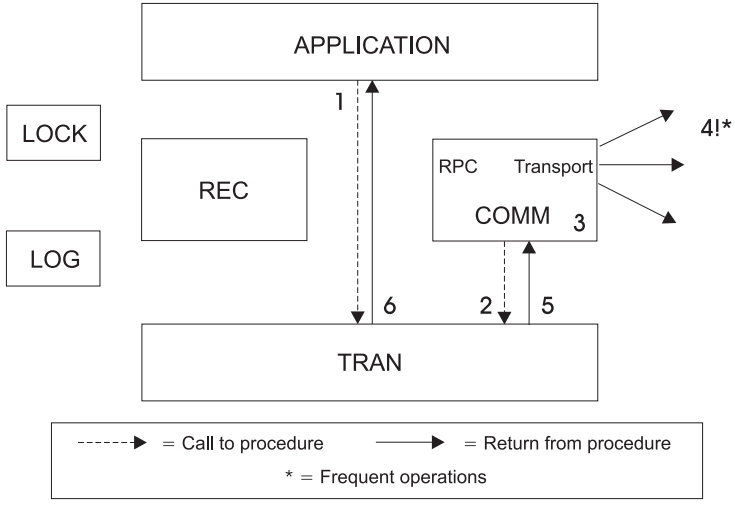

Figure 6. Asynchronous communication example

TRAN provides several optional functions that are part of the communications service interface. A communications service can call the **tran_CommBlockFunctions** function to supply special functions that TRAN can call to wait for communications service functions to complete. If the

communications service does not call this function, TRAN uses the default blocking functions registered as part of the environment (see "Application environment specification" on page 53).

A communications service can call the **tran_CommServicePromisesToMatchReplies** function during its initialization to inform TRAN that it can provide the transaction identifier to the call that completes a remote invocation. Normally, the **tran_CommReceivedReply** function returns the transaction identifier to which a remote invocation applies. Many communications services match requests and replies such that they know the transaction identifier. If the communications service can provide the transaction identifier in the **tran_CommReceivedReply** call (instead of it being returned as a result), TRAN can simplify the state information it needs to transmit. Calling **tran_CommServicePromisesToMatchReplies** enables this optimization.

A communications service can call the **tran_CommServiceAlwaysSendsReply** function during its initialization to inform TRAN that it delivers Transaction Service data on each RPC reply even when the **tran_CommSendingReply** function fails. Normally, if the transaction involved in an RPC is aborted, TRAN does not deliver Transaction Service data with the RPC reply; it delivers the data asynchronously, which can cause abort information to be delivered late and a second abort to occur. The communications service must call **tran_CommSendingReply** for every RPC, including ones for aborted transactions, to ensure that Transaction Service data is delivered on the RPC reply.

The interface also defines a pair of functions that can be used to simplify the initiation of communications in applications that do not require or cannot easily determine the identifier and address of the destination application. A communications service typically uses these functions for the first RPC to a destination, before the application identifier and address of the destination are known. TRAN does not need the identifier and address of the destination application at the time an RPC is initiated, but this information must be specified to TRAN before the RPC completes. Calling the **tran_CommSendingBlindRequest** function initiates a *blind* RPC; the **tran_CommIdentifyBlindRequest** function can be called to provide the application identifier and address to TRAN after they are known. These two functions can be used in place of the **tran_CommSendingRequest** function (normally used to initiate an RPC), which requires that the identity and address of the destination application be known before the RPC is initiated.

Note: A disadvantage to using these calls instead of the **tran_CommSendingRequest** call is that TRAN cannot contact the

server if the client aborts the transaction while the RPC is in progress. In this case, the server discovers the failure eventually but not promptly.

The **tran_CommProvideAddressInfo** function allows a communications service to provide feedback about its communication abilities. A communications service can call this function to provide additional addresses for applications or to provide an estimate of time required for message delivery using a particular address.

Recovery Service interface

A recovery service is a module within an application that manages recoverable storage. A recovery service provides an interface to the remainder of the application for modifying permanent storage transactionally; this interface is not important to the Transaction Service (TRAN). A recovery service also provides an interface to TRAN for commit processing. TRAN uses this interface to direct a recovery service to prepare to commit a transaction, to commit or abort work associated with a transaction, to forget about a transaction, and to perform logging for TRAN. TRAN provides an interface that the recovery service must use during application restart to allow TRAN to resume commit processing for existing transactions. Finally, TRAN provides features to aid a sophisticated recovery service in minimizing logging.

Each recovery service must register its interface to TRAN at initialization time. A recovery service calls **tran_RecInit** to provide the upcall functions in this interface. TRAN returns a service identifier that it uses when invoking an upcall function and that the recovery service must use when invoking Transaction Service functions. A recovery service can call **tran_RecBlockFunctions** to supply special functions that TRAN can call to wait for recovery service functions to complete; otherwise, TRAN uses the default blocking functions registered as part of the environment (see "Application environment specification" on page 53).

The recovery service interface includes functions for participating in transaction commitment and for logging Transaction Service data. TRAN invokes the recovery-active upcall when a transaction becomes active in the application. TRAN invokes the recovery-prepare upcall to ask the recovery service to make preparations to accept an outcome (commit or abort) for a transaction family. It invokes the recovery-commit or recovery-abort upcall to inform the recovery service of the outcome for a transaction. TRAN invokes the recovery-write upcall to ask the recovery service to permanently record data that will be needed after an application failure. The recovery service acknowledges that the data has been recorded. When TRAN no longer needs this data, it invokes the recovery-finished upcall to ask the recovery service to discard it. Figure 7 on page 47, Figure 8 on page 48, and Figure 9 on page 49

depict the upcalls delivered during simple transaction commitment. In these figures, arrows denote the call to and return from a procedure. Operations that may take place several times are denoted with an asterisk.

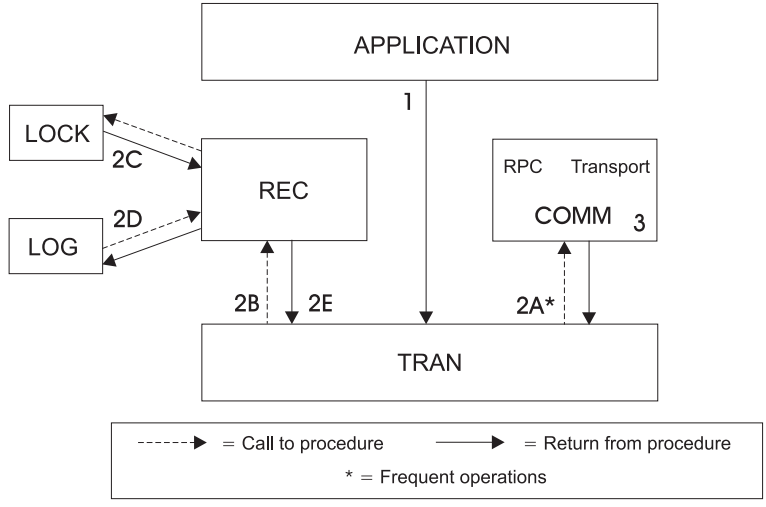

Figure 7. Commitment protocol: prepare

The prepare phase is described in the following steps:

1 The application calls **tran_End** to end the (top-level) transaction. TRAN begins the commitment protocol.

2 Each participant is asked to *prepare*:

2A* TRAN calls the communications service to deliver messages to other participants (see Figure 6 on page 44). These messages ask the other participants to prepare. TRAN waits for the communications service to deliver reply messages (indicating that those other participants are prepared).

2B TRAN asks the recovery service to prepare. The recovery-prepare upcall includes some Transaction Service data that the recovery service should include in its own prepare log record.

2C The recovery service calls the lock service to acquire lock information to include in the prepare log record. The recovery service must be able to reacquire locks for prepared transactions when it restarts after a system failure. The lock service may be an integral component of the recovery service, or it may be an independent application component.

2D The recovery service calls the log service to permanently record its state information and TRAN data. The log service may be an integral component of the recovery service, or it may be an independent application component.

2E=2B The recovery service returns from the recovery-prepare upcall.

3 – 4 TRAN delivers an outcome (see Figure 8) and a finished indication (see Figure 9 on page 49) to each participant.

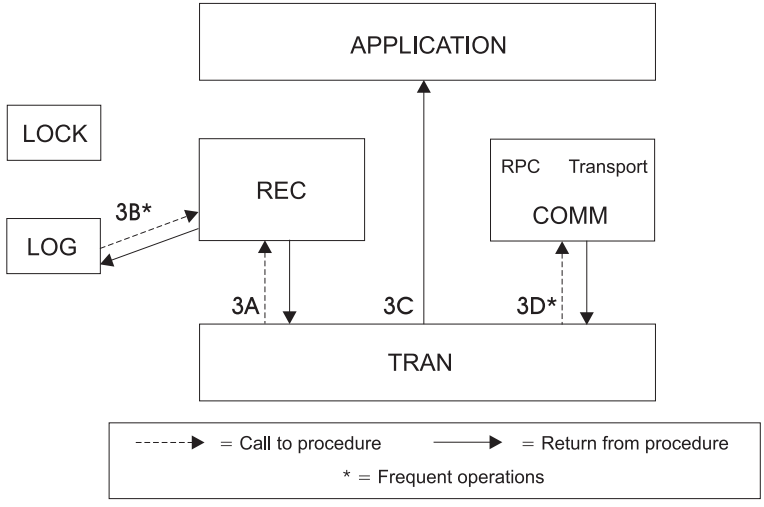

Figure 8. Commitment protocol: outcome

The outcome phase is described by the following steps:

1 – 2 TRAN has begun the commitment protocol, and each participant has prepared (see Figure 7 on page 47).

3 Each participant is informed of the *outcome*.

3A TRAN calls the recovery-commit upcall to indicate that the transaction has committed or the recovery-abort upcall to indicate that the transaction has aborted. The recovery service may release any of its resources associated with the transaction, but it must retain TRAN log data until TRAN calls the recovery-finished upcall.

3B* The recovery service calls the Log Service to write a commit indication. This indication need not be forced to permanent storage.

3C=1 The **tran_End** call returns. Other participants may not have received an outcome yet; however, they all are guaranteed to receive the same outcome.

3D* TRAN calls the communications service to deliver outcome messages to other participants. It does not wait for acknowledgments for these messages.

4 TRAN tells all participants to forget about the transaction (see Figure 9 on page 49). This part of the commitment protocol proceeds asynchronously.

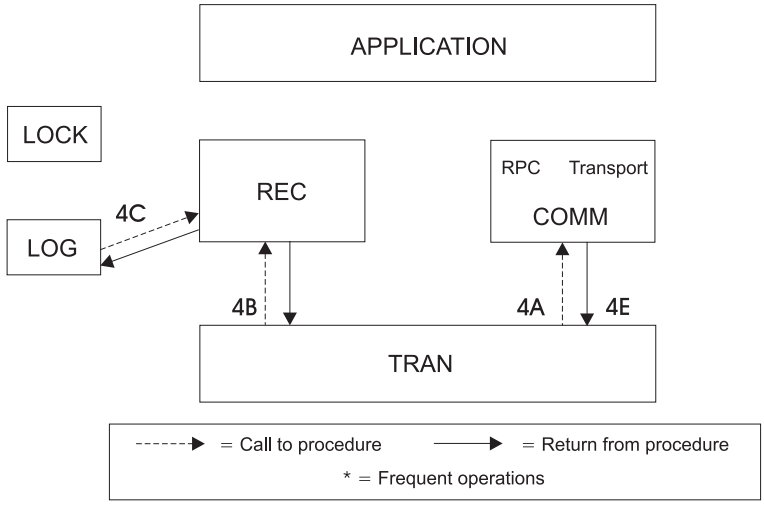

Figure 9. Commitment protocol: finished

The finished stage is described by the following steps.

1 – 3 TRAN has begun the commitment protocol and has arrived at an outcome (see Figures Figure 7 on page 47 and Figure 8 on page 48). The local recovery service has been notified of the outcome. Messages have been queued to other participants, but responses have not been received. The recovery service still must retain the last Transaction Service log data for the transaction.

4 Each participant is informed that the transaction is *finished*.

4A The communications service calls **tran_CommReceived** when it receives a message from another participant in the transaction. This message contains an acknowledgment of the transaction outcome. When TRAN receives the last acknowledgment, it can complete the commitment processing.

4B TRAN delivers the recovery-finished upcall to indicate that the recovery service should forget about the transaction.

4C The recovery service logs some indication that it should not replay any log records for this transaction. This indication may be a separate log record, or it may be collected into another log record (such as a checkpoint record), or, it may erase records for the transaction if it can.

4D=4A The **tran_CommReceived** call returns.

If the recovery service cannot prepare, it must call the **tran_RecRefuse** function. If it does not need to prepare, it must call the **tran_RecReadOnly** function.

Recovery service upcalls that take a log record as a parameter do not destroy those log records automatically. When the application no longer needs the specified log record, it must then destroy that record using the destructor function **tran_LogRecordDestroy**, as discussed in "Object destruction functions" on page 31.

TRAN provides an interface for the recovery service to replay log data when the application restarts. The recovery service calls **tran_RecReplay** to return the data it recorded on behalf of TRAN. This data restores the correlation between local transaction identifiers and the distributed transactions, and allows TRAN to resume commit processing for those transactions. After restart is complete, TRAN issues the appropriate recovery service upcalls to deliver transactions outcomes and forget transactions.

Restart

The recovery service must replay Transaction Service log records each time it restarts. These log records permit TRAN to reestablish the state of prepared transactions. TRAN resumes commit processing on each transaction for which a log record is replayed, and delivers an outcome as soon as one can be determined.

The recovery service must replay the most recently written Transaction Service log record for each *eligible* transaction identifier. A transaction identifier becomes eligible when any Transaction Service log record is written on its behalf, and the TID remains eligible until TRAN includes it in a call to the recovery-finished upcall. The special transaction identifier value `TRAN_TID_NULL` is always eligible; TRAN uses this identifier to write a log record that contains state pertinent to all transactions. The recovery service may replay TRAN log records in any order.

The recovery service may replay other Transaction Service log records under certain circumstances:

- Any log record for an eligible transaction identifier may be replayed in addition to the most recently written log record.
- Log records for a transaction for which the recovery service has completed a recovery-finished upcall may be replayed provided that an empty log record is also replayed for that transaction.
- Log records for other ineligible transactions may be replayed provided that its most recently written log record has been replayed in each complete restart since it was written. A complete restart is one in which the application calls the **tran_Ready** function.
- An empty log record may be replayed for an ineligible transaction to prevent its transaction identifier from being reused. An empty record must not be replayed for an eligible transaction, regardless of what the recovery service knows about that transaction.

A recovery service should also replay commit and abort indications found in its log for eligible transactions. These indications should be logged by the recovery-commit and recovery-abort upcalls. The commit and abort indications should be replayed as the special log record constants `TRAN_LOG_RECORD_COMMIT` and `TRAN_LOG_RECORD_ABORT`, respectively, regardless of how the recovery service actually logs them.

TRAN continues commit processing and will deliver the appropriate upcalls for each transaction for which a record was replayed. An outcome will be delivered for each transaction that was not already finished. A finished indication will be delivered for every transaction.

Each Transaction Service log record must be replayed using a separate call to **tran_RecReplay**. The recovery service may write more than one Transaction Service log record together for efficiency, but it must return them individually.

Optimizations

TRAN provides a variety of interface features that recovery services can use to achieve better performance. The recovery service upcalls that involve logging include a parameter containing optimization information. Additional interfaces are provided to allow a recovery service to complete upcalls asynchronously, to help recovery services make use of log services shared among several applications, and to allow a recovery service to register upcalls dynamically.

Optimization parameter

TRAN provides optimization information in each recovery service upcall. The information is passed by reference in a parameter of type **tran_recOptimization_t**.

In addition, TRAN defines two global variables for use with recovery service optimizations. The *tran_recOptimizationsSupported* variable specifies the optimization fields that are supported. Fields with a nonzero value indicate those optimizations that are supported. The *tran_recOptimizationsSize* variable specifies the size of this structure. These variables are defined as follows:

```
tran_recOptimization_t tran_recOptimizationsSupported;
unsigned long tran_recOptimizationsSize;
```

Asynchronous upcall completion

If recovery service logging upcalls can not complete, instead of waiting, the recovery service can elect to complete the logging upcalls asynchronously. After calling **tran_RecInit** (but before **tran_Ready**), the recovery service can call **tran_RecExplicitlyAcknowledges** to specify which of its upcalls will return before the associated work is completed. The only requirement is that the specified upcalls must eventually be acknowledged by a call to **tran_RecAcknowledge**. Acknowledgment may take place after the upcall has returned, or even in another thread.

Sharing log services

Recovery services that use the same physical log can improve performance by doing only one log force for each transaction. They must agree on a common name for each *force group*, or set of recovery service logs that are forced as a single unit. This name, the *force group identifier*, must be globally, permanently unique to the force group. The force group identifier is not interpreted or modified by TRAN; it is passed unchanged to the other participants.

Each recovery service can call **tran_RecUsingForceGroup** during a transaction to declare the force group that it is using. It can make this call more than once to declare that it is using multiple force groups for the same transaction. TRAN uses the force group declarations made during a transaction to designate one or more recovery services to force each group. It then stages upcalls among the recovery services that use a given group to guarantee that the group is forced when necessary.

Each recovery service can call **tran_RecMustForceGroup** during the recovery-prepare upcall to find out if it has been designated to force a particular group. If it is not designated, then it must not return from a log write upcall before ensuring that its log records will be made permanent if another application forces records to the same force group.

Dynamic recovery service registration

TRAN allows a recovery service to register upcalls dynamically for each transaction. Dynamic registration enables TRAN to avoid making unnecessary upcalls, eliminating communication or logging for applications that do not actively modify data on behalf of the transaction.

During initialization of a recovery service the prepare, outcome, and finished upcalls can be specified as dynamic. A recovery service can call the **tran_RecDynamicallyRegisters** function to specify that all or some of these upcalls will be requested dynamically for each transaction. The upcalls themselves cannot be changed; the **tran_RecDynamicallyRegisters** function only changes whether they are delivered automatically (the default) or on request. The **tran_RecDynamicallyRegisters** function must be called after the **tran_RecInit** function and before the **tran_Ready** function.

During a transaction the recovery service calls the **tran_RecRegister** function to specify which upcalls are needed for that transaction. An upcall that has been specified as dynamic must be registered using this function before doing work that requires the upcall. An optimal recovery service only registers for those upcalls it actually needs. A recovery service should register dynamic upcalls as follows:

- Register for a prepare upcall only when the recovery service does recoverable (not read-only) work.

- Register for a outcome upcall only when the recovery service becomes involved in a transaction (read-only or not).
- Register for a finished upcall only when the recovery service needs to release resources acquired in the other upcalls.

Typically, all the upcalls are specified as dynamic, and then one or more is requested as needed for each transaction. All upcalls are always delivered for replayed transactions.

Heuristic outcomes

TRAN provides functions to force heuristic transaction outcomes and to record and report heuristic decisions taken by other participants. Once an application has completed the prepare phase for a transaction, it is no longer allowed to abort that transaction. The application must hold locks and other resources in order to abide by the result of the commitment protocol. An application can call **tran_ForceHeuristicOutcome** to ask TRAN to deliver a specific *heuristic outcome* immediately, breaking its promise to abide by the distributed outcome; it is expected that this call would be invoked as the result of administrator interaction, not normal program control. An application module that interacts with a foreign system can call **tran_RecordHeuristicOutcome** to indicate that the other system has taken such a heuristic decision; this merely facilitates reporting and does not cause the local application to receive a heuristic outcome itself. An application can call **tran_CallOnHeuristicDamage**, **tran_DeclareReportableHeuristicDecisions**, and **tran_RequireHeuristicDamageReporting** to indicate how heuristic decisions should be reported to this and other applications. An application can call **tran_ForciblyFinish** to ask TRAN to finish a transaction that has been resolved, abdicating responsibility for providing outcome information to other applications.

Application environment specification

The Transaction Service (TRAN) depends on several basic *environment* services:

- *Identification*. TRAN needs to acquire a new application identifier the first time an application executes.
- *Synchronization*. TRAN permits the application to use multiple threads of control within its address space. TRAN must coordinate access to shared data structures when more than one application thread invokes an interface function.
- *Scheduling*. TRAN may need to prevent an interface function from returning until some other event (for example, logging or communication) occurs.

- *Memory management*. TRAN needs to allocate memory for its data structures. This memory must be accessible whenever an application thread invokes an interface function.
- *Time*. TRAN needs time information so that it can periodically retransmit messages or take alternative steps to complete transactions.
- *Termination*. TRAN may detect unrecoverable application errors. Examples of unrecoverable errors include memory corruption and improper behavior of application-provided upcall functions. TRAN attempts to report these errors and terminate without further corruption.

The application must register functions to implement these services before it initializes any of TRAN interfaces.

Environment registration

An application can use the **tran_SpecialEnvironment** function to establish its environment functions. The same function may be used repeatedly to selectively replace environment functions. The **tran_StandardEnvironment** function (see "Initialization and termination" on page 32) itself uses **tran_SpecialEnvironment** to establish the Toolkit standard environment.

Initialization and termination upcalls

TRAN makes environment upcalls at initialization time and in the event of termination. The initialize upcall is invoked before any other environment functions. The terminate upcall is invoked when TRAN is terminated, either normally or because an irrecoverable condition is encountered.

Application identifier generation upcall

TRAN needs a new, permanently, globally unique identifier for an application the first time it executes. TRAN invokes the create-application-identifier upcall to ask the application to generate a new identifier.

Synchronization upcalls

The environment must implement a synchronization mechanism to permit TRAN to coordinate access to shared data. TRAN defines a mutual exclusion variable data type, **tran_mutex_t**, and requires that the environment provide functions to initialize, lock, unlock, query, and terminate objects of this type.

Scheduling upcalls

TRAN may need to prevent an interface function from returning until some other event occurs. For example, the **tran_End** function cannot return until the local recovery services have prepared, and until all other participants have reported that they are prepared. TRAN invokes a block upcall to give the application an opportunity to trigger these events; it later invokes a wake-up upcall to indicate that the associated block function can return. The environment contains a default set of block and wake-up functions; a communications or recovery service may register its own set of functions.

TRAN invokes a block upcall when it cannot return from an interface call until some application module takes other action. The appropriate action might be an interface call: **tran_CommReceived**, to deliver a message from another TRAN; **tran_RecAcknowledge**, to report that a log record has been made permanent; or, **tran_Alarm**, to report that an alarm interval has expired. The appropriate action might be returning from an upcall or callback.

The block upcall should take appropriate steps to trigger the action that will allow the original interface function to return. If the application is a multi-threaded program in which the actions take place in other threads, the upcall can merely block the thread that is executing so that others may run. If the application is a single-threaded program, the upcall may need to take explicit action (such as attempting to receive a message, forcing a log record to permanent storage, or enabling asynchronous time notification).

The block upcall should not return until a matching wake-up upcall takes place. The block upcall includes a *block identifier* parameter that, along with the identifier of the transaction that is being blocked, identifies the particular invocation of the upcall. The wake-up upcall refers to the particular invocation that can return. The wake-up upcall may take place before the corresponding block upcall invocation in a multithreaded application program; in this case, the block upcall should return immediately. The block upcall also has a time interval parameter; if the upcall implementation would interfere with the alarm upcall implementation, it should call **tran_Alarm** when the specified time has elapsed.

The default set of block and wake-up upcall functions is used when TRAN must wait for the completion of a callback or an upcall that is not part of the communications or recovery service interface.

A communications or recovery service may specify its own set of block and wake-up functions. TRAN calls the block function associated with a particular service when an interface call must await work that must be done by that service. The service may provide its own block function so that it can take appropriate action: a communications service may need to explicitly try to receive TRAN messages from the communications service in another application; a recovery service may opt to delay forcing a TRAN log record to permanent storage until a block upcall takes place. The **tran_CommBlockFunctions** and **tran_RecBlockFunctions** functions permit services to register their own upcalls. If a service does not register its own upcalls, TRAN will use the default upcalls (provided as part of the environment) instead.

Memory allocation upcalls

TRAN needs to allocate memory dynamically for its own use and to return variable-sized data to the application. TRAN calls the allocate-memory and

free-memory upcalls to allocate and relinquish memory. These functions have the same form as the standard C library **malloc** and **free** functions; applications that do not have special memory allocation requirements may use these functions. The memory-allocation upcalls must not call any other TRAN interface functions or become blocked awaiting TRAN calls made by other threads.

Time upcalls

TRAN requires time information in order to retransmit messages or take other action to complete a transaction. TRAN invokes the current-time upcall to acquire the current time, specified as a number of seconds and microseconds since some origin time. TRAN calls the alarm-set upcall to request a notification after a given interval, specified as a number of microseconds; the environment alarm implementation must call **tran_Alarm** when the interval elapses. The level of precision needed is on the order of ten round-trip network messages. If the implementation supplies even less precision it will degrade performance but not affect correctness.

Properties

The following table lists the properties defined by TRAN. The first column of the following table lists the name of the property key constants exported in the interface declaration file. These constants may be used in calls to **tran_PropertyRetrieve**. The second column describes the meaning of the value(s) associated with the property key.

Table 3. TRAN property keys

Property key	Property value description
TRAN_PROPERTY_KEY_ABORT_DATA	The abort data describes the reason that TRAN aborted a transaction. This data can only be interpreted using **tran_AbortDataToReason**. This property is recorded only when TRAN aborts a transaction.
TRAN_PROPERTY_KEY_ABORT_FORMAT	The abort format describes another property key in which abort data is stored. A module that aborts a transaction should encode its reason for aborting in a property of its own, and record the key for that property (as a property value) in the abort format property. When TRAN aborts a transaction, it stores TRAN_PROPERTY_KEY_ABORT_DATA (converted to a property value) in this property.
TRAN_PROPERTY_KEY_ABORT_SOURCE	The abort source is the identifier of the application that called **tran_Abort** (converted to a property value).

Table 3. TRAN property keys (continued)

Property key	Property value description
TRAN_PROPERTY_KEY_GLOBAL_IDENTIFIER	The global identifier is a unique name for the transaction that is the same in all applications. The local transaction identifier (**tran_tid_t**) that is used in most TRAN interface calls is unique only within the application; other applications may have a different local identifier. The global identifier is the same in all applications, and can be used for correlation between applications. The global identifier will never be reused for another transaction (in this or in any other application, including after restarts), so it can be used to construct auditing information. Unlike the local identifier, which is fixed in size, the global identifier can be large.

Diagnostics

This section documents the diagnostic support provided by the Transaction Service (TRAN). This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal-errors) during use of TRAN, informative messages that trace calls on TRAN (tracing), and snapshots of the state of TRAN (state dumps).

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring-buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

TRAN generates a *fatal run-time error* whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility (see "Trace and state dump information" on page 58) and the application's execution is terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility

(see "Directing output" on page 57). This section defines the fatal error messages that may be output by TRAN, and briefly describes why they occur, and how they may be avoided or corrected.

TRAN provides the following fatal error messages:

- "Local transaction identifiers have been exhausted"—All local transaction identifiers allocated have been used. Restart the application.
- "cannot create alarm thread"—The underlying thread package could not create an alarm thread.
- "incorrect parameter type"—A TRAN call was made using an invalid parameter that was detected at runtime. Correct the call that uses the illegal parameter.

Warning messages

TRAN provides the following warning messages:

- "A protocol violation was detected"—Another application sent a properly formed, but illegal message at some point. The error may not be detected until a later message arrives or some other event occurs. The transaction involved in the protocol violation may need to be forcibly completed using the administrative tools.

Audit messages

TRAN does not emit any audit messages.

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes calls on TRAN. All TRAN interface functions are capable of tracing their entry, parameters, and exit.

The level of tracing is controlled by the value of a single variable that is exported by TRAN:

```
unsigned long tran_traceMask
```

The Encina Trace Facility defines a number of "bit constants" that, when set in the above variable, cause a specific form of tracing to occur. For example, the `TRACE_ENTRY` constant enables entry/exit tracing of TRAN interface functions.

"Global trace levels" describes the levels of tracing supported by TRAN.

Global trace levels

TRAN supports the global trace levels defined by the Encina Trace Facility. These levels are as follows:

- `TRACE_ENTRY` — traces the entry and exit of functions that make up TRAN interface.

- TRACE_PARAM — traces the parameters passed to the interface functions.
- TRACE_EVENT — traces all events in TRAN. This trace class outputs large amounts of information, and for this reason, is not always recommended.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- TRACE_GLOBAL
- TRACE_NONE

Transaction Service trace levels

TRAN provides no specific trace levels.

State dump

TRAN provides the **tran_DumpState** function to dump the state of TRAN in an application.

Chapter 3. Thread-to-Tid Mapping Service (ThreadTid)

Thread-to-Tid Mapping Service overview

The Encina Toolkit Thread-to-Tid Mapping Service (ThreadTid) is a library that associates transactions with threads. A transaction is associated with a thread through the use of a *transaction identifier* (TID). The Thread-to-Tid Mapping Service chapter is organized as follows:

- The remainder of this section explains basic concepts and the ThreadTid model.
- "Application interface" on page 63 describes the application interface to ThreadTid.
- "Diagnostics" on page 64 describes the diagnostic support provided by ThreadTid.

Unlike the Encina Toolkit Distributed Transaction Service (TRAN), which allows a transaction to have several threads working for it and a thread to work on several transactions, the Thread-to-Tid Mapping Service implements a more structured one-thread-to-one-transaction model. The ThreadTid model only allows a thread to work on behalf of one transaction at a time; there can, however, be multiple threads that are each using ThreadTid and are working on behalf of the same transaction. The ThreadTid module maintains a stack of transactions for each thread; the current transaction for a thread is at the top of its stack.

The ThreadTid component provides the following functionality for applications:

- *Setting or suspending the current transaction of a thread*: Transaction programming language modules, like Tran-C, make ThreadTid calls to set or suspend a thread's TID. The **threadTid_Begin** call assigns a TID to the current thread in an application and certifies it. The thread may be explicitly decertified with the **threadTid_Decertify** function.
- *Retrieving the TID of the thread's current transaction*: Any module in the application can use ThreadTid to determine the transaction currently associated with the calling thread.
- *Registering callback functions*: Any module in the application can register ThreadTid callback functions, which are invoked whenever a thread sets or suspends its TID.

When a transaction begins in a thread, **threadTid_Begin** is called to map the thread to the TID by pushing the TID on top of the thread's TID stack. At that time, the thread is certified to operate on behalf of the transaction. A thread

that is certified need not be concerned with the transaction aborting while it is working. Certification prevents the transaction service from invoking recovery service abort and application after-resolution procedures. As a result, threads should give up their certification whenever they will remain blocked for a long time.

When the transaction finishes, **threadTid_End** is called to update the thread to the TID of the previous transaction by popping the TID of the completed transaction from the top of the thread's TID stack. The thread of the completed transaction is decertified, and the previously associated thread is recertified.

A thread can temporarily leave its current transaction to do nontransactional work. A transaction is suspended by a call to **threadTid_Suspend**, which pushes a `TRAN_TID_NULL` on top of the thread's TID stack and will automatically decertify the thread. After the thread returns from the nontransactional work, a call to **threadTid_Resume** pops the `TRAN_TID_NULL` from the stack, resetting the thread to the previous TID and recertifying the thread. To retrieve a thread's current TID, ThreadTid provides a query function, **threadTid_Lookup**.

A set of certification functions including **threadTid_Decertify**, **threadTid_Certify**, and **threadTid_IsCertified**, provide increased control over certification. The functions that result in the certification of a thread return Boolean values which indicate the success or failure of the certification. Certification fails if the transaction has been committed or aborted.

The Thread-to-Tid Mapping Service also provides callback registration. All the callback functions registered in an application are called each time a thread is set to a TID. The callback functions are called in the reverse order they were registered; functions registered first are called last. Callback functions allow a thread to perform necessary cleanup work before finishing a transaction. For example, Tran-C, the application's programming language module, first calls **threadTid_Begin** to set the current thread to a TID. After that, another module participating in the application registers a callback function using **threadTid_RegisterCallback**. When the transaction finishes, the programming language module calls **threadTid_End** to reset the thread to the previous TID. When the thread's TID stack is modified, the registered callback function is called and the cleanup work is done.

CAUTION:
The DCE threading model permits the cancellation of threads. However, the Thread-to-Tid Mapping Service assumes that none of threads it has mapped to a transaction will be interrupted by such a cancellation. Thread-to-Tid Service calls should only be made with thread cancellation disabled. If a thread for which a mapping is maintained by the Thread-to-Tid Service is cancelled, the results are undefined.

Thread-to-Tid Mapping Service header files and libraries

Header files
Applications that link with ThreadTid must include the header file **threadTid/threadTid.h** in their C program.

Libraries
The Encina ThreadTid functions are contained in the **Encina** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Application interface

This section describes the application interface to ThreadTid. The application interface consists of functions to begin, end, suspend, resume and query a transaction identifier in a thread and a function to register callbacks.

Initialization

The ThreadTid component initializes itself. No explicit initialization procedure is needed.

Setting and querying a thread's current transaction

The following five functions are related to TID mapping:

- **threadTid_Lookup**
- **threadTid_Begin**
- **threadTid_End**
- **threadTid_Suspend**
- **threadTid_Resume**

Explicitly decertifying and certifying threads

The functions that explicitly decertify and certify threads can be used by applications that require control of thread certification beyond what is provided by the **threadTid_Begin**, **threadTid_End**, and **threadTid_Suspend** functions. The ThreadTid component provides the **threadTid_Decertify** function to decertify a thread from working on behalf of a transaction, the **threadTid_Certify** function to certify a thread to work on behalf of a transaction, and the **threadTid_IsCertified** function to check the certification status of a thread.

Certification prevents the transaction service from undoing the effects of a transaction while there are still threads executing on behalf of that transaction. A thread can be temporarily decertified with the **threadTid_Decertify** function or permanently decertified with the **threadTid_End** function. If there are multiple threads executing on behalf of a single transaction, recovery can only take place when all of these threads are decertified. Threads should be decertified if they block, for example, by making an RPC.

Registering callbacks

The ThreadTid component provides the **threadTid_RegisterCallback** function to register a callback. The **threadTid_event_t** data type is used to enumerate the events that can cause a registered callback to be invoked.

The ThreadTid component also provides the **threadTid_RegisterTrpcCallbacks** function to register TRPC callbacks. The reference pages for these functions describe how to register callback functions in an application and how and when the callback functions are invoked.

Diagnostics

This section documents the diagnostic support provided by the Thread-to-Tid Mapping Service. This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal-errors) during use of ThreadTid, informative messages that trace calls on ThreadTid (tracing), and snapshots of the state of ThreadTid (state dumps).

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring-buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard input stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

The Thread-to-Tid Mapping Service does not emit any fatal error messages.

Warning messages

The Thread-to-Tid Mapping Service generates a *warning run-time error message* whenever an thread-specific error occurs. Since the errors are specific to a thread, ThreadTid does not cause the application to terminate, but instead emits a warning message, and terminates the execution of the thread by waiting on a condition variable that is never signaled. The run-time error is presented as an output message through the Encina Trace Facility (see "Trace

and state dump information"). The destination of the error message can be defined by using the functions provided by the Encina Trace Facility (see "Directing output" on page 64). This section defines the warning error messages that may be output by ThreadTid, and briefly describes why they occur, and how they may be avoided or remedial action taken. The Thread-to-Tid Mapping Service has the following fatal error messages:

- "threadTid_End: No previous matching threadTid_Begin call provided."—The **threadTid_End** call was called when the previous matching call was not **threadTid_Begin** (there may have been no matching **threadTid_Begin** call or a **threadTid_Suspend** call may have been made immediately before the **threadTid_End**). This message signals a program error. You must correct the program error.
- "threadTid_Resume: No previous matching threadTid_Suspend call provided"—The **threadTid_Resume** call was called when the previous matching call was not **threadTid_Suspend** (there may have been no matching **threadTid_Suspend** call or a **threadTid_Begin** call may have been made immediately before the **threadTid_Resume**). This message signals a program error. You must correct the program error.
- "threadTid_Certify: No transaction"—The **threadTid_Certify** call was made when the thread was not executing on behalf of a transaction. This message signals a program error. You must correct the program error.
- "threadTid_Decertify: No transaction"—The **threadTid_Decertify** call was made when the thread was not executing on behalf of a transaction. This message signals a program error. You must correct the program error.

Audit messages

The Thread-to-Tid Mapping Service does not emit any audit messages.

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes calls on ThreadTid. All ThreadTid interface functions are capable of tracing their entry, parameters, and exit. In addition, important events during the execution of a ThreadTid interface function may be traced. Clients of ThreadTid can enable a specific class of event tracing, such as entry/exit, or a collection of events (entry/exit and parameter tracing).

The level of tracing is controlled by the value of a single variable that is exported by ThreadTid:

```
unsigned long threadTid_traceMask
```

The Encina Trace Facility defines a number of "bit constants" that when set in the above variable cause a specific form of tracing to occur. For example, the `TRACE_ENTRY` constant enables entry/exit tracing of Thread-to-Tid Mapping

Service interface functions. The Thread-to-Tid Mapping Service also defines ThreadTid-specific bit constants for important classes of internal event.

Global trace levels

The Thread-to-Tid Mapping Service supports the global trace levels defined by the Encina Trace Facility. These levels are as follows:

- `TRACE_ENTRY` — traces the entry and exit of functions that make up ThreadTid interface.
- `TRACE_PARAM` — traces the parameters passed to the interface functions.
- `TRACE_EVENT` — traces all events in ThreadTid. This trace class outputs large amounts of information, and for this reason, is not always recommended.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- `TRACE_GLOBAL`
- `TRACE_NONE`

Thread-to-Tid Mapping Service trace levels

The Thread-to-Tid Mapping Service provides no specific trace levels.

State dump

The Thread-to-Tid Mapping Service provides the **threadTid_DumpState** function to dump the state of a current thread executing an application.

Chapter 4. Transactional RPC Service (TRPC)

TRPC overview

The Encina Transactional Remote Procedure Call (TRPC) Service enhances the Distributed Computing Environment (DCE) Remote Procedure Call (RPC) package offered by the Open Software Foundation (OSF). DCE RPC is a remote procedure call (RPC) system that implements nontransactional RPCs for use by Encina Toolkit components.

TRPC provides the same basic interface as DCE RPC. However, there is one significant difference. TRPC implements transactional and nontransactional RPCs; DCE RPC implements nontransactional RPCs only. To accommodate the transactional nature of TRPC, the Transactional Interface Definition Language (TIDL) was developed. TIDL is described in the *Encina Transactional Programming Guide*.

The Transactional RPC Service chapter is organized as follows:

- The remainder of this section explains the basic concepts behind DCE RPC and TRPC. Also described are the components, terminology, and models of DCE RPC and TRPC. Read this section before using the TRPC calls.
- "Data types" on page 80 describes the data types used by TRPC.
- "Application interface" on page 81 describes the TRPC functions and callbacks.
- "Functions for manipulating binding handles" on page 84 describes the TRPC functions that parallel the DCE RPC functions for manipulating binding handles. These "wrapper" functions incorporate the transactional aspect of TRPC and are used to create, destroy, and modify transactional handles.
- "Diagnostics" on page 85 describes the diagnostic support provided by TRPC.

TRPC and the Toolkit

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

CAUTION:
The DCE threading model permits the cancellation of threads. However, the TRPC Service assumes that its work will not be interrupted by such a cancellation. TRPCs should only be made from threads with thread cancellation disabled. If a thread executing a TRPC is cancelled, the results of that function are undefined.

Figure 10 depicts the interaction between the application, the Transaction Service (TRAN), TRPC, DCE RPC, and the threads package. The application interacts with TRPC via the TIDL preprocessor (**tidl**) and the TRPC run-time interface. TRPC interacts with TRAN by calling TRAN run-time library functions. TRAN invokes TRPC run-time library functions through a system of upcalls which are functions registered with TRAN by TRPC. The application also interacts with TRAN, DCE RPC and the threads package directly.

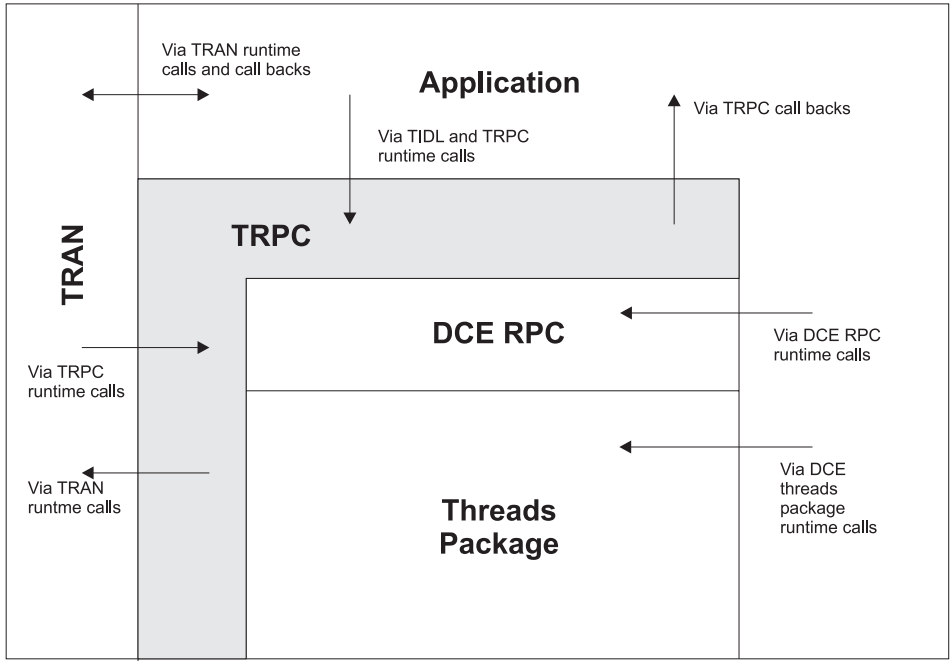

Figure 10. TRPC architecture

Remote procedure calls (RPCs)

A *remote procedure call* (RPC) is a programming paradigm similar to the well-known procedure call mechanism. Both procedure call mechanisms transfer control and data within a program.

When a remote procedure is called, the parameters of the call are passed over the network to the environment where the call is actually executed. Meanwhile, the calling environment waits for the results of the procedure execution. Typically the calling environment is a program that is referred to as a *client*. The environment where the call is executed is referred to as a *server*. When the server (the called environment) finishes executing the procedure, it ships the results back to the client (the calling environment) which then resumes execution as if returning from a local procedure call.

Transactional RPCs

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

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

Communication support for TRAN

TRPC provides communication support for the Distributed Transaction Service (TRAN). TRPC piggybacks TRAN data on user-initiated RPCs (this is also known as synchronous communication) which have been preprocessed by TIDL. TRPC also provides a set of library functions that TRAN uses to do asynchronous communication, where TRAN data is not piggybacked on RPCs. Messages sent by TRAN by this special asynchronous communication mechanism are called out-of-band messages.

Since it is important to have an overview of DCE RPC in order to clearly understand how TRPC is layered on it. DCE RPC is described first in the next section, TRPC in the following one.

DCE RPC

DCE RPC components

DCE RPC consists of the following two components:

- **Interface Definition Language** (IDL) is used loosely to describe both the interface definition language and the compiler that accepts a file (the file extension is **.idl**) containing operation descriptions (functions implemented by remote procedures). An operation description includes the types of parameters, and the return values of operations. IDL produces C files that contain code to marshal and unmarshal operation parameters to and from the request and response messages. The output C files are appropriately linked with the server and the client programs.
- **DCE RPC run-time library** is the library exported by the DCE RPC system. The DCE RPC run-time library uses the Concert Multi-thread Architecture (CMA) exception model to raise and catch exceptions. For further information, see the *OSF DCE Application Development Guide*.

DCE RPC terminology

The following DCE RPC terms are useful when describing how clients and servers are built in the DCE RPC and TRPC environments. Refer to the *OSF DCE Application Development Guide* for further details.

- *Interface definition files* are produced by the server writer and contain declarations of remote procedures and their parameters and return values. IDL accepts such a file and produces code to marshal and unmarshal parameters and do the necessary send and receive operations.
- *Manager functions* are provided by the server writer to implement the actual operations of the remote procedure. They are linked with the server code. This document also refers to them as *native manager functions*.
- *Client stubs* are produced by IDL and are linked with the client code. These functions marshal the *in* parameters of a remote procedure call and send the request. When a reply is received, the client stubs unmarshal the *out* parameters. This document also refers to them as *native client stubs*.
- *Client switch functions* are produced by IDL and provide an entry point for the remote procedures invoked by an application. They in turn call the corresponding client stubs.
- *Server stubs* are produced by IDL and are linked with the server code. They unmarshal the *in* parameters and call the user-provided manager functions. When the manager function returns, the server stubs marshal the *out* parameters and send the reply. This document also refers to them as *native server stubs*.
- *Entry point vector* (EPV) is an initialized structure of function pointers. The corresponding functions are invoked by passing execution control to the address in the fields of this structure. There are different types of entry point vectors used by DCE RPC:
  - *Server stub entry point vector* is the vector used by the RPC run time to call the appropriate server stub routine when an RPC arrives. It is initialized in the server stub file.
  - *Client stub entry point vector* is the vector of client stub procedures. It is used by the run time to call the appropriate client stub when the application initiates an RPC. It is initialized in the client stub file.
  - *Manager entry point vector* is the vector of manager functions. It is used by the run time to invoke the appropriate manager function from within the server stub. It is provided by the server application.

DCE RPC model

DCE RPC is a remote procedure call system. The server writer describes each operation or remote procedure in an interface definition file. IDL accepts the interface definition file and produces client and server stub files. The stub files contain calls to functions in the DCE RPC run-time library that marshal and unmarshal the parameters of the RPC and send and receive messages. The stub files are appropriately linked with the client and server programs.

When a client initiates an RPC, it actually calls the client stub function generated by IDL. The client stub marshals the *in* parameter and sends the request message. The client stub then waits for the reply message to return. On the server side, when the DCE RPC run time receives the request message, it calls the server stub generated by IDL. The server stub unmarshals the *in* parameter and then calls the server manager function, provided by the server writer for the corresponding RPC. When the manager function returns, the server stub marshals the *out* parameter and sends a reply message to the client. The client DCE RPC run time receives the reply message and passes it to the client stub. The client stub unmarshals the *out* parameter and returns control back to the application. The entire operation provides the semantics of a local procedure. Figure 11 illustrates the flow of a DCE RPC.

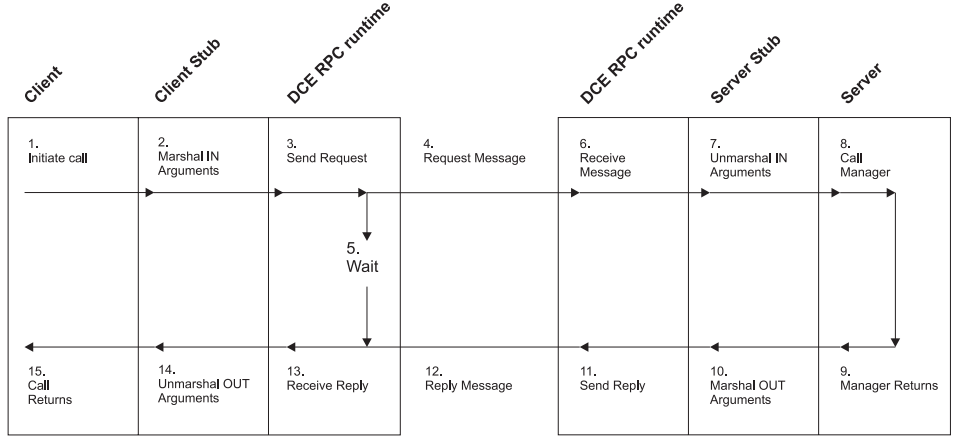

Figure 11. Flow of an RPC

The following is a description of the steps shown in Figure 11:

1. The client invokes the remote call.
2. The client stub marshals the *in* parameters and passes them to the DCE RPC run time.
3. The DCE RPC run time sends the request.
4. The request message goes over some communication channel to the server.
5. The DCE RPC run time at the client waits for the reply message.
6. The DCE RPC run time at the server receives the request message and passes it to the server stub.
7. The server stub unmarshals the request message.
8. The manager function is called by the server stub.
9. The manager function returns.

10. The server stub marshals the *out* parameters and passes them to the DCE RPC run time.
11. The DCE RPC run time sends the reply message.
12. The reply message goes over some communication channel.
13. The DCE RPC run time at the client receives the reply message and passes it to the client stub.
14. The client stub unmarshals the *out* parameters.
15. The remote procedure returns to the client.

TRPC

TRPC components

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

The library of functions provide the relevant communication support for the Distributed Transaction Service (TRAN). TRPC also exports interface calls to the application developer. The functions can be categorized into sets as follows:

- Functions to support the TRAN communication interface. The TRAN communication interface is described in "Chapter 2. Transaction Service (TRAN)" on page 11. This aspect of TRPC is not discussed in this document.
- Functions used to initialize the TRPC run time interface, provide information about communication protocols and endpoints, and register callbacks (see "Application interface" on page 81).
- Functions that wrap certain DCE RPC run time functions that manipulate RPC handles (see "Functions for manipulating binding handles" on page 84).

TRPC model for transactional communication

TRPC provides the TIDL preprocessor to support transactional RPCs and provides functions that implement the TRAN asynchronous communication interface. Together, they comprise the model used by TRPC to provide the required communication support to TRAN.

Transactional RPCs: To implement transactional RPCs, the **tidl** command accepts an interface definition file and produces the following:

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

IDL interface definition file. IDL accepts the modified interface definition file and produces the native client and server stubs.

- *Shadow client stubs* — These stubs are invoked when the client initiates an RPC. The shadow client stubs then invoke the native client stubs produced by IDL.
- *Shadow manager functions* — These functions are invoked when the RPC run time invokes manager functions within the server stubs. The shadow manager functions eventually call the user provided manager functions.

The shadow client stubs provide entry points for remote procedures initiated by the client. Each shadow client stub first calls a function called **tran_CommSendingRequest** to get transaction service data for the remote TRAN. It then invokes the native client stub which marshals the *in* parameter of the RPC as well as the transaction service data. The native client stub sends the request message and waits for the response. On receiving the response message, the native client stub unmarshals the *out* parameter which now include a parameter for the received transaction service data. The shadow client stub then invokes another function called **tran_CommReceivedReply** to pass the received transaction service data to the local TRAN.

The shadow manager functions are invoked by the native server stubs. Each shadow manager function must guarantee its local TRAN the authenticity of the *application identifier* (see "Application identifier" on page 78) in the received request. The shadow manager function calls a TRAN function called **tran_CommReceivedRequest** to pass the transaction service data in the request message to its local TRAN. After calling **tran_CommReceivedRequest** the shadow manager function calls the native manager function and then calls a function called **tran_CommSendingReply** to request its local TRAN for data for the client's TRAN.

Figure 12 on page 77 illustrates the flow of calls and messages in a transactional RPC.

Transactional asynchronous communication: TRAN also requires TRPC to provide a mechanism for asynchronous transaction communication. Asynchronous transaction communication implies that TRAN data is not piggybacked on application RPCs. Instead, TRAN expects the communications service (for example, TRPC) to deliver the data using separate messages. TRPC provides this mechanism via a library of functions. Some functions in this library provide a way to do special-purpose RPCs to allow asynchronous transaction communication. TRAN messages delivered asynchronously are also referred to as out-of-band TRAN messages in this document. TRPC does the authentication for out-of-band TRAN messages.

Limitations on transaction state data: The TRPC run-time library imposes limitations on the amount of TRAN state information that can be transmitted

("piggybacked") on client RPCs. These limitations are imposed in the interests of normal-case performance, where the amount of state data that must be transmitted is expected to be small. Unfortunately, this may indirectly limit the complexity of transactions.

The following factors contribute to the amount of TRAN state data that must be transmitted with an RPC:

- The number of participants in the transaction family
- The depth to which a transaction is nested within other transactions
- The number of nested transactions in the same transaction family (and the depths to which they are nested)
- The number of applications that have caused heuristic damage

When the amount of TRAN data exceeds the limit imposed by the TRPC run-time library, transactional RPCs will fail, and the encompassing transaction will abort. Depending on the point at which this limit is actually exceeded, an error message may be displayed, stating "fault invalid bound (dce / rpc)".

To approximate transaction complexity, add the number of instances of each limiting item from the previous list. Encina systems should support all transactions with a complexity of ten or less, and most transactions with a complexity of twenty or less.

Advantages for transaction programming environments

In addition to providing transactional RPCs, TRPC also provides functionality for transaction programming environments like Transactional-C. For transactional programming environments, TRPC supports the following:

- Piggybacking data on an RPC
- Handling exceptions during the course of an RPC
- Aborting an RPC and the enclosed transaction
- Invoking functions at specific places in an RPC sequence

TRPC provides a general callback mechanism (see "Callbacks") for the last item.

Callbacks

Callbacks are procedures that are registered with TRPC by transaction programming environments (for example, Transactional-C) and the application. They are invoked by TRPC when specific events occur. TRPC provides callbacks to enable different transaction programming components of an application to piggyback data on RPCs, set up data structures and clean data structures before and after the RPC in both the server and the client, and to provide an opportunity to the application to do security checks. The

application developer's use of callbacks must be compatible with the use of the callbacks by the transaction programming environments. The compatibility issues are outlined in the description of each callback in "Application interface" on page 81.

The shadow client stubs and the shadow manager functions invoke callback procedures in addition to the TRAN procedures. TRPC exports functions to the application to register callbacks. For further information, see "Application interface" on page 81.

The following callback procedures are invoked at the same four points in the RPC sequence as the TRAN functions called by TRPC:

- Before-sending-request callbacks
- After-receiving-request callbacks
- Before-sending-reply callbacks
- After-receiving-reply callbacks

As an example, before-sending-request callbacks can be used on the client side to set up authentication and authorization information, while after-receiving-request callbacks can be used on the server side to retrieve the authentication and authorization information. Figure Figure 12 on page 77 depicts the manner in which these callbacks are invoked during a transactional RPC.

TRPC also provides other callback functions to provide transaction identifiers and handle exceptions. A transaction identifier is a TRAN abstraction which identifies a transaction in an application. The get-transaction-identifier callback is invoked to get a transaction identifier if the transaction identifier is not passed as a parameter of an RPC. This callback must be registered with TRPC by calling **trpc_CallToGetTid**. A fatal error results if the callback function is not registered.

TRPC invokes exception handling callbacks in the server and the client when it detects an exception. If an exception occurs on the server side, the shadow manager functions cause the server-side-exception callbacks to be invoked. After the exception callbacks have executed, the exception is passed to the client, causing an exception to occur in it. If an exception occurs in the client, the shadow client stubs cause the client-side-exception callbacks to be invoked. After the execution of the exception handlers, the shadow client stubs do the following:

1. If the RPC is transactional, the shadow client stubs abort the transaction.
2. Control is passed to a function registered with the shadow client stub via the TRPC function **trpc_CallOnRpcTermination**. Such a function is

provided by certain transactional run-time environments like Transactional-C (Tran-C). In the absence of Tran-C, the application can register this function.

3. If the **trpc_CallOnRpcTermination** function returns and an error parameter is specified in the interface definition file, the exception code is passed to the application via the error parameter.

The exception callbacks can be used to do any cleanup associated with the RPC. The exception callbacks, along with **trpc_TerminateRpc** and **trpc_CallOnRpcTermination** (see "Abort RPC functions" on page 83), provide a mechanism to **longjmp** out of the scope of the RPC.

Callbacks and manager functions that raise exceptions must use exception codes that have a value greater than or equal to the symbolic constant `TRPC_APPLICATION_ERROR`. The value must be carefully selected, as it must not collide with the status value used by DCE RPC. Also, arbitrary values are likely to confuse the client about the actual reasons for aborting the RPC.

TRPC model for nontransactional RPCs

TRPC provides a mechanism allowing for nontransactional RPCs. Nontransactional RPCs provide support for the callback and exception handling mechanisms described earlier. However, nontransactional RPCs do not carry any TRAN data. Nontransactional RPCs issued within the scope of a transaction (that is, when the transaction identifier obtained by the get-transaction-identifier callback is nonzero) are treated as standard function calls. If an exception occurs during their execution, the exception callbacks are still executed, and the exception code is returned in an *out* parameter or as a return value (type **error_status_t** or **trpc_status_t**) of the RPC. The transaction is not aborted automatically. For further information, see the TIDL documentation in the *Encina Transactional Programming Guide*.

Flow of a transactional RPC

During the course of a transactional RPC, TRPC invokes both the TRAN calls and the callbacks registered with TRPC in a specific order. The sequence of these calls is illustrated in Figure 12 on page 77.

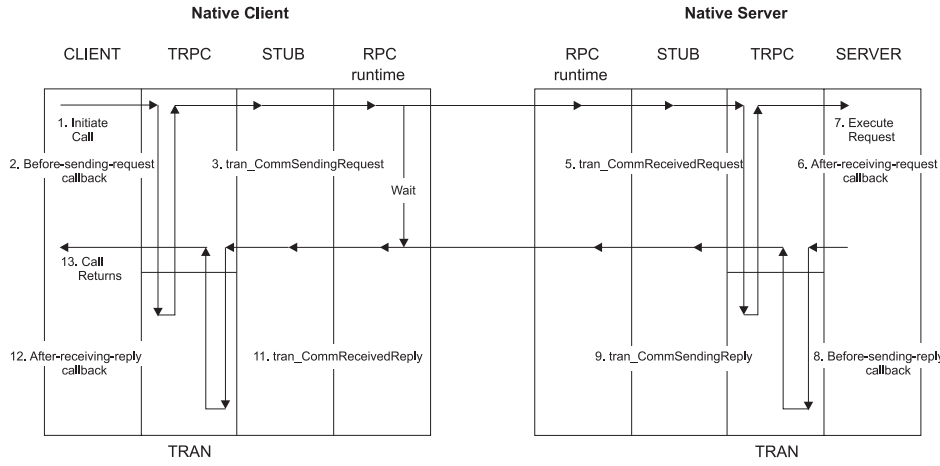

Figure 12. Flow of a transactional RPC

The following is a description of the steps in Figure 12:

1. The client initiates an RPC, causing the shadow client stub to be invoked.
2. The shadow client stub then invokes the before-sending-request callbacks. These callback procedures may return data that is piggybacked on the RPC request.
3. The shadow client stub calls local TRAN and requests transaction service data to be sent.
4. The shadow client stub invokes the native client stub that causes the request message to be delivered. The request message contains the callback data and the transaction service data, in addition to the *in* parameters of the operation specified in the TIDL file.
5. On the server side, the shadow manager function is invoked. It first authorizes the use of the application identifier in the received request and then passes any transaction service data to the local TRAN.
6. The shadow manager function then invokes the after-receiving-request callbacks. It passes to each callback the portion of the callback data meant for it.
7. The shadow manager function invokes the actual operation.
8. The shadow manager function calls the before-sending-reply callbacks. These callbacks may return data that is piggybacked on the RPC reply.
9. The shadow manager function calls local TRAN to request transaction service data that must be sent back to the client.
10. The run time delivers the RPC reply. The reply message contains callback data and transaction service data in addition to the *out* parameters of the operation specified in the TIDL file.

11. The shadow client stub receives the reply. It passes any transaction service data to the local TRAN.
12. The shadow client stub calls the after-receiving-reply callbacks. It passes to each callback the portion of the callback data meant for it.
13. The shadow client stub passes the *out* parameters of the operation specified in the TIDL file to the client and the RPC returns.

Important abstractions

This section describes certain abstractions that are used by TRPC. They include the application identifier, application addresses and the transactional handle.

Application identifier

TRAN uses an *application identifier* to uniquely name each application that participates in a transaction. The type of the application identifier (**tran_applId_t**) is exported by TRAN. During the course of a transactional RPC, TRAN requires TRPC to provide the application identifier of the remote application. Conversely, when TRAN initiates asynchronous communication, it provides TRPC with the application identifier of the remote application and an entity described by the type **tran_address_t** along with the message. If necessary, TRPC queries a directory service for the communication endpoint of the remote application using the application identifier as the key. As part of the TRPC initialization, each application also registers the application identifier and the communication endpoints with the directory service to allow the lookup mentioned earlier.

Application addresses

TRAN expects the communications subsystem to assign addresses to each application. Each *application address* is contained in the data type **tran_address_t** exported by TRAN. TRAN does not create or interpret application addresses; it merely maintains a binding between the application identifier and the application address. When TRAN initiates asynchronous transactional communication, it provides the communications subsystem with both the application identifier and the application address of the remote application. The application address can be used by the communications subsystem to locate the remote application.

There are several possible reasons why an application has multiple addresses.

- An application may support multiple communications services like TRPC and another communications service based on some other communication paradigm. Each communications service may assign a different address to itself.
- An application using TRPC may decide to use all the communication protocols supported by DCE RPC. TRPC creates a different address for each communication protocol.

- An application may decide to create multiple communication endpoints for a single communication protocol that it plans to use. TRPC creates different addresses for each communication endpoint.

Application addresses created by TRPC provide either the logical or the physical address of an application. As an example of the logical address, the application address may merely contain a cell name, allowing TRPC to contact the appropriate directory service to initiate a lookup using the application identifier as the key. On the other hand, the application address may in fact contain the actual communication endpoint of an application.

TRPC creates local application addresses as part of its initialization process. Some of the interface functions and the state information that TRAN provides enable TRPC to create the appropriate kind of application addresses. As a general rule, TRPC creates application addresses with actual physical addresses for ephemeral applications and applications that use well-known communication endpoints. Further, in the absence of a directory service, TRPC insists that recoverable applications provide well-known communication endpoints. If well-known endpoints are not provided, the application terminates with a fatal error message.

Transactional handles

DCE RPC provides an abstract data type called an *RPC handle* that describes the binding between the client and the server. TRAN uses the application identifier to identify each application and provides the application address to enable the communications subsystem to address an application. It expects the transactional communications service to deliver messages by providing application identifiers, application addresses and messages for the corresponding applications. Initially TRPC provides TRAN with the application identifier and the application address of a remote application to which a transactional RPC is initiated. Eventually TRAN passes this information back to TRPC during asynchronous transactional communication. TRPC maintains a binding between application identifiers and addresses (TRAN's way of addressing applications) with RPC handles (DCE RPC's way of addressing applications).

TRPC exports an abstraction called a *transactional handle* which maintains such a binding. The application must use transactional handles in place of the regular RPC handles. The type of the regular RPC handles is **handle_t** (DCE RPC equivalently also uses the type **rpc_binding_handle_t**), while the type of the transactional handles is **trpc_handle_t**.

When an application initiates a transactional RPC, the shadow client stubs must know the application identifier and the address of the remote application. The shadow client stub needs to pass these to the local TRAN

and the callback functions while requesting data to be piggybacked on the RPC. The following two situations may arise:

- The transactional handle does not have the application identifier and the application address. In this situation the shadow client stub tries to determine the remote application's application identifier. If it succeeds, then it initiates the transactional RPC. After the transactional RPC completes the transactional handle will contain the application identifier and the application address for future calls. If it fails, then the shadow client stub aborts the RPC and consequently the enclosing transaction.
- The transactional handle contains both the application identifier and the application address. In this situation, the shadow client stub uses the application identifier and the address in the transactional handle. This can happen if the application knows the application identifier and the address of the remote server at the beginning of execution.

TRPC caches transactional handles with application identifiers, application addresses, and bound RPC handles. The cache is used to determine the RPC handle of the remote application when TRAN initiates out-of-band communication and the corresponding application address does not contain the actual physical address of the remote application.

TRPC header files and libraries

Header files

The file **trpc/trpc.h** contains the TRPC interface declarations for the C language. This file must be included in any file that uses the TRPC interface functions or data types.

Libraries

The Encina TRPC functions are contained in the **Encina** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Data types

TRPC data types

The following are the data types defined by TRPC:

- **trpc_status_t** — return type for most TRPC functions
- **trpc_outOfBandMode_t** — enumerated type used to determine the mechanism used by TRPC to deliver out-of-band TRAN messages
- **trpc_handle_t** — handle used in transactional RPCs
- **trpc_tranInfo_t** — a structure containing the transaction identifier and application identifier of a remote application

- **trpc_ifSpec_t** — a structure containing the name of an operation, the name of the interface to which the operation belongs, and another structure that contains the version number and the UUID of the interface

TRPC also defines the **trpc_Free** function to free memory that is allocated dynamically by TRPC.

Imported DCE RPC data types

Parameters to some TRPC functions use DCE RPC data types, for example, functions that wrap certain DCE RPC functions. These functions are described in "Functions for manipulating binding handles" on page 84. TRPC imports the following DCE RPC data types:

- **rpc_protseq_vector_t**
- **rpc_binding_vector_t**
- **uuid_t**
- **rpc_binding_handle_t**
- **unsigned32**
- **signed32**
- **unsigned_char_p_t**

Application interface

The TRPC application interface provides functions for initializing and terminating TRPC, for registering callbacks, for making callbacks to get transaction identifiers, for aborting RPCs, for manipulating application addresses, for getting interface specifications, and for determining if the RPC is transactional or not.

Several DCE RPC calls are mentioned in this TRPC document. Their complete descriptions are in the *OSF DCE Application Development Reference*.

Initialization functions

There are three possible steps in initializing TRPC:

1. Specify communication protocols and well-known endpoints, if necessary. The **trpc_UseProtseqVector** function specifies information about communication protocols. The **trpc_UseWkEndpoints** function and the **trpc_BindWkEndpoints** function specify information about well-known endpoints. The **trpc_InitWithTrdce** function can specify information about both if the server has created and registered binding handles. (An application that can use all the communication protocols supported by DCE RPC, and does not use well-known endpoints, can omit this step.)
2. Set directory service and security specifications, if necessary, using the **trpc_SetEnvironment** function. If this function is not explicitly called in an Encina application, TRPC uses a set of default values for these

specifications. These values are correct for most cases, and they should only be changed if an application explicitly needs to override one or more of these defaults. See the reference page for the **trpc_SetEnvironment** function for a discussion of the default values used by TRPC.

3. Initialize the TRPC runtime interface. The **trpc_Init** function initializes basic features of the TRPC application interface. If the first part of the initialization (step 1) is omitted, this function uses default communication protocols and creates suitable communication endpoints.

If used, the **trpc_SetEnvironment** function must be called before the **trpc_Init** function, because it defines aspects of the environment used when initializing TRPC. Similarly, the **trpc_Init** function uses default communication protocols and endpoints if others are not specified by the **trpc_UseProtseqVector**, **trpc_UseWkEndpoints**, **trpc_BindWkEndpoints**, and **trpc_InitWithTrdce** functions. The **trpc_Init** function must be completed prior to calling the **tran_Ready** function.

The **trpc_GetEnvironment** function can be called after the **tran_Ready** function to retrieve the environment values registered by Encina. The **trpc_SetTranTimeout** function can be used to set a timeout for inactive transactions.

Before-sending-request callbacks

TRPC invokes a set of callback procedures in the shadow client stubs before shipping the RPC request to the server. These callback procedures are registered by calling the **trpc_CallBeforeSendingRequest** function.

After-receiving-request callbacks

TRPC invokes a set of callback procedures in the shadow manager functions after receiving an RPC request. These callback procedures are registered by calling the **trpc_CallAfterReceivingRequest** function.

Before-sending-reply callbacks

TRPC invokes a set of callback procedures in the shadow manager functions before shipping the RPC reply to the client. These callback procedures are registered by calling the **trpc_CallBeforeSendingReply** function.

After-receiving-reply callbacks

TRPC invokes a set of callback procedures in the shadow client stubs after receiving the RPC reply from the server. These callback procedures are registered by calling **trpc_CallAfterReceivingReply**.

Client-side-exception callbacks

TRPC invokes a set of callbacks in the shadow client stubs when an exception is detected in the client. These callbacks are registered by calling **trpc_CallOnClientException**.

Server-side-exception callbacks

TRPC invokes a set of callbacks in the shadow manager functions when an exception is detected in the server. These callbacks are registered by calling **trpc_CallOnServerException**.

Get-transaction-identifier callbacks

The client invokes this callback to get the transaction identifier of the transaction on whose behalf the RPC is being initiated. This callback is registered by calling **trpc_CallToGetTid**.

Callback data

The TRPC Service allows callback data to be passed from client to server. A client uses the **trpc_SendCallbackData** function to provide the data to be piggybacked on a TRPC. A server uses the **trpc_ReceiveCallbackData** function to retrieve the data that has been piggybacked on a TRPC. Both of these functions use the *callbackDataId* parameter (an unsigned long integer) to identify the module sending the piggybacked data. These identifiers must have a value less then 32768; values greater than or equal to 32768 are reserved for use by TRPC.

Abort RPC functions

Server and client applications can abort RPCs by calling the **trpc_TerminateRpc** and **trpc_CallOnRpcTermination** functions, respectively. These functions are meaningful in the shadow manager functions and the shadow client stubs and therefore can be called from within the callback functions. The **trpc_CallOnRpcTermination** is used to register the callback. The callback is invoked by TRPC on the client side when an RPC is aborted.

Application address manipulation functions

TRPC provides the **trpc_GetCompatibleLocalAddress** function to allow an application to get a local address corresponding to a given DCE RPC protocol sequence. These local addresses can be registered with directory services (or otherwise distributed) so that clients can call **trpc_ConsBinding** to produce a fully bound handle.

Server-side transaction functions

This section describes the functions used for creating and managing server-side transactions. A *server-side transaction* is a transaction that is initiated by a client application but executed at the server.

A client application can specify that a transactional RPC should be executed in a server-side transaction by associating a special transaction identifier (acquired with the **trpc_GetWrapTid** function) with the TRPC. The manager function in the server is executed within a transaction begun and ended by the TRPC runtime in the server. Nontransactional RPCs are not eligible for execution in server-side transactions; their manager functions are always executed nontransactionally. If a server-side transaction aborts, the client can

retrieve the abort reason by calling the **trpc_ServerSideAbortReason** function. The server can call the **trpc_IsLocallyWrapped** function to determine whether a transaction that is executing locally is a server-side transaction.

The **trpc_ServerSideIgnoreAbort** function can be used to request that the current TRPC be allowed to return normally, even if the server-side transaction aborts.

Server-side transactions can improve the performance of an application because transactional information is *not* sent on each TRPC. However, because the client and server do not share transactional information, only client applications that do not rely on transactional guarantees should use server-side transactions.

Termination functions

An application must terminate using the **trpc_Terminate** function to orderly shutdown the server or to exit a client application.

Functions for manipulating binding handles

The TRPC library provides functions that manipulate transactional handles. Many of these functions wrap functions of the DCE RPC runtime library that manipulate RPC binding handles. For further information see the *Functional Specification for OSF DCE RPC*.

TRPC defines the following functions:

- **trpc_BindingCopy**
- **trpc_BindingFromStringBinding**
- **trpc_BindingToStringBinding**
- **trpc_ConsBinding**
- **trpc_CreateBinding**
- **trpc_FreeBinding**
- **trpc_GetRpcHandleFromBinding**
- **trpc_InqObjectFromBinding**
- **trpc_InqTimeoutFromBinding**
- **trpc_ResetBinding**
- **trpc_SetObjectBinding**
- **trpc_SetTimeoutBinding**

TRPC also defines two functions used to get information from a transactional handle: the **trpc_GetApplIdFromBinding** function and the **trpc_GetAddressFromBinding** function.

The **trpc_GetManagerInfo** function can be used to return the binding handle, transaction information, and interface information in a manager function.

Diagnostics

This section documents the diagnostic support provided by TRPC. This support takes the form of informative messages that trace or unrecoverable erroneous events (warnings or fatal-errors) may generate during use of TRPC. It also documents the snapshot that the state dump may produce.

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring-buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

TRPC generates a *fatal run-time error* whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility (see "Trace and state dump information" on page 88) and the application's execution terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility (see "Directing output"). This section defines the fatal error messages that may be output by TRPC, and briefly describes why they occur, suggesting how they may be avoided, where possible.

TRPC has the following fatal error messages:

- ″Runtime library could not determine which protocol sequences are in use.″—The **trpc_Init** function finds that for some reason the underlying DCE does not support any protocol sequences. This is really a very obscure error and should rarely occur. Verify the DCE protocol sequences.
- ″Runtime library could not create a thread pool to handle asynchronous messages.″—A DCE RPC call failed. Obtain a functional DCE library.
- ″Runtime library could not acquire the identifier associated with its own interface specification.″—A DCE RPC call failed. Obtain a functional DCE library.
- ″Runtime library could not get the DCE RPC runtime to begin listening for asynchronous RPCs.″—A DCE RPC call failed. Obtain a functional DCE library.

- ″Runtime library could not use an endpoint specified in trpc_BindWkEndpoints.″—This message is generated when TRPC finds that it cannot create a valid endpoint for one of the pairs of protocol sequence endpoint specification pairs that are passed to it as parameters. Typically, this occurs if some other process (including itself) on the host is already using the specified endpoint and protocol sequence. If this is the case, the specified endpoint and protocol sequence pair can only be used after the other process has been killed. If the same process has already created an endpoint, ensure that it does not try to create it again.
- ″A call to trpc_BindWkEndpoints specified endpoints that have not been bound by the application.″—Well-known endpoints have not been bound by the application. Bind the well-known endpoints.
- ″Runtime library could not register its own interface with the DCE RPC runtime.″—This message is generated when TRPC cannot register its interface with the DCE RPC runtime and the **rpc_server_register_if** function returns an error. Another possible cause is that the TRPC interface is already registered; however, this should not happen. TRPC guards against duplicate registration (unless the application developer deliberately tries to do it. Typically, this would imply a serious failure, for instance, the process running out of memory.
- ″Runtime library could not register an endpoint with the DCE Directory Service. The Directory Service path used was %s. The rpc_ns_binding_export function returned 0x%x.″—TRPC cannot register its interfaces and endpoints with the DCE Directory Service. This will occur during a call to **tran_Ready**. The other possible reason for this error is that the path specified in the **trpc_SetEnvironment** call may not have been created in the name space, or the process does not have the privilege to create entries in the path specified. Ensure that the DCE Directory Service is active and can be contacted from the local host.
- ″Runtime library could not register an endpoint with the endpoint mapper (rpcd). The rpc_ep_register function returned 0x%x.″—TRPC cannot register its interfaces and endpoints with the endpoint mapper. This will occur during a call to **tran_Ready**. The other possible reason for this error is that the path specified in the **trpc_SetEnvironment** call may not have been created in the name space, or the process does not have the privilege to create entries in the path specified. Ensure that the DCE Directory Service is active and can be contacted from the local host.
- ″Threading package could not create listener thread.″—One of the threads that TRPC creates calls **rpc_server_listen** which returns a failure. This can occur if the application has already called **rpc_server_listen**. To avoid this, application developers are advised to call **bde_rpc_server_listen** which waits on a condition variable until the server is shutdown.

- ″Threading package could not create cleanup thread.″—TRPC cannot use the underlying threading package to create the threads need for operation. Typically, a thread package fails when the system runs out of memory.
- ″The runtime library being used does not match the version of TIDL used. Interface name %s, function name %s, stubs version %d.″—An outdated interface is in use. Run TIDL on the outdated interface.
- ″TRAN gave TRPC an incorrect service identifier during a message-sending upcall.″—TRAN provided an incorrect service identifier to TRPC. An incidence of this error would indicate a problem with TRAN. Contact Product Support.
- ″Runtime library could not acquire a string binding for a well-known endpoint.″—A DCE RPC call failed. Obtain a functional DCE library.
- ″Runtime library was provided a well-known endpoint that uses an invalid protocol sequence.″—The **trpc_BindWkEndpoints** function was passed an invalid protocol sequence. Refer to the *Functional Specification for OSF DCE RPC* for a list of valid protocol sequences.
- ″Runtime library was provided a well-known endpoint lacking network or endpoint information.″—A well-known endpoint lacks network or endpoint information. Fix the endpoint.
- ″Runtime library has been given a well-known endpoint that has not been bound by the application.″—A well-known endpoint has not been bound by the application. Bind the endpoint.
- ″Runtime library needs either a well-known endpoint or the ability to use the name service because the application is recoverable.″—The application is recoverable and is not using the DCE Directory Service to register itself using dynamic bindings. This check is performed when the application calls the **tran_Ready** function which then calls a TRPC function. Provide a well-known endpoint or allow the application to use the DCE Directory Service.
- ″Runtime library could not create a dynamic endpoint for some protocol sequence.″—TRPC cannot create bindings (using **rpc_server_use_protseq**) for protocol sequences for which no bindings currently exist. TRPC tries to create the additional bindings because the application wishes to use certain protocol sequences but has not bothered to create bindings for them. This is a DCE problem, or perhaps, the system is out of dynamic endpoints.
- ″A transactional RPC was made using an implicit transaction identifier, but no callback was registered.″—TRPC was not initialized correctly.
- ″A transactional RPC was made before the TRPC runtime library was initialized.″—TRPC was not initialized correctly. A transactional RPC was attempted before **trpc_Init** and **tran_Init** completed.
- ″The TRPC principal environment value does not contain a cell names, and the runtime library cannot determine the local cell name.″—The principal argument passed to the **trpc_SetEnvironment** function, or provided in the

ENCINA_TRPC_PRINCIPAL or ENCINA_PRINCIPAL environment variables does not contain a cell name. TRPC must be able to obtain a fully-specified principal name, including the cell name. This error message means that the runtime library has tried to get the local cell name in order to form a fully-specified name, but the DCE runtime function **dce_cf_get-cell_name** has failed. The probable cause of this error is that the DCE is not installed, or is not installed correctly, on the machine.

Warning messages

TRPC has the following warning messages:

- ″The runtime library could not establish authentication when sending an asynchronous message. The rpc_binding_set_auth_info function returned 0x%x.″—The DCE RPC function that is used to make a communication channel secure has failed, so TRPC cannot use that channel for asynchronous (TRAN commitment) messages. Check to be sure that the DCE Security Service is still running and that the principals used to make TRPCs are still valid. Look at the DCE RPC runtime status code in the message for further information.
- ″The runtime library could not establish authentication when caching an RPC handle. The rpc_binding_set_auth_info function returned 0x%x.″—The DCE RPC function that is used to make a communication channel secure has failed, so TRPC cannot use that channel when caching an RPC handle. Check to be sure that the DCE Security Service is still running and that the principals used to make TRPCs are still valid. Look at the DCE RPC runtime status code in the message for further information.

Audit messages

TRPC does not emit any audit messages.

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes calls on TRPC. All TRPC interface functions are capable of tracing their entry, parameters, and exit. In addition, important events during the execution of transactional and nontransactional RPCs and commit processing may be traced. Clients of TRPC can enable a specific class of event tracing, such as entry/exit, or specific events.

The level of tracing is controlled by the value of a single variable that is exported by TRPC:

```
unsigned long trpc_traceMask
```

The Encina Trace Facility defines a number of ″bit constants″ that when set in the above variable cause a specific form of tracing to occur. For example, the

TRACE_ENTRY constant enables entry/exit tracing of TRPC interface functions. TRPC also defines TRPC-specific bit constants for important classes of internal event.

Global trace levels

TRPC supports the global trace levels defined by the Encina Trace Facility. These levels are as follows:

- TRACE_ENTRY — traces the entry and exit of functions that make up the TRPC interface.
- TRACE_PARAM — traces the parameters passed to the interface functions.
- TRACE_EVENT — traces all events in TRPC. This trace class outputs large amounts of information, and for this reason, is not always recommended.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- TRACE_GLOBAL
- TRACE_NONE

TRPC trace levels

TRPC exports TRPC-specific bit constants, which enable tracing for the two major TRPC events: out-of-band message processing and caching of transactional handles. These bit constants are defined as follows:

```
TRPC_TRACE_HANDLE_CACHE_EVENT     0x00010000L
TRPC_TRACE_ASYNC_COMM_EVENT       0x00020000L
```

The TRPC_TRACE_HANDLE_CACHE_EVENT trace bits enable tracing the caching of the transactional handles by TRPC. The TRPC_TRACE_ASYNC_COMM_EVENT trace bits enable tracing the out-of-band message processing.

All trace statements that use the above trace bits are EVENT statements and must be compiled out of TRPC when performance is critical.

State dump

TRPC provides the **trpc_DumpState** function to dump its important internal state information.

Chapter 5. Abort Facility

Abort facility overview

The Encina Abort Facility provides support for setting and retrieving abort reasons. Abort reasons are used to determine why a transaction aborted and are typically presented as strings or codes. A number of Encina components support the use of abort reasons. The functionality of the Abort Facility is generalized to allow it to be used with any component that requires abort reasons.

The Abort Facility chapter is organized as follows:

- "Abort reasons" describes the abort reason data structure used to define an abort reason. It also describes related functions for formatting, setting, and retrieving abort reasons.
- "Abort strings and codes" on page 92 describes the Abort Facility functions for setting and retrieving abort codes and strings.
- "Exported variables and constants" on page 93 describes the variables and constants exported by the Abort Facility.

Abort reasons

An abort reason is a structure (of type **encina_abortReason_t**) that consists of a format identifier and an abort code and/or abort data. The format identifier is a DCE universal unique identifier (UUID) that uniquely identifies the abort reason. The abort code is a signed, 32-bit integer that defines the reason for aborting a transaction. Abort data is component specific, and may consist of a string describing the reason for an aborted transaction and an integer defining the length of that string. Abort data can also be used to further qualify an abort reason described by the abort code.

Defining abort codes as part of abort reasons allows abort reasons to be easily compared. That is, the abort codes can simply be compared instead of trying to compare variable-length strings. In addition, an abort code can be encoded in such a way that a formatting function can be used to convert the abort code to an NLS-compliant string for printing.

A formatting function is used to format abort codes and data for a specific format identifier into a null-terminated string for printing. A format identifier is a UUID created with the **uuidgen** utility provided by the DCE; the format identifier is referred to as the *format UUID*. Abort formatting functions can be registered with a component or an application by calling the

encina_RegisterAbortFormatter function. After an abort reason has been retrieved, a formatting function can be invoked by passing the abort reason to the **encina_FormatAbortReason** function; if the format UUID of the abort reason matches the format UUID of a registered formatting function, that formatting function is invoked to format the abort reason.

For example, all the abort codes defined in Encina components follow the format used by the Encina status codes. A formatting function, which is registered for all the Encina components that define abort codes, interprets the abort codes using the **encina_StatusToString** function.

Setting and retrieving abort reasons can be done using the **encina_SetAbortReason** and **encina_GetAbortReason** functions. These two functions require that an **encina_abortReason_t** structure be defined, and the **encina_SetAbortReason** function requires that it has a valid format UUID. Also, a formatting function should be defined and registered with the same format UUID if the abort reason can be converted into a printable string. Abort reasons set for a transaction with the **encina_SetAbortReason** function can contain abort data, an abort code, or both.

The **encina_FreeAbortReason** function can be used to de-allocate the memory allocated by the **encina_GetAbortReason** function. This function only needs to be called when an abort reason contains abort data.

Abort strings and codes

The Abort Facility provides an alternative to defining an **encina_abortReason_t** structure, generating a format UUID, and writing a formatting function; it includes four higher-level functions that can be used to set and retrieve abort reasons. These functions set and retrieve the abort reason for a transaction either as an abort string or an abort code only.

The **encina_SetAbortString** and **encina_GetAbortString** functions treat abort reasons as null-terminated character strings. The **encina_SetAbortCode** and **encina_GetAbortCode** functions treat abort reasons as integer codes. An abort reason that is set for a transaction with these higher-level functions can have either an abort string or an abort code—it cannot have both.

Encina automatically registers formatting functions for abort reasons that are set as strings (or as abort codes that use the format defined for Encina status codes). The formatting function for strings simply returns the abort string as a null-terminated character string.

Exported variables and constants

Encina exports two variables and two constants that are used by the Abort Facility. The variables define format UUIDs that can be used to label abort reasons. Encina provides formatting functions that are automatically registered to format abort reasons that use either of these two format UUIDs. The exported variables are defined as follows:

```
uuid_t ENCINA_STANDARD_FORMAT_UUID
uuid_t ENCINA_STRING_FORMAT_UUID
```

The `ENCINA_STANDARD_FORMAT_UUID` variable is used to label an abort reason that follows the format used by Encina status codes. Abort codes that can be decoded by the **encina_StatusToString** function should be registered with this format UUID. Encina components, such as TRAN, that return an Encina status code when a transaction is aborted use this format UUID.

The `ENCINA_STRING_FORMAT_UUID` variable is used to label an abort reason that has been specified as a string using the **encina_SetAbortString** function.

In addition to the two variables, the following constants are also exported:

```
#define ENCINA_STRING_FORMAT_UUID_STRING  \
     "007b53a4-cf9c-1cfc-b6cc-9e620503aa77"

#define ENCINA_STANDARD_FORMAT_UUID_STRING  \
     "004bdeda-d0c2-1cfc-8e01-9e620503aa77"
```

These constants can be used to create a variable of type **uuid_t** (using the **uuid_from_string** DCE function) with the same value as the exported variables. These constants are used by Encina components that use Tran-C, such as the Encina Structured File Server, because Tran-C requires that the format UUID be specified as a string.

The abort facility functions return a value of the **encina_status_t** type, which can also be converted into a string using the **encina_StatusToString** function.

Chapter 6. IBM Transarc/Encina DCE Utilities (TRDCE)

TRDCE overview

IBM The Transarc/Encina Distributed Computing Environment Utilities library (TRDCE) provides utilities for constructing DCE client and server programs. Some of these functions are intended to provide a simpler interface to common services, such as server registration and client binding, while other functions add functionality to existing DCE functions. Refer to the *OSF DCE Application Development Guide* for information on DCE.

In addition to server registration and client binding, the TRDCE functions are used to listen for remote procedure calls (RPCs) and control RPC dispatching, provide administrative control of RPC interfaces, and cope with security requirements. Several utility functions and destruction functions are also defined.

The TRDCE chapter is organized as follows:

- "Client binding and server registration functions" on page 96 describes functions for registering servers and enabling clients to acquire bindings to servers.
- "Server listening and dispatch-handling functions" on page 97 describes the functions for controlling the number of threads that service an application's RPC interfaces.
- "Interface control functions" on page 97 describes the functions for registering RPC interfaces and getting information about registered interfaces.
- "Security functions" on page 99 describes the functions used to handle the security requirements of applications.
- "Deallocation functions" on page 99 describes functions for freeing the memory allocated by other TRDCE functions.
- "General utility functions" on page 99 describes functions that provide general support for the TRDCE functions.
- "Diagnostics" on page 100 describes the diagnostic support provided by TRDCE.

TRDCE header files and libraries

The **trdce/trdce.h** header file contains structure and data type declarations required by the Encina TRDCE interface functions for the C language. This file must be included in any C file that uses TRDCE functions.

The Encina TRDCE functions are contained in the **Encina** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Client binding and server registration functions

Encina provides simplified functions for registering servers and enabling clients to acquire bindings to servers. Using these functions, servers can register with the DCE Directory Service or RPC runtime and obtain the server binding handles for any well-known endpoints registered by the server. Clients can lookup server binding handles using the name that the server registered, get string bindings from binding handles, and set protection levels for bindings. Either DCE Directory Service names or RPC string bindings can be used for registration and lookup.

Servers use the **trdce_ServerRegister** function to initialize and register binding handles. The function registers servers at specified locations, which can be either DCE Directory Service names or string bindings for well-known endpoints. The **trdce_ReturnWkEndpoints** function returns the subset of the server's bindings that were generated from well-known endpoints (specified using string bindings).

Clients use the **trdce_BindingImport** function to obtain a server binding handle, given the name of the server. Once the binding handle is obtained, the **trdce_ReturnCallbackBinding** function can be used to get a *string binding* (binding information in string form) from the binding handle, so the server can use the string binding to make RPCs to the client. Clients use the **trdce_BindingSetProtectionLevel** function to set an explicit protection level for use by the **trdce_BindingImport** function call.

Both clients and servers use the **trdce_QualifyName** function. The **trdce_QualifyName** function guarantees that a server name is fully qualified and, therefore, unambiguous. For example, the name **logfile1**, which is relative to a component, could be translated into **/.:/encina/log/logfile1** to make it a fully-qualified name.

Encina offers a simple translation facility that allows names to be associated with RPC string bindings. The ENCINA_BINDING_FILE environment variable can be set to the name of a text file that contains name-to-binding translations. Each line of the binding file must contain a single translation consisting of a fully-qualified DCE Directory Service name followed by any amount of white space followed by a string binding. Whenever a name matching one in the translation file is passed to one of the TRDCE binding functions described in this section, the corresponding string binding is used

instead. This facility lacks the flexibility of the DCE Directory Service and is not intended as a full replacement, but it may be useful in situations where the Directory Service cannot be used.

Server listening and dispatch-handling functions

Extensions to the DCE RPC dispatch mechanism allow independent libraries within an application to control the number of threads that service their interfaces. Any component, the server main program or an independently-developed library, can allocate a *thread pool* to service its RPC interfaces. A component can use the **trdce_CreateThreadPool** function to create a pool. A component can create any number of thread pools.

If a component allocates a thread pool, it can then register a *dispatch function* to determine whether a given RPC should use its thread pool; this determination is typically based on the RPC interface identifier. The **trdce_RegisterSimpleDispatch** function causes all calls to a specified RPC interface to be directed to a specified thread pool.

Each component that uses a thread pool to service its RPC interface must ensure that the application is listening for incoming RPCs by calling the **trdce_ServerListen** function. DCE system limitations require that a default thread pool be created for servicing those calls that do not get dispatched to a specific thread pool. Because the DCE RPC dispatch mechanism requires that a default thread pool be initialized when an application begins listening, the **trdce_ServerListen** function includes a parameter for specifying the size of the default thread pool.

Server main programs that require a thread pool of a specific size should create a thread to call the **trdce_ServerListen** function with the correct number of threads before initializing other components that might also make the call. Library components that need to receive RPCs should call the **trdce_ServerListen** function asking for zero threads; a default number of threads will be provided. This function returns only when the application invokes the **rpc_mgmt_stop_server_listening** function.

Interface control functions

Encina provides several functions that can be used to control RPC interfaces. Server applications can call these functions to register interfaces and to obtain information about the interfaces that are registered. The interface control functions support administrative RPCs that can be used to register and unregister RPC interfaces at runtime.

Encina programs can register interfaces with the TRDCE library and the RPC runtime. The **trdce_OfferInterface** function registers an RPC interface

specification and entry point vector with TRDCE and associates a descriptive string with the interface. TRDCE registers the interface with the RPC runtime automatically (using the DCE RPC function **rpc_server_register_if**) and stores the registration information for later retrieval.

Alternatively, the **trdce_DefineInterface** function registers an interface with TRDCE and stores the registration information, but it does *not* register the interface with the RPC runtime. Unlike the **trdce_OfferInterface** function, the **trdce_DefineInterface** function allows a manager type UUID to be specified when registering an interface.

Two interface control functions can be used to return information about registered interfaces. The **trdce_ListInterfaces** function returns a list of interfaces registered with TRDCE. The **trdce_QueryInterface** function returns the registration information—interface specification, entry point vector, and descriptive string—for a given interface and manager type UUID.

Encina also provides the ability to create an RPC *interface database*. An interface database contains information about the RPC interfaces registered by an application server. These interfaces are defined in Interface Definition language (IDL) or Transactional Interface Definition Language (TIDL) files. An interface database consists of one or more of database files, which are created from individual IDL or TIDL files. The database lists the operations that are available from each interface. By querying the database, you can discover which interfaces and operations are supported by a particular application server. You can also find this information by reading the database files directly.

To create an interface database file, use the **extractIdlNames** command:

```
% extractIdlNames -auto_create filename
```

where *filename* is the name of an IDL or TIDL file. The `-auto_create` option creates a database file with the same name as the interface UUID contained in the IDL or TIDL file. For more information about other command-line options, see the reference page for the **extractIdlNames** command. The `NLSPATH` environment variable specifies the location of the interface database.

To query the interface database, use the **trdce_LookupInterface** function. This function takes a n interface identifier as an input parameter. If the specified interface is found, it returns the interface name, its source file, the number of operations associated with the interface, and a list of operation names.

After you have finished using the list of operation names, deallocate its memory by using the **trdce_FreeOpNames** function.

Security functions

Encina provides several functions that are used to handle the security requirements of applications. These extensions to the DCE Security Service are used for managing key files and principals and establishing login contexts.

The key file for an application can be set using the **trdce_SetKeyFile** function and queried using the **trdce_ReturnKeyFile** function. The principal for an application can be set using the **trdce_SetPrincipal** function and queried using the **trdce_ReturnPrincipal** function. The **trdce_IsPrincipalSet** function determines whether a principal is set for the application.

After a key file and principal are set, a new login context can be created using the **trdce_SecLoginContextCreate** function. To prevent the login context from expiring, the **trdce_SecLoginContextRefresh** function periodically refreshes the login context. Similarly, the key for the principal using the key file is periodically refreshed using the **trdce_SecKeyManagement** function. As an alternative to calling each of these three functions individually, the **trdce_SecManagement** function combines them in one call. In addition, the **trdce_SecManagement** function calls the **rpc_server_register_auth_info** function to register an authentication service. (For simplicity, the period and policies used in the refresh functions are unspecified; if an application has specific policy requirements, management functions should be implemented using DCE facilities.) The **trdce_SecLoginContextCertify** function enables and disables certification for new login contexts. By default, login contexts are not certified when they are created or refreshed.

Deallocation functions

Encina provides functions for freeing the memory used by data structures created as a result of using some of the TRDCE functions.

- The **trdce_FreeBindingVector** function (used by servers) frees the memory that stores the vector of binding handles created by the **trdce_ReturnWkEndpoints** function.
- The **trdce_FreeProtseqVector** function frees the memory that stores the vector of protocol sequences created by the **trdce_ReturnSupportedProtseqs** function.
- The **trdce_Free** function frees allocated memory for which no specific deallocation function is provided.

General utility functions

Encina provides several utility functions that support the TRDCE functions.

- The **trdce_GetDCEStatus** function returns a status code associated with an *exception* (an object providing information about an error condition), as implemented by the DCE Exception package.
- The **trdce_NormalizeProtseq** function translates an abbreviated protocol sequence into its expanded form. For example, “ip” is translated to “ncadg_ip_udp.”
- The **trdce_ProtectLevelFromString** function can be called by a client to obtain a protection level for communication with a server. The protection-level value is obtained either from a specified string or the ENCINA_PROTECT_LEVEL environment variable.
- The **trdce_ReturnSupportedProtseqs** function returns a vector of RPC protocol sequences supported by the DCE RPC runtime and by the Encina components in an application.

Diagnostics

This section documents the diagnostic support for the TRDCE functions.

Directing output

All trace, dump, warning, and fatal error output can be directed to user-specified files or a ring buffer by using the Encina Toolkit Trace Facility functions. By default, this output is sent to the ring buffer. The ring buffer can be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See “Appendix A. Tracing and debugging Encina Toolkit applications” on page 231 for more details on the Encina Trace Facility.

Warning messages

The TRDCE functions that support the use of well-known endpoints issue the following warning messages:

- ″ENCINA_BINDING_FILE, *file name*, could not be opened.″—The ENCINA_BINDING_FILE environment variable is set to *file name*, but either the **trdce_BindingImport** or **trdce_ServerRegister** function is unable to open *file name* for reading. Verify that ENCINA_BINDING_FILE is set correctly and that *file name* is readable.
- ″No bindings were found in ENCINA_BINDING_FILE *file name*.″—The ENCINA_BINDING_FILE environment variable is set to *file name*, but either the **trdce_BindingImport** or **trdce_ServerRegister** function found no bindings for the server in *file name*. Verify that *file name* is the correct file and that the bindings contained in the file use supported protocol sequences. Also, verify that the RPC_SUPPORTED_PROTSEQS environment variable is set correctly.

- "Ignoring well-known endpoint *string binding* - not my network address."—The *entryName* given to the **trdce_ServerRegister** function is a string binding or matches a string binding in the file named by ENCINA_BINDING_FILE environment variable. This string binding contains a network address that is not the current host's primary address. Verify that the string binding is correct and that the server is running on the correct host.

Part 3. Encina Toolkit Server Core

Chapter 7. Lock Service (LOCK)

Lock Service overview

The Encina Toolkit Lock Service is a component that can be used to support the serializability property of transactions: the property which guarantees that even if transactions execute concurrently, they appear to execute in some serial order (one after another). The Lock Service chapter is organized as follows:

- The remainder of this section explains basic locking concepts and the locking model supported by the Lock Service. Read this section before using any Lock Service calls.
- "Data types" on page 26 describes the Lock Service data types.
- "Application interface" on page 113 defines the functions that make up the application interface to the Lock Service.
- "System interface" on page 114 defines the system interface functions.
- "Diagnostics" on page 116 describes the diagnostic support provided by the Lock Service.

Lock Service

Ensuring that transactions are serializable requires a *concurrency control mechanism*. The Lock Service cooperates with applications to support a concurrency control mechanism suitable for guaranteeing transaction serializability. The Lock Service implements a concurrency control technique known as *locking*; an application obtains a *lock* on a resource before accessing it. Locks are always obtained of behalf of transactions in a given *lock mode*. A lock request on behalf of one transaction will not be granted if an unrelated transaction holds the lock in a conflicting mode.

The Lock Service supports a range of different lock modes. In addition to conventional read/write locks, the Lock Service provides *intention locks* which can be used to lock resources that are arranged in a hierarchical fashion. The duration of a lock can be either the duration of the transaction in which the lock is obtained, or can be an instant duration. An *instant duration lock* is a lock that is granted but not held, and can be used to implement complex locking techniques such as key-range locking.

The Lock Service does not define the granularity of the resources that are locked, instead the Lock Service provides a *logical locking* facility. Logical locking involves clients associating *lock names* with resources, where the lock names are the entities that are locked. In addition to a lock name, a client

must specify a *lock namespace*. A lock only conflicts with another lock of the same name if both locks share the same lock namespace.

The Lock Service supports nested transactions by allowing a transaction to obtain a lock held by an ancestor. Nested transactions that have a common ancestor are said to belong to the same *transaction family*. All members of a transaction family commit together and drop their locks. When a nested transaction aborts any newly obtained locks are released.

Normally, the locks obtained during a transaction are not released until the transaction completes (by either committing or aborting). The Lock Service therefore supports transaction duration *two-phase locking*. Occasionally however, a transaction may wish to release a lock early when it is known to be safe to do so. The Lock Service therefore provides a primitive for releasing individual locks.

The Lock Service has been designed as a modular component that manages a transaction's locks in conjunction with applications that use the Encina Distributed Transaction Service and the Encina Recovery Service. An ephemeral application (an application that does not use the Encina Recovery Service) can also use the Lock Service by relying on support from the Transaction Service.

The Lock Service interface can be divided into two parts: an application part and a system part. The application part provides functions to obtain and release locks on logical lock names. The system part consists of the lower level calls necessary to support the application interface. In general, the application programmer can ignore the system interface.

Lock Service model

This section explains the locking model supported by the Lock Service and briefly discusses how the Lock Service should be used by applications.

The following sections are brief explanations of the important issues that relate to the locking model supported by the Lock Service. The functions provided by the Lock Service, if used in the manner intended, efficiently support two-phase locking for both top-level and nested transactions. For a complete discussion of these issues the reader is directed to one of the standard texts on the subject (*Concurrency Control and Recovery in Database Systems* by P.A. Bernstein, V. Hadzilacos, and N. Goodman, or "Notes On Data Base Operating Systems" by J.N. Gray in *Operating Systems: An Advanced Course* (ed. Bayer, Graham, and Seegmuller)).

CAUTION:
The DCE threading model permits the cancellation of threads. However, the Lock Service assumes that its work will not be interrupted by such a cancellation. Locks should be obtained only from threads with thread cancellation disabled. If a thread holding a lock is canceled, the results are undefined.

Two-phase locking

The Lock Service provides primitives to support transaction duration locking. During the normal execution of a transaction, no locks will be dropped before the end of the transaction. When the transaction completes, the Lock Service must be informed so that the locks the transaction holds may be released. While releasing locks, no new locks may be obtained by the transaction.

Transaction duration locking is a special case of *strict two-phase locking*. In the first phase (the *growing phase*), a transaction obtains locks that are kept until the second phase (the *shrinking phase*), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction. Two-phase locking is sufficient to guarantee serializability.

When a transaction obtains a lock that is not needed to ensure the transaction's serializability, it is permissible to release the lock. The Lock Service therefore provides a function that releases individual locks. This function should only be used to drop locks that have been acquired on resources that have not been accessed, and on whose value the outcome of the transaction does not depend.

Lock modes

The TRAN-C lock interface supports five different lock modes, which ensure that different transactions cannot access the same data at the same time, and also allow varying degrees of concurrent access to resources within an application. When a transaction locks data in *read* mode, other transactions are permitted to read the locked data; no transaction can modify the data while the read lock is in effect. If a transaction holds data in *write* mode, no other transaction is permitted any access to the data. *Intention* mode locks (read and write) provide locking for hierarchical resources, which can minimize potential conflicts on lower-level resources without necessarily locking the entire data structure. A transaction can use an *upgrade* mode lock when required to read data that it possibly will need to modify. Table 4 on page 108 lists the compatibility between the various locking modes (the symbol **Yes** is used to indicate when locks do not conflict).

Table 4. Lock modes

	Transaction B requests mode				
Transaction A has mode	**IR**	**R**	**U**	**IW**	**W**
Intention Read (IR)	**Yes**	**Yes**	**Yes**	**Yes**	
Read (R)	**Yes**	**Yes**	**Yes**		
Upgrade (U)	**Yes**	**Yes**			
Intention Write (IW)	**Yes**			**Yes**	
Write (W)					

Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required. Coarse-grained locks incur low overhead (since there are fewer locks to manage) but reduce concurrency since conflicts are more likely to occur. Fine-grained locks improve concurrency but result in a higher locking overhead since more locks are requested Since the Lock Service supports logical locking, an application can be developed to use coarse-grained or fine-grained locks. The Lock Service also supports *variable granularity* locking that exploits the natural hierarchical relationship between locks of different granularities.

Consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a field of a record, a coarse-grained lock may be set on the database, but at the cost of restricting other transactions from accessing the database. Clearly, this level of locking is unsuitable, as is only setting a lock on the record (since another transaction may set a lock on the file holding the record and delete or modify it).

Variable granularity locking provides intention locks for read and write locks, allowing a transaction to obtain an intention lock on the ancestor of the required resource. To read a field, the transaction obtains an *intention read* lock (IR) on the database and file (in this order), before obtaining the read lock (R) on the record. Intention read locks conflict with write locks (W), and intention write locks (IW) conflict with read and write locks.

An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock involving a transaction reading and then writing resources based on the values read. If two transactions do this concurrently then they will deadlock. If both transactions attempt to obtain an upgrade lock instead, this situation will be avoided.

When a transaction requests a lock that cannot be granted because an unrelated transaction holds the lock in a conflicting mode, the transaction must wait until the holding transaction releases the lock. The Lock Service

enforces a queueing policy such that all transactions waiting for a new lock are serviced in a first in, first out order, and subsequent requests are blocked by the first request waiting to be granted the lock, unless the transaction on whose behalf the lock is being acquired is a member of the same transaction family as an existing holder of the lock.

Multiple possession semantics

In the conventional locking model (see "Notes On Data Base Operating Systems" in *Operating Systems: An Advanced Course* (ed. Bayer, Graham, and Seegmuller) by J.N. Gray for further information), when a transaction holds a lock and requests a lock on the same name in a stronger mode, the lock is *promoted* from the weaker mode to the stronger once the stronger lock can be granted without causing a conflict. Since lock modes form only a partial order, there will not always be a stronger mode; in cases where neither mode is stronger, the lock will be promoted to the weakest mode that is at least as strong as either of the two modes. Additionally, a count is associated with each lock, describing the number of times the transaction has obtained the lock. The count is necessary to ensure that the unlock operation has a predictable effect.

The Lock Service supports a slightly different locking model, wherein a transaction can hold a lock in multiple modes simultaneously. When a lock is held by a transaction in more than one mode, other transactions will not be granted the lock unless the requested lock is compatible with all of the modes in which the lock is held. A separate count is associated with each mode, describing the number of times the transaction has been granted the lock in that mode. The set of counts for a transaction are referred to collectively as the transaction's *possession count vector* for the lock. This model is referred to as *multiple possession semantics*. The model facilitates the definition of unlock and related operations.

The Lock Service unlock operation decrements the count associated with the given lock mode. If the count is greater than zero after being decremented, the operation has no other effect; if it is reduced to zero, the mode in which the lock is held is weakened appropriately, potentially resolving one or more lock conflicts. The Lock Service operation to change the mode of a held lock waits until the transaction can obtain the lock in the new mode and then decrements the count associated with the old mode and increments the count associated with the new one. As with the unlock operation, further action is necessary only if the count associated with the old mode is reduced to zero.

Nested transactions

Lock conflicts within a transaction family are treated somewhat differently than conflicts between unrelated transactions. The underlying principle is the same for both: transactions must not be able to observe the effects of other

transactions that might later abort. Unrelated transactions can abort independently; therefore, one transaction must not be permitted to acquire a lock held by an unrelated transaction.

Nesting imposes abort dependencies among related transactions. These dependencies make it possible to relax the rule that two transactions cannot acquire the same lock without breaking the underlying principle. Conventional systems describe the nesting rules as though only one transaction holds a lock, with locks being transferred among related transactions appropriately. Our Lock Service, instead, describes the rules in terms of multiple related transactions all holding a lock at once; this model better supports multiple possession semantics and the ability to release locks early.

The Distributed Transaction Service imposes abort dependencies on nested transactions. A parent transaction cannot abort without causing all of its children to abort. A child transaction that ends successfully cannot abort without causing its parent to abort. A transaction that cannot abort without causing another related transaction to abort (according to these guidelines and logical deductions) is said to be *committed relative to* that other transaction.

These abort dependencies inherent in the nesting structure allow the lock conflict rules to be relaxed safely. No partial effects can be observed and committed if all transactions that have done work cannot abort without the observer being aborted. This translates into a simple rule for nested locking: if all transactions that hold a lock are committed with respect to a transaction trying to acquire the lock, no conflict exists.

Conventional systems (see *Nested Transactions: An Approach To Reliable Distributed Computing* by J.E.B. Moss for further information) continue to treat a lock as being owned by a single transaction. When a nested transaction requests a lock that is already held by an ancestor transaction, it becomes the new owner. When a nested transaction commits, ownership of all of its locks is transferred to its parent. When a nested transaction aborts, ownership of its locks reverts the previous owners. The locking facility performs these lock transfers automatically.

The Lock Service provides a functionally identical model in which a lock may be held by multiple related transactions at once. When a nested transaction requests a lock, it is granted if all of the transactions holding the lock are committed relative to the requestor. Both the requestor and previous holders are then considered to hold the lock.

The Lock Service model interacts well with multiple possession semantics (see the previous section). Each transaction has its own possession count vector for a lock. A child transaction can acquire a lock held by its parent and then drop

that lock without causing its parent to lose the lock; the parent retains its possession. A transaction cannot drop a lock that it did not acquire itself. The lock possession semantics also require that each transaction acquire locks on its own behalf. It is improper to take locks on behalf of another transaction, and it is inappropriate to depend on locks held by other transactions.

Deadlock detection

The Lock Service provides a function that can be called to determine whether a transaction is deadlocked or not. This function, **lock_TranDeadlockDetect**, only detects deadlocks between transactions that are local to an application. The function is designed to be called either by a background thread that periodically checks for deadlocks, or in response to a timeout for a transaction that requests or holds locks. When a deadlock is detected, the Lock Service returns the identifiers of the transactions involved in the *wait-for cycle*. The Lock Service does not resolve the deadlock by aborting, or releasing a lock held by a transaction that is a member of the wait-for cycle. The resolution of the deadlock is the responsibility of clients of the Lock Service, thereby allowing a variety of different deadlock resolution policies to be implemented. One possible approach is to determine the transaction family that holds the fewest write locks (using the **lock_GetFamilyInfo** function) and abort the members of that family. To support this capability, the Lock Service provides a function that returns the number and mode of locks held by a transaction. By calling this function on behalf of the members of a wait-for cycle, a Lock Service client can, for example, abort the transaction with the fewest locks or the smallest number of write locks.

When transactions obtain locks at multiple applications they can also deadlock, however this situation will not be detected by the Lock Service deadlock detection mechanism. Applications must therefore rely on an external mechanism to detect, or undo, the deadlock. One simple technique is to associate idle period time-outs with transactions, and abort those transactions that exceed the idle period (the assumption being that they must be deadlocked).

Conflict callbacks

The Lock Service provides functions to register and unregister conflict callbacks for transaction families. A *conflict callback* is a function that will be delivered when the transaction family on whose behalf it is registered holds a lock that another transaction family is attempting to acquire in a conflicting lock mode (conflict callbacks are not delivered when the conflict is between members of the same family). Once registered, a conflict callback will be delivered for each conflict until either explicitly unregistered, or all locks are released by that transaction family using calls to the **lock_TranCompleted** function. A conflict callback may be registered and unregistered on behalf of a transaction family using the identifier (TID) of any member of the transaction family.

If a transaction holds locks that other transaction families are waiting for at the time of registration, then the callback will be delivered for all conflicting transactions. A conflict callback has the option to return a value, also called a vote, that will be used by the Lock Service to determine whether the conflict should be ignored. In the absence of a conflict callback that votes to allow the conflict, the requesting transaction would either not be granted the lock in the case of a non-blocking call, or block waiting for the lock in the case of a blocking call.

The Lock Service provides the **lock_RegisterConflictCallback** function to register conflict callbacks and the **lock_UnregisterConflictCallback** function to unregister conflict callbacks.

Using the Lock Service

The Lock Service supports serializability through strict two-phase locking, but relies on clients of the Lock Service to use the interface correctly. If locks are released before a transaction completes then serializability is not guaranteed. Since nested transactions *complete* along with their top-level ancestor, a nested transaction's locks must not be released before the top-level ancestor completes. The Lock Service interface does not stop a transaction's locks from being released before its ancestor completes; therefore, clients of the Lock Service must ensure that the function to indicate a transaction has completed is called at the correct time for all transactions in the family.

The Encina Recovery Service interface allows an application to register functions that will be called at the appropriate time to drop locks and reobtain locks after crash recovery. The Lock Service system interface functions can be used indirectly by these functions. The Distributed Transaction Service provides a callback that is suitable for ensuring that a transaction's locks are released correctly. This callback (**tran_CallAfterResolution**) can be used to register a function that informs the Lock Service that a transaction has completed for applications that do not use the Recovery Service.

LOCK header files and libraries

Header files

The file **lock/lock.h** contains the LOCK interface declarations for the C language. This file must be included in any file that uses interface functions or data types.

Libraries

The LOCK functions are in the **EncServer** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Data types

The Lock Service interface defines and uses a number of data types throughout the interface. All functions return a status code of type **lock_status_t**. The application interface functions all take arguments of type **lock_mode_t**, **lock_name_t**, **lock_space_t**, and **lock_waitMode_t**. When a lock is acquired, the duration of the lock can be specified with an argument of type **lock_duration_t**.

The interface also defines the **lock_lockList_t** and **lock_tranList_t** data types used by the system interface.

The **lock_Free** function can be used to free a block of memory allocated dynamically by the Lock Service.

Application interface

This section describes the application interface to the Lock Service. The functions provided by the application interface allow an application to acquire locks, wait for locks to become available, and release locks. The application interface also provides a function for creating a new lock namespace.

All functions provided by the application interface return a **lock_status_t** code and (with the exception of the lock namespace function) take the identifier of a transaction (a variable of type **tran_tid_t**) as an argument.

Acquiring locks

An application attempts to acquire a lock by calling **lock_Acquire**. If the lock is not held in a conflicting mode by an unrelated transaction, and the transaction on whose behalf the call is being made has not been aborted, then the lock will be granted.

To support more complex locking techniques (for example, Gray's technique for key-range locking), it is occasionally necessary to determine whether a lock could be granted. One approach is to obtain the lock and immediately release it, however this approach is undesirable since it involves internal state changes by the Lock Service, and (potentially) the allocation and immediate deallocation of resources to maintain the new lock.

To avoid any internal state changes and allocation/deallocation, the following call provides a mechanism for defining the duration of a lock; the lock may be held for the duration of a transaction, or for an instant duration. When an *instant duration lock* is granted, the Lock Service has determined that a transaction duration lock could have been granted (that is, the lock was not held in a conflicting mode by an unrelated transaction). Since an instant duration lock is not held, calls to release or change the mode of the lock are not valid.

Releasing locks

An application may acquire a lock, find that the lock is unnecessary and consequently wish to release the lock. There are two functions that support this capability: **lock_Release** and **lock_ReleaseAll**. They should be used with care because releasing a lock before the end of a transaction on a resource that has been read or modified may result in unpredictable (and non-serialized) behavior. Calling **lock_Release** releases a single lock possession (that is, undoes the effect of a single call to **lock_Acquire**), whereas calling **lock_ReleaseAll** releases all possessions of a lock (that is, undoes the effect of multiple **lock_Acquire** calls).

Changing lock modes

Occasionally a transaction may obtain a lock on a lock name in one mode and later decide to change the mode of the lock to another (stronger or weaker) mode. The **lock_ChangeMode** function is provided for this purpose. It is equivalent to obtaining the lock in the new mode followed by dropping the lock in the old mode but is more efficient (and expressive). Again, it should be used with care as weakening the mode of a lock may produce non–serialized behavior.

Creating lock namespaces

To create new unique lock namespaces, the **lock_CreateNameSpace** function is provided. The lock namespaces produced are volatile, so that it is the responsibility of clients of the Lock Service to ensure that a lock namespace can be recovered after a crash if new locks are to be obtained in that namespace.

System interface

This section describes the system interface to the Lock Service, a collection of functions that support the application interface described in "Application interface" on page 113. These functions can be categorized as initialization and termination, recovery, deadlock detection, and diagnostic support functions.

To ensure that lock names are unique, the Lock Service must maintain some state that can be restored when an application is restarted. The function **lock_Init** initializes the Lock Service and enables the application to return the last restart data logged by the application on behalf of the Lock Service. After the Lock Service has been initialized for the application, it does not need to be initialized for specific transactions; this occurs automatically the first time a lock is obtained on behalf of a transaction and involves the Lock Service creating an internal data structure to manage the transaction's locks. The Lock Service must be informed, however, when a transaction completes so that the transaction's locks can be released and the internal data structure deallocated. The Lock Service provides the **lock_TranCompleted** function for this purpose.

To guarantee consistency, the locks held by a transaction on recoverable resources must be capable of being saved in such a way that they can be reestablished after a crash and held until the resources have been fully recovered. The system interface to the Lock Service provides a function (**lock_Save**) that returns a *lock description*. An associated function (**lock_Restore**) is provided to restore the locks held given a lock description. The client is responsible for deciding when to call the **lock_Save** function, for logging the lock description this function returns, and calling **lock_Restore** with a lock description during the application's recovery phase.

To detect *deadlocks*, the situation where two (or more) transactions that hold locks are waiting for a lock held by the other waiting transaction, the Lock Service provides a deadlock detection function. The **lock_TranDeadlockDetect** function determines whether a specified transaction is a member of a wait-for cycle.

To support administration, the Lock Service provides a number of diagnostic functions. The **lock_GetTranList** function returns a list of transactions that hold locks or a list of transactions that are waiting for locks, or both. The **lock_GetTranInfo** function returns a list of locks held or being waited for by a transaction. The **lock_GetLockInfo** function returns a list of transactions that hold or are waiting for a specific lock.

Initialization and termination

Before any application interface functions can be called, the Lock Service must be initialized with the **lock_Init** function. Once initialized, an application can obtain locks and call the function to retrieve a lock description. When a transaction completes, the application must call the application interface function to inform the Lock Service. The **lock_TranCompleted** function will release all locks held by the transaction. This section describes the initialization and termination functions provided by the system interface.

Recovery interface

To support crash recovery, an application must be able to save its locks in, and recover its locks from, nonvolatile storage. The **lock_Save** and **lock_Restore** functions provide this support by allowing the locks held by a transaction to be saved into, and restored from, an in-memory data structure. They do not support the movement of the in-memory data structure to and from nonvolatile storage. This is the responsibility of clients of the Lock Service, which typically rely on their recovery service to perform this task.

Deadlock detection

To detect deadlocks, the Lock Service provides the **lock_TranDeadlockDetect** function. If a transaction is thought to be deadlocked, the function may be used to determine whether this transaction is a member of a wait-for cycle. Multiple concurrent invocations of this function are serialized to ensure correct behavior.

Conflict callbacks

The Lock Service provides the following functions to register and unregister conflict callbacks:

- **lock_RegisterConflictCallback**
- **lock_UnregisterConflictCallback**

Diagnostic support

The Lock Service diagnostic functions include the following:

- **lock_GetFamilyInfo** – returns a list of transactions belonging to a transaction family that hold locks, or are waiting for locks, or both, and the locks held or being waited for.
- **lock_GetLockInfo** – returns a list of transactions that hold or are waiting for a specific lock.
- **lock_GetTranInfo** – returns a list of locks held or being waited for by a transaction.
- **lock_GetTranList** – returns a list of transactions that hold locks, that are waiting for locks, or both.

Diagnostics

This section documents the diagnostic support provided by the Lock Service. This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal-errors) during use of the Lock Service, informative messages that trace calls on the Lock Service (tracing), and snapshots of the state of the Lock Service (state dumps).

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default, this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

The Lock Service generates a fatal run-time error whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility (see Section "Trace and state dump information" on page 117) and the application's execution is terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility (see "Directing output"). This section defines the fatal error

messages that may be output by the Lock Service, and briefly describes why they occur and how they may be avoided or remedied.

The Lock Service has the following fatal error messages:

- ″lockSpace_Create out of lock namespaces.″—The Lock Service has a finite number of lock namespaces since they must be guaranteed to be eternal yet small enough not to place a large penalty on the cost of their use. The application created too many lock namespaces. Lock namespaces should be considered a scarce resource and allocated only when necessary. There is no recoverable action from this error.
- ″lock_Save failed to allocate large enough buffer.″—The Lock Service enables clients to save an opaque description of the locks held by a transaction. A client-allocated buffer must be supplied to store the description. The Lock Service could not allocate a large enough buffer. There is no recoverable action from this error.

Warning messages

The Lock Service has the following warning message:

- ″lock_Free argument was a NULL pointer.″—The **lock_Free** call was made with a NULL pointer. No action.

Audit messages

The Lock Service does not emit any audit messages.

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes calls on the Lock Service. All Lock Service interface functions are capable of tracing their entry, parameters, and exit. In addition, important events during the execution of a Lock Service interface function like blocking on a condition variable to wait for a lock conflict to be resolved, may be traced. Clients of the Lock Service can enable a specific class of event tracing, such as entry/exit, or a collection of events (entry/exit and parameter tracing).

The level of tracing is controlled by the value of a single variable that is exported by the Lock Service:

```
unsigned long lock_traceMask
```

The Encina Trace Facility defines a number of ″bit constants″ that when set in the above variable cause a specific form of tracing to occur. For example, the `TRACE_ENTRY` constant enables entry/exit tracing of Lock Service interface functions. The Lock Service also defines Lock Service-specific bit constants for important classes of internal events.

The following section describes the different methods for setting the value of the above variable to obtain the required level of tracing. It is followed by a description of the levels of tracing supported by the Lock Service.

Global trace levels

The Lock Service supports the global trace levels defined by the Encina Trace Facility. These levels are as follows:

- `TRACE_ENTRY` — traces the entry and exit of functions that make up the Lock Service interface.
- `TRACE_PARAM` — traces the parameters passed to the interface functions.
- `TRACE_EVENT` — traces all events in the Lock Service. This trace class outputs large amounts of information. For this reason, it is not always recommended.

In addition, the following two trace constants are provided for enabling *all* or *no* tracing respectively:

- `TRACE_GLOBAL`
- `TRACE_NONE`

Lock Service trace levels

The Lock Service provides no specific trace levels.

State dump

The Lock Service provides the **lock_DumpState** function to dump the state of the Lock Service at an application.

Chapter 8. Log Service (LOG)

Log Service overview

The Encina Toolkit Log Service (LOG) provides permanent storage and retrieval of recovery logs used by other components of the Toolkit Server Core. LOG is also used to manage recovery logs and other sequentially written data for Toolkit applications. This chapter is organized as follows:

- The remainder of this section explains the basic concepts necessary to understanding the LOG interface.
- "Data types" on page 127 describes the LOG data types and functions that manipulate these data types.
- "Opening and closing log files" on page 128 describes the functions used to establish or relinquish access to log files.
- "Log administration" on page 129 describes the functions used to create and delete log files and volumes, and to determine their status.
- "Write operations" on page 129 describes the functions used to add records to a log file and to ensure that they become permanent.
- "Read operations" on page 129 describes the functions used to retrieve individual records or groups of related records from a log file.
- "Restart record operations" on page 130 describes the functions used to write and retrieve the contents of a log file's restart record.
- "Archive management" on page 130 describes the functions used by clients to control archiving within log files.
- "Archive filter functions" on page 131 describes the filter function variables, predicates, and operators in addition to a function to set the filter function.
- "Configuration constants" on page 132 describes the constants exported by the Log Service for configuration. Each description provides a brief explanation of the constant's meaning and its value.
- "Diagnostics" on page 133 documents the diagnostic support provide by the Log Service.

Conceptual introduction

LOG provides permanent storage for log files. Log files are efficient stable storage abstractions on top of which other Toolkit components and Toolkit applications can build more complex recoverable storage objects. A log file stores recovery data that allows its user to recover from application failures, system crashes, and storage media failures.

System administration of the LOG is done using Enconsole and the **tkadmin** commands, enabling administrators to create and manage log files and the

physical storage that they occupy. See *Encina Administration Guide Volume 1: Basic Administration* for more information. Also, like other Encina Toolkit components, LOG provides a set of administrative remote procedure call (RPC) interfaces for administration. The RPC interfaces for LOG administration are described in "Appendix C. Administrative RPC interfaces for the Encina Toolkit" on page 251 of this document.

CAUTION:
LOG functions cannot tolerate thread cancellation. Callers of LOG functions are expected to disable thread cancellation before making a call and restore thread cancellation after the call returns.

Log files

LOG stores data in multiple, logically-independent data repositories called log files. A *log file* is a record-structured sequential file. Each log record is identified by a *log sequence number* (*LSN*), which is assigned when the record is written. LSNs are unique within a log file and increase monotonically in the order records are written. An LSN serves as an address by which a record may be read from the log file, and allows the record to be associated with other client data.

Each log file has its own name (or identifier) and data contents. Log file names must be unique to a specific Encina server and host. A single client can have more than one log file open concurrently; however, a log file can be opened by only *one* client at a time.

Restart record

Each log file includes a single atomically updated record called the *restart record*. The restart record can be used to store data that allows the client to simplify and optimize its recovery, for instance, it may contain the location of a client checkpoint in the log file.

Log records and log sequence numbers

A log file contains an arbitrary number of *log records*. When the client writes a new log record, it is appended to the head of the log file. The LSN is assigned to each log record when it is written. After being written, log records cannot be updated. A log record consists of a record header, used by the LOG to control reading and archiving, and a record body, which is not interpreted by the LOG. The record header consists of an LSN, zero or more links to related log records, and a record type. The record body contains client-supplied data.

An LSN identifies a specific record. An LSN may also be used to refer to a specific point within a log file. The LOG provides functions for comparing LSNs so clients may determine the order in which records were written. It is not possible to predict the log sequence number that will be assigned to the next record. Log sequence numbers from different log files are unrelated.

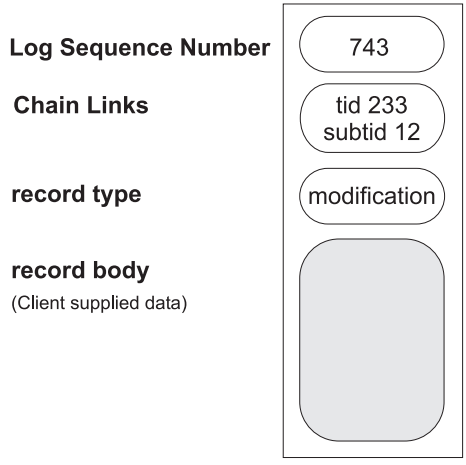

Figure 13. A log record

Chain links: When writing log records, related records (such as log records for the same transaction) are linked into chains, where each such log record contains a reference to the previous related log record. This chaining of log records enables related records to be easily located when necessary, such as when replaying log records during a server restart or when rolling back the changes made by a transaction.

A log record can be included in zero or more chains. The record header contains a chain link for each chain to which the record belongs. The chain link consists of a *chain identifier* and data LOG uses to find adjacent records in the chain. A chain identifier is a client-specified variable-length sequence of bytes. A client includes a record in a chain by specifying a chain identifier when it writes the record. The client can read records in a chain without knowing their specific LSNs. For further information see "Reading log records" on page 124.

Record types: The client assigns a record type to each log record when the record is written. LOG uses the record type only when filtering records during archiving (provided that the client has defined the filter function using the **RecordType** variable). The record type is returned when a record is read, allowing the client to use it to help interpret the contents of the record.

Record body: The *body* of a log record is not interpreted by the LOG.

Archives and log file archives

LOG maintains two different types of archives of previously written log records. An *online archive* is used to reclaim space in a log file by discarding records that are no longer needed. An online archive can be thought of as a way of compressing a log volume, and is used when space is low. The other

type of archive, an *offline archive*, is used to migrate log records to permanent, offline storage. This type of archive is used administratively to recover from media and general hardware problems.

An online *log file archive* is associated with each log volume, and is used to free space in that volume by discarding log records that are no longer needed for recovery from a crash. A *filter function* is associated with every log file archive. The filter function is used to preen the log records to be preserved in that archive, discarding log records that are no longer needed by the server. An online archive can be thought of as a compressed version of log records that must be preserved in order to successfully restart a server. "Archive filter functions" on page 131 describes archive filter functions in detail.

Typically, an online archive only contains a fraction of the records originally written to a log file. An online archive only contains the log records that allow rebuilding dirty pages in the client's buffer pool, rolling back uncommitted transactions, and rolling forward from the last media dump.

A server can explicitly set the archive's tail, which is the LSN of the oldest record needed in the archive. LOG discards the records in an archive that are older than the archive tail and reclaims the physical storage they occupy. Servers can avoid the overhead of archiving by periodically setting the archive tail. Moving the archive tail permits LOG to discard older records rather than moving them to the log file archives.

Writing log records

Buffering and forcing

Log records are buffered in memory until they are written to permanent storage. When a log record is *buffered*, it is assigned an LSN greater than that of any previous log record in the log file. It is stored in memory in a manner that guarantees it will be forced to the log file in the order in which it was written.

When a log record is *forced*, it and all previous records from the same log file are written to permanent storage in the order that they were written. Records are forced as log buffers fill, or at the explicit request of the client. Forcing a log record implicitly forces all previously flushed records in other log files that are stored in the same log volume.

Chains

A *chain* is a sequence of related log records in a log file that are not necessarily written contiguously, but are often read sequentially. Figure 14 on page 123 shows how chains work. A record can only be included in a chain when the record is written. Chains are used by cursor reads (see "Cursors" on page 124) and can be used to define filter functions (see "Archive filter functions" on page 131).

Each chain has an identifier which is a server-specific, variable-length sequence of bytes. For example, a server can use a chain to link together all records belonging to a transaction, deriving the chain identifier from the transaction identifier. Each log record is linked into each chain for which the server provides a chain identifier when the record is written.

A server opens a chain by writing the first record in it. A chain is closed when a record with the chain close flag set is written to the chain. Chains stay open across system crashes, and must eventually be closed by the server that is using the log.

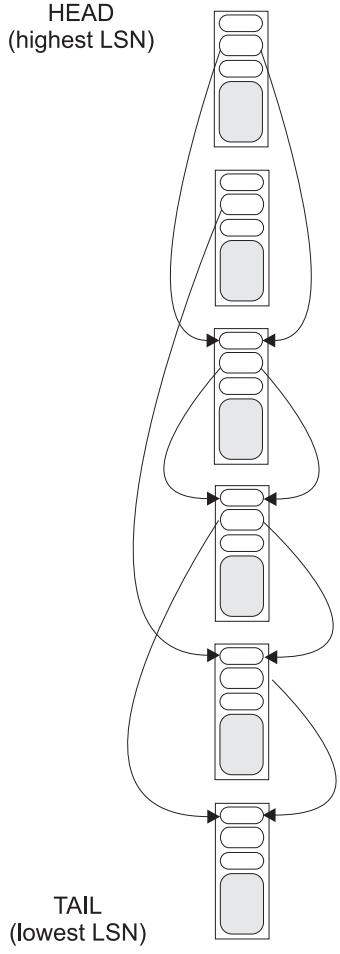

Figure 14. Log records linked by chains

Reading log records

A server that uses LOG can read an individual log record at random by specifying its LSN, or it can read a sequence of log records using a cursor. Normally, reading a sequence of records with a cursor is more efficient than reading the same records individually. Servers read log data from an online archive and its associated log file rather than just from the log file.

Cursors

Cursors are used to sequentially read records in an archive. A range of LSNs must be specified when a cursor is opened on an archive. When it is opened, the cursor can be specified to read all records in the range of LSNs, or only the records in the range of LSNs that belong to one or more specified chains.

The group of log records specified when the cursor is opened is called a *cursor set*. The cursor set changes dynamically to reflect records being written, chains being opened or closed, and archives being filtered and truncated.

If a new set of log records is written using a chain identifier previously used to define a chain which has since been closed, LOG will create a new chain for this new set of records only. Records belonging to the earlier, closed chain will not be part of the cursor set of a cursor which specifies this new instance of the chain identifier. However, the earlier records can still be read individually using their LSNs.

A database example

This section describes how LOG can be used by a database manager. This example describes the process of configuring the database log files, and the interactions between the database manager and LOG.

Administrative configuration

Before any server can use LOG, LOG must be configured by a system administrator using Enconsole and the **tkadmin** configuration commands described in *Encina Administration Guide Volume 1: Basic Administration*. A log volume must be created for storing log files. Log volumes should be large enough so that the need to do online archiving is minimized.

Because log files are stored in the logical volumes managed by the Encina Toolkit Volume Service (or a system-specific volume service), log volumes in a production environment should be mirrored. As explained in "Chapter 13. Volume Service (VOL)" on page 207, mirroring a volume means that multiple copies of the volume are kept. These copies of the log volume are written and updated in parallel, so that their contents are always identical. Mirroring provides protection against hardware failures.

Application initialization

When the database manager is started for the first time (*cold started*), it creates a log file in a log volume and defines an online archive for that log file. The online archive, used for recovery from crashes, has an associated filter function that describes the records that may be needed for crash recovery at any point in time. An application uses the **log_SetFilterFunction** function to set the filter function. If the log is running short of space, it uses this filter function to decide which records may safely be discarded from online storage (that is, from the log volume). This process is called *online archiving* or *compression*. For example, the filter can be defined to pass only records belonging to open chains (that is, active transactions) or to pass all records with an LSN greater than a specific value (that is, the LSN of the oldest record for any dirty page).

If the database intends to support media recovery by using the log file to roll forward from a media dump, the application must create (or verify the existence of) an offline archive when the application initializes. This offline archive is a specified area in the file system where log records that are no longer needed for recovery from server failures are moved. These offline archives should periodically be moved to tape or other offline storage media, both to permanently preserve them and to reclaim the space in the file system that they consume.

Logging during database operation

The database manager uses the **log_WriteRecord** function to write log records. Most records do not need to be immediately forced to disk, so the database can optimize log writes by passing the value `LOG_BUFFER` as the value of the *writeMode* parameter to the **log_WriteRecord** call. Records that need to be forced to disk, such as commit protocol log records, should be written with the *writeMode* parameter set to `LOG_FORCE`, which will ensure that the record is written to permanent storage when the **log_WriteRecord** call returns successfully. Rather than synchronously forcing the record with the **log_WriteRecord** function, the database can alternately write the record with the *writeMode* parameter set to `LOG_BUFFER` and later force it to disk by calling the **log_CheckLsn** function. The database could also wait for other **log_WriteRecord** calls to force its records to disk, and calling the **log_HighStableLSN** function to determine when the record is on permanent storage.

Normally, an application links all records belonging to a transaction (or alternatively, all of a transaction's undo records) into a chain whose identifier maps to the transaction ID. The chain can be used to selectively read the transaction's log records using a cursor (for example, to abort the transaction) and can be used by the filter function associated with the crash recovery log file archive to retain only the log records of active transactions.

The database is responsible for moving the archive tail forward, in each of its log file archives, from time to time to allow the Log Service to discard records and free the storage space they occupy. To do this, the database calls the **log_SetArchiveTail** function with the new archive tail being the LSN of the oldest record in the archive that the client still needs. For example, in the offline archive, records older than the oldest media dump may be discarded. In the online archive, the database can set the tail to discard records too old to belong to active transactions or reflect updates to dirty pages.

Most applications periodically write checkpoint records to the log file, and start their recovery at that checkpoint. After writing checkpoint records and forcing them to permanent storage, the database can save the LSN of the first checkpoint record in the log file's restart record to determine where in the log to start its recovery.

Recovery

After a system crash, the database recovers its state using log records it wrote before the crash. This example explains the workings of a conventional three-pass recovery algorithm to show how the database uses its log file.

After opening its log file, the database reads its restart record which contains the LSN of the most recent successfully written checkpoint. The database performs its log analysis pass by reading the online archive from the checkpoint forward to the end of the log. During this pass, the database builds a list of active transactions and dirty pages at the time of the crash. To do this, it opens a cursor on the online archive ranging from the checkpoint LSN to the head of the log file, and uses it to read the checkpoint and all subsequent records. Among other things, the analysis pass determines the oldest dirty page update LSN, and the LSN of the oldest record belonging to each active transaction.

Next, the database performs its forward recovery pass, during which it redoes updates to dirty pages that may not have been written to disk. To do this, the database opens a cursor on the online archive ranging from the oldest dirty page update LSN (determined during the analysis pass) forward to the head of the log. It reads the records in the cursor set and redoes the dirty page updates.

In the final pass, the database aborts (or undoes the changes of) all transactions that were active at the time of the crash. This can be done in any of three ways:

- For each active transaction, open a cursor on the crash recovery archive using the transaction's chain, ranging from the head of the online archive (or better, the transaction's newest record LSN) to the tail of the log. Read each record in the cursor set and perform all undo operations.

- Open a cursor on the crash recovery archive using all open transaction chains, ranging from the head of the online archive (or better, the newest LSN of any active transaction) to the tail of the log. Read each record in the cursor set and perform all undo operations.
- Open a single cursor on the crash recovery archive that returns all records ranging from the head of the online archive (or better, the newest LSN of any active transaction) to the tail of the log. Read each record and perform all undo operations for active transactions.

The first alternative is how the database would abort a single transaction. It returns only the required records, but requires one pass over the online archive for each transaction being aborted. The second alternative performs only one pass, and only reads records logged on behalf of active transactions. The third alternative performs one pass, but reads all records.

LOG header files and libraries

Header files

The **log.h** header file exports the LOG function prototypes, LOG data types, and LOG macros. A client calling any LOG function or using any LOG data type must include this header file.

Libraries

The Encina LOG functions are contained in the **EncClient** libraries. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Data types

The LOG interface provides data types for each principal abstraction or argument used in the interface. Where applicable, the LOG interface provides functions and macros that manipulate these date types and symbolic representations for values they may take.

The Log Service exports data types for the following abstractions:

- *Archive identifier* — the name by which an archive is known to the LOG Service. The exported type is **log_archive_t**.
- *Archive type* — an enumerated type identifying whether a specified archive is online or offline. The exported type is **log_archiveType_t**.
- *Archive upcall* — a function called by the LOG Service when archiving is imminent. The exported type is **log_archiveCallback_t**.
- *Chain identifier* — the client-defined identifier of a chain of log records. The exported type is **log_chain_t**.
- *Cursor handle* — the identifier that specifies which open cursor a cursor function is to operate upon. The exported type is **log_cursor_t**.

- *Log file handle* — the identifier of an open log file. The exported type is **log_handle_t**.
- *Log file status* — information about a log file. The exported type is **log_fileStat_t**.
- *Log sequence number* — an address of a log record in a log file. The exported type is **log_lsn_t**. (See "Log sequence number functions" for a list of related functions.)
- *Log volume status* — information about a log volume. The exported type is **log_volStat_t**.
- *Record type* — a client-defined value that differentiates various kinds of records written by that client. The exported type is **log_recordType_t**.
- *Status code* — indicates whether the requested operation succeeded, or how it failed. The exported type is **log_status_t**.

The LOG interface also provides the following modes for arguments that affect the behavior of the interface functions. The meaning of these modes will be reviewed in the description of the interface functions that use them.

- *Write mode* — specifies whether a record is to be buffered or forced by the write call. The exported type is **log_writeMode_t**.
- *Archive tail inclusion mode* — specifies whether the log record with LSN equal to the archive tail should be kept or discarded from the archive by the **log_SetArchiveTail** function call. The exported type is **log_tailInclusive_t**.

Log sequence number functions

The following functions are used to compare and copy log sequence numbers:

- **log_CopyLsn**
- **log_LsnEqual**
- **log_LsnGreaterThan**
- **log_LsnGreaterThanOrEqualTo**
- **log_LsnLessThan**
- **log_LsnLessThanOrEqualTo**

Opening and closing log files

The **log_Init** function initializes the LOG for use by a server and (if explicitly called) must be called before other LOG functions. The first call to the **log_Open** function implicitly calls the **log_Init** function if the client has not already done so.

The **log_Open** function must be called once for each log file to be accessed before any other operations can be performed on that log file. A successful call returns a log file handle which must be used to access the log file in other LOG calls.

The **log_Close** function should be called when a log file will no longer be accessed or when a log file's fatal error flag needs to be reset.

Log administration

The functions listed below create, initialize, check the status of, and delete log files; they also initialize, check the status of, and restore the logical volumes on which they are located. Functions to create logical volumes are provided in the VOL component of the Encina Toolkit Server Core.

The following functions all take the name of a log file as an argument, with the exception of the **log_StatFile** function, which takes a handle to an active log file:

- **log_CreateLogFile**
- **log_DeleteLogFile**
- **log_InitVol**
- **log_InitVolWithExtentSize**
- **log_RestoreVol**
- **log_StatFile**
- **log_StatVol**

Write operations

The **log_WriteRecord** function allows a server to write records to a log file. The **log_CheckLsn** allows the server to make sure a previously written record has been written to permanent storage. The **log_HighStableLSN** function allows a server to determine if a specific record has been forced to disk.

Read operations

The LOG interface provides a server with two methods for retrieving log records. A server can retrieve individual records by using the **log_ReadRecord** function, or retrieve groups of records using cursors.

A particular log record can be retrieved by specifying the log sequence number returned by the **log_WriteRecord** function. Records may be read before they have been forced to permanent storage, but it may not be possible to read the same record again after a system failure.

Since a single log file can be associated with more than one log file archive, read operations specify the archive name (which consists of the log file and a log file archive) from which the read will occur.

Cursors permit convenient and efficient retrieval of groups of related records. A cursor can be defined to traverse all log records in an archive, or only those log records associated with a client-specified set of open chains. The following functions are provided for cursors.

- The **log_OpenCursor** function defines a set of records, known as a cursor set, in an archive that can be read by subsequent **log_ReadCursor** calls.
- The **log_SetCursorAttribute** function modifies the cursor attributes. In particular, it can be used to specify whether the cursor should be advanced on partial reads (which occur if the buffer is not big enough to fit the entire record). The **log_GetCursorAttribute** function can be used to determine the current cursor attributes.
- The **log_ReadCursor** function sequentially reads records from a cursor set.
- The **log_CloseCursor** function closes a cursor and releases the system resources associated with it.

Restart record operations

Each log file contains a *single* restart record. Unlike other log records, the restart record can be updated. The restart record is typically used by a server to store the LSN of a checkpoint log record that is used to begin recovery after a system failure. The LOG provides the **log_WriteRestart** function to atomically overwrite the restart record, and provides the **log_ReadRestart** function to read the restart record.

Archive management

The **log_CreateArchive** and **log_DeleteArchive** functions, respectively, create and delete archive log file archives on an archive device. An archive device is associated with a logical volume when that volume is initialized. If an archive file required by the LOG Service has been moved offline, that archive file must be restored to the archive device so that the LOG Service can use that archive file. In this case, the **log_EnableArchFile** function must be called to notify the LOG Service that the requested archive files are now available.

During archiving, log records are selectively moved from the log file to zero or more log file archives which are defined when the log file was configured. Archiving is done automatically by the LOG. However, the client can explicitly control archiving. Clients can explicitly move log records from the log file to log file archives using the **log_FlushToArchives** function. Clients can truncate the archive with the **log_SetArchiveTail** function, causing old

records to be discarded before the LOG begins its archiving process. Clients can determine the archive file that contains a specific LSN with the **log_LocateLsn** function.

Archive filter functions

Each log file archive has a filter function which is defined to be TRUE when the log file archive is configured. Having the filter function set to TRUE causes all records to be eligible for archiving. Servers using the Log Service can set the filter functions of their log file archives with the **log_SetFilterFunction** function. The filter function allows the LOG to discard log records that are not needed in an archive and thereby save storage. Different filter functions can be defined for each log file archive, allowing a log file to keep different sets of log records in different log file archives.

The **log_SetRestartPoint** function allows a server to set the RestartPoint archive filter function variable. This variable is only used to define archive filter functions.

Note: Filtering out log records that belong to open chains may cause back pointer LSN values to be incorrect, for example, *prevChainLsn[N]* in the **log_ReadRecord** function.

Filter function grammar

A filter function is a Boolean expression composed of predefined variables.

```
<filter_fn> := <null> | { <or_expr> <null> }
<or_expr> :=<filter_clause> { <or_tok> <filter_clause> }
<filter_clause>:= <restartPtClause> | <openChainClause> | recordTypeClause>
                                  | <TrueClause> | <FalseClause>
<restartPtClause> := LSN>=RestartPoint
<openChainClause> := AnyOpenChain()
<recordTypeClause> := RecordType=<number>
<TrueClause>:= TRUE
<FalseClause> := FALSE
<or_tok> := |
```

Example filter functions

Some sample filter functions are the following:

- **LSNGreaterThanRestartPt | AnyOpenChain**
- **AnyOpenChain | RecordType=5**
- **LSNGreaterThanRestartPt | AnyOpenChain | RecordType=5**
- **RecordType=5 | RecordType=3**

Configuration constants

Table 5 shows the symbolic constants defined by LOG, showing the value of the constant as configured in the current version of the LOG.

Table 5. Log Service configuration constants

Log Service constant	Values of Log Service constants	Description
LOG_CFG_MAX_ARCHIVE_LENGTH	256	Maximum length of variables of type **log_archive_t** used in log configuration. This includes the NULL character that signifies the end of the string.
LOG_MAX_ARCHIVENAME_LENGTH	256	Maximum length of variables of type **log_archive_t** including the NULL character that signifies the end of the string.
LOG_MAX_ARCHIVE_ID_LENGTH	256	Maximum length of variables of type **log_archive_t** including the NULL character that signifies the end of the string.
LOG_MAX_CHAIN_ID_LENGTH	64	Maximum length of variables of type **log_chain_t**.
LOG_MAX_FILE_NAME_LENGTH	384	Maximum length of variables of type **log_file_t** including the NULL character that signifies the end of the string.
LOG_MAX_FILTER_LENGTH	512	Maximum length of the names of filter functions, including the NULL character that signifies the end of the string.
LOG_MAX_RECORD_CHAINIDS	2	Maximum number of chains that a log record can belong to.
LOG_MAX_RESTART_RECORD_LENGTH	512	Maximum length of a restart record.
LOG_RECORD_TYPE_MAX	2^{16}	Maximum number of different log record types that a client can write.

Diagnostics

This section documents the diagnostic support provided by LOG. This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal-errors) during the use of LOG, and informative messages that trace calls on LOG (tracing).

Directing output

All Encina trace and error output can be directed to user-specified files and to the main memory ring buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default, all diagnostic output (output of all trace classes) is sent to the ring buffer. In addition, the fatal, error, and audit trace class output is directed by default to the standard error stream of the server. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Error messages

LOG generates fatal error messages and terminates when detecting an unrecoverable error condition. LOG generates run-time warning messages for notification of nonfatal error conditions, such as when it is running out of space on the log volume.

Trace and state dump information

Encina trace output can be generated to follow the execution path of LOG. All LOG functions are capable of tracing their entry, parameters, and exit. In addition, important events during the execution of a LOG interface function can be traced. Clients of the Log Service can enable a specific class of event tracing, such as entry/exit tracing, or a collection of events (entry/exit and parameter tracing).

The level of tracing is controlled by the value of the following variables that are exported by LOG:

```
unsigned long log_traceMask
```

The Encina Trace Facility defines a number of "bit constants" that, when set in the above variables, cause a specific form of tracing to occur. For example, the `TRACE_ENTRY` constant enables entry/exit tracing of LOG functions.

Global trace levels

LOG currently supports the global trace levels defined by the Encina Trace Facility:

- `TRACE_ENTRY` — traces the entry and exit of LOG functions.
- `TRACE_PARAM` — traces the parameters passed to the LOG functions.

- TRACE_EVENT — traces all events in the Log Service.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- TRACE_GLOBAL
- TRACE_NONE

Log Service trace levels

Log supports many different levels of tracing. The types of situations for which tracing can be activated, and the hexadecimal flags associated with them, are the following:

Volume Manager tracing:

0x00000100 - LOG_ARCH_DEV_DEBUG
0x00000200 - LOG_VOL_BUF_MGR_DEBUG
0x00000400 - LOG_VOL_EXTENT_DEBUG
0x00000800 - LOG_VOL_READ_DEBUG
0x00001000 - LOG_VOL_WRITE_DEBUG
0x00002000 - LOG_VOL_DEBUG

File Manager Tracing:

0x00010000 - LOG_FM_METAMEM
0x00020000 - LOG_FM_MEM
0x00040000 - LOG_FM_WRITEPATH
0x00080000 - LOG_FM_CHECKPOINT
0x00100000 - LOG_FM_RECOVERY
0x00200000 - LOG_FM_FILEMGMT
0x00400000 - LOG_FM_READPATH
0x00800000 - LOG_FM_METADATA
0x01000000 - LOG_FM_ARCHIVE
0x02000000 - LOG_FM_ADMIN

Severity level flags:

0x10000000 - LOG_VOL_MGR_EVENT
0x20000000 - LOG_VOL_MGR_SIG_EVENT
0x40000000 - LOG_FM_EVENT
0x80000000 - LOG_FM_SIG_EVENT

Enabling tracing for the log product or for one of its components at any of these levels generates execution event trace output. Higher levels of tracing provide increasingly greater volumes of execution status information.

State dump

The **log_DumpState** function dumps the state of LOG and its components.

Chapter 9. Transaction State Log (tranLog)

Introduction

The Encina Transaction State Log (tranLog) programming interface simplifies the creation of a recovery service for applications that use multiple recoverable servers. Ordinarily, recoverable Toolkit servers (that is, servers not running under the Encina Monitor) are required to maintain individual log files for transaction state information. Using the tranLog interface allows you to centralize all recovery operations on a single server.

The server that manages recovery is called a tranLog server. Other recoverable servers that use the recovery service are called tranLog clients. This arrangement simplifies programming, because only one server needs to go though the steps required to initialize the Log Service (LOG) and the Distributed Transaction Service (TRAN). In addition, the tranLog interface reduces the program's administrative overhead by maintaining only one log file, as shown in Figure 15 on page 138

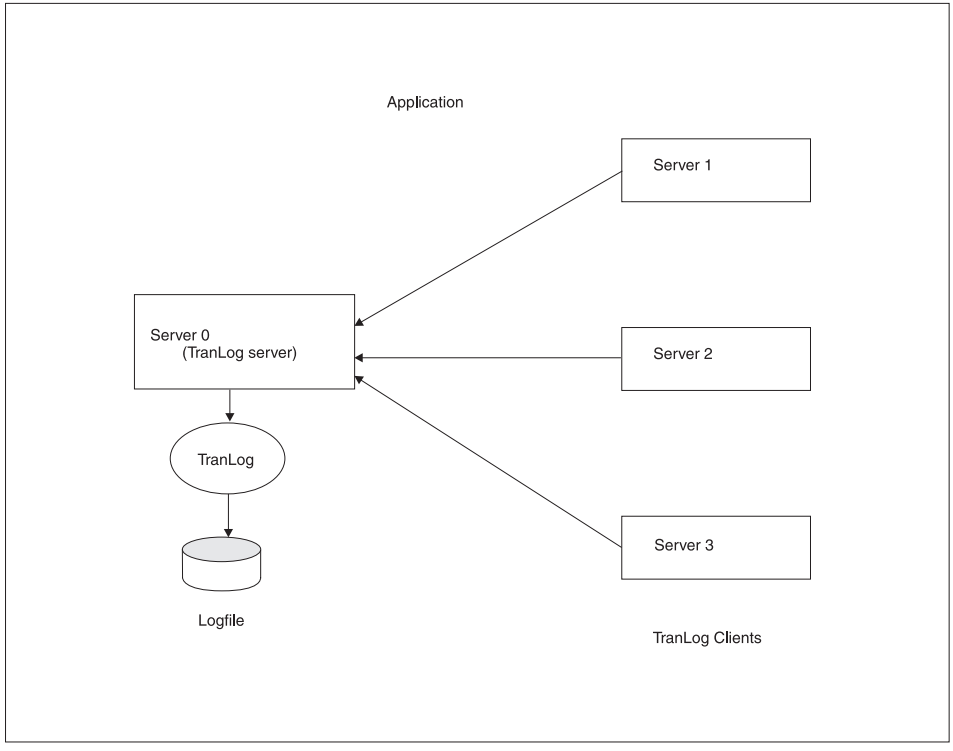

Figure 15. The tranLog Recovery Service

Once the server and clients are initialized, all logging of record data is done automatically.

Writing tranLog servers

A tranLog server logs recovery information for itself and other servers within an application. It can be one of the servers already in place for your application; you do not need to create a separate server for the tranLog interface. The tranLog server must include the **tranLog/tranLog_log.h** header file and must initialize LOG and TRAN.

The tranLog service is initialized when the application calls the **tranLog_Init** function. The **tranLog_Init** function takes one parameter, which specifies the name of the log file for storing the record data. This file must already have been created, either by the application program or by using the**tkadmin** interface, and must be both readable and writable by the server. The log file name must appear in the following format: *volumeName/fileName*, where *fileName* is the log file name, and *volumeName* is the name of the logical volume where the primary segment of the log file resides. For example, **logvol/file1** specifies**file1** on a volume called **logvol**.

The tranLog server also must call the **tranLog_ProvideService** function, which exports the service for use by remote clients. When a tranLog server receives a client call, **tranLog_ProvideService** invokes the specified callback function, providing the binding handle for the remote procedure call (RPC) and the identifier of the tranLog client. The callback returns 1 (one) to authorize the call, or returns 0 (zero) to reject it. These parameters are specified in the **tranLog_authorizationFn_t** data type. An example of this callback is shown in Figure 16.

```
int authorization_function (void **handle,
       encina_unsignedInt32_t owner)
 {
  /* check owner if authorization checking is being done */
  /* **handle is the binding handle for the RPC that this function is to */
  /* authorize owner is the owner identifier of the tranLog client */
  . . .
  /* return 1 if the call is permitted */
  return 1;
 }
```

Figure 16. A tranLog server callback

Figure 17 shows the calls required for the tranLog server.

```
#include <tranLog/tranLog_log.h>

char *logfile;

int main(void)
{
/* initialize TRAN   */

/* initialize LOG  */
. . .


tranLog_Init (logfile);

tranLog_ProvideService (authorization_function);

}
```

Figure 17. A tranLog server

Writing tranLog clients

To establish a relationship with a tranLog server, a client must call the **tranLog_UseService** function. This function enables the client to become a recoverable server with the recovery information logged by the tranLog server. Before making this call, the client must obtain an RPC binding handle to the server (for example, by using the **trdce_BindingImport** function). The tranLog client must include the **tranLog/tranLog_tran.h**header file.

The call to the **tranLog_UseService** function takes four parameters: a pointer to the server handle, the name of the client, a number that identifies the client (used by the server as a log stream identifier), and a pointer to a function that is called if there is a failure.

Next, the client must call the **tranLog_LocalRecovery** function, which registers the upcalls necessary for recovery used by TRAN at different points in the transaction. The upcalls registered by **tranLog_LocalRecovery** are the same as the upcalls registered by the **tran_RecInit** function, including pointers to the following functions:

- A function TRAN calls during the prepare phase of a transaction family
- A function TRAN calls when a transaction commits
- A function TRAN calls when the transaction aborts
- A function TRAN calls when the transaction has finished
- A function TRAN calls when the transaction becomes active or inactive

These upcalls are seldom used; they must not do any recovery-related work. Parameters for upcalls that are not required are specified as NULL in the call to **tranLog_LocalRecovery**. Parameters passed to these upcalls must not be altered.

The **tranLog_LocalRecovery** function must appear in the application between the calls to the **tran_Init** function and the **tran_Ready** function. After the call to **tranLog_LocalRecovery**, the client has done all that is required—the record data is then maintained by tranLog.

Figure 18 on page 141 shows how a recoverable server becomes a tranLog client:

```
#include <tranLog/tranLog_tran.h>
void *serverhandle;
char *ownerName;
encina_unsignedInt32_t *ID;
void (*failureHandlerFn) (void **handleP, void *exception);
int main(void)
{
. . .
tranLog_UseService(serverHandle, ownerName, ID, failureHandlerFn);
tranLog_LocalRecovery(NULL, NULL, NULL, NULL, NULL);
. . .
}
void failureHandlerFn(void **handleP, void *exception)
{
/* rebinding and failure recovery procedures */
/* **handleP is the handle to the tranLog server */
/* *exception describes the nature of the failure */
return;
}
```

Figure 18. A tranLog client

Chapter 10. Recovery Service (REC)

Recovery Service overview

The Encina Toolkit Server Core Recovery Service (REC) implements recovery for Toolkit applications. The Recovery Service chapter is organized as follows:

- The remainder of this section explains basic concepts and the Recovery Service model.
- "Data types" on page 154 describes the data types used by the Recovery Service.
- "Initialization and restart" on page 156 describes the functions used for initializing and restarting the Recovery Service.
- "Operation management" on page 159 describes the functions used for operation management.
- "Buffer management" on page 159 describes the functions used for buffer management.
- "Media recovery" on page 160 describes media-recovery functions.
- "Diagnostics" on page 161 describes the diagnostic support provided by the Recovery Service.

Recovery Service model

REC is a library that provides clients with recoverable storage for transactional Encina Toolkit applications. REC implements this recoverable storage by keeping track of all the client's updates. REC is built on top of the Toolkit Log Service (LOG) and the Toolkit Volume Manager (VOL). It is a layer between the client and VOL that keeps track of all updates to VOL, writing that information to a log file (maintained by LOG) using a write-ahead logging scheme. The place of REC in the Encina Toolkit is illustrated in Figure 19 on page 144.

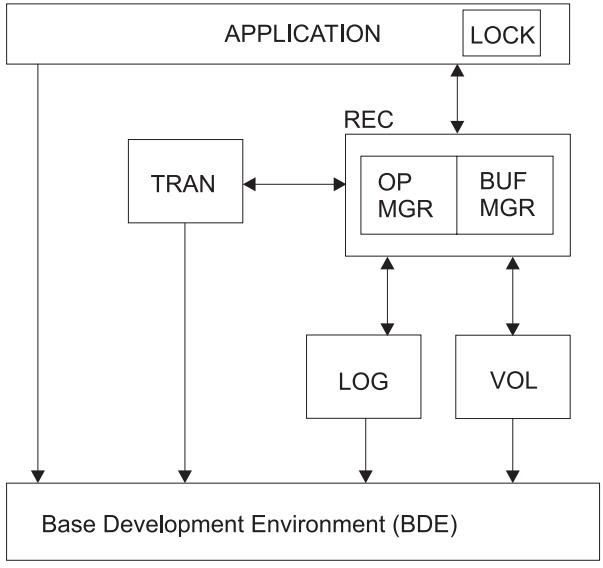

Figure 19. Recovery Service architecture

The recoverable storage provided by REC is organized in volumes, each of which is a collection of pages. REC provides a *buffer manager*, which coordinates access to the volumes maintained by VOL. REC's buffer manager provides the client with a consistent view of the volume.

Using the information stored in the log file, REC provides three recovery services: Abort Recovery, Crash Recovery, and Media Recovery:

- *Abort Recovery*: REC cooperates with the client's transaction manager to ensure that each transaction commits or aborts consistently everywhere in the network. This function is modeled closely on the requirements described in "Recovery Service interface" on page 46. This aspect of REC is not discussed further in this document.
- *Crash Recovery*: REC restores the database to a consistent state after a system failure.
- *Media Recovery*: The client can call REC to restore the contents of one or more volumes after a hardware failure.

REC provides an *operation manager* that exports the notion of an *operation* as a logically invertible set of page updates (that is, an update to recoverable storage whose effects can be undone without necessarily affecting the same set of pages).

CAUTION:
The DCE threading model permits the cancellation of threads. However, REC assumes its work will not be interrupted by such a cancellation. Calls to REC should only be made from threads with thread cancellation disabled. Should a thread executing a REC function be cancelled, the results are undefined.

Value vs. operation logging

REC provides two types of logging, *value* and *operation*, to track an application's updates to recoverable data.

With value logging, REC logs each change made to memory in REC, and when it redoes the application's modifications to its data, REC executes those same changes to the same pages. REC arranges to reobtain any locks required to protect these updates. The locks belong to the transaction IDs that were in effect when these locks were originally held, and the locks are held for the duration of the transactions. When undoing modifications to the memory, REC reverts those changes on the same pages in VOL.

With operation logging, REC logs information about operations. An operation is a set of modifications to pages that is associated with a logical method for undoing the operation. The undo procedure may not affect exactly the same pages modified during the operation, providing opportunity for the undo to be more efficient than the original data modifications. When redoing data changes, operation logging works the same way as does value logging. When using operation logging, the application can avoid locking large amounts of the data for the duration of the transaction. Instead, the application can transactionally lock a small part of the affected data and use an auxiliary mutual-exclusion mechanism on most of the affected data. The other mutual-exclusion mechanism need only have effect for the duration of the operation. Because operations have much finer grain control than transactions have, using them can substantially improve concurrency.

For example, consider an application that uses a recoverable B-Tree to store its data. If a transaction inserts a record into the tree, it might cause the tree to split. Splitting the tree moves multiple records and updates multiple keys. With value logging, the transaction holds locks on much of the B-Tree, including the root, for the duration of the transaction. When the application obtains locks for the duration of the transaction on all the pages affected by the B-Tree split, it severely limits access to the B-Tree. Furthermore, if the transaction aborts after this modification, the application will undo every write on every page changed, ultimately undoing the B-Tree split. While the application must remove the inserted record due to the abort, undoing the B-Tree split is unnecessary. Operation logging avoids the cost of locking an excessive amount of the data and undoing every individual modification to the data.

Using operation logging in the above B-Tree example, the transaction holds a transactional lock on the new record's key, and for much of the B-Tree it uses an alternative mechanism to exclude other transactions from reading or writing during the operation. Operations typically commit much earlier than transactions, so the alternative mechanism for mutual exclusion is in effect for much less time than the lock held for the duration of the transaction. If the transaction aborts after this modification, the undo method for the operation removes the new record and leaves the B-Tree split, which is efficient use of the B-Tree.

Value logging is a good choice for an application that uses an array to store information. Consider a merchandise database where each product is identified by an integer, and let the integer index an array in which each elements records the number available for each item. As operation logging was clearly the best choice for the B-Tree example, value logging is more appropriate for a merchandise server built around the array just described. To place an order with value logging, a transaction obtains a lock on an array element corresponding to some garment and writes a value into the appropriate location on the page REC has obtained from VOL. When the transaction ends, the application drops the lock. With operation logging, a transaction obtains a lock on an array element, and the operation arranges to prevent other transactions from accessing the location on the page corresponding to the array element. Then the operation writes a value into the location and drops whatever lock it has on the location. At this point the transaction drops its lock on the array element.

Value logging and operation logging used in the merchandise server described above provide the same amount of concurrency because the recoverable array only supports reading and writing elements, which means operation locks would be held for the same amount of time as the transaction locks. However, operation logging does increase costs compared to value logging: beginning and ending operations, the costs of the alternative mechanism for mutual exclusion, and extra data logging.

Operations

A primary abstraction exported by REC is the *operation*: an explicitly demarcated, logically invertible set of page updates. The inverse of an operation need not modify the same data as during the forward execution.

In a transactional system, each transaction performs actions. The system guarantees that each transaction is atomic, consistent, isolated, and durable (ACID). The actions a transaction can perform are system-dependent. REC allows the system designer to combine several actions into an operation and export the operation, making it available to the transaction designers. Operations allow greater concurrency without violating the ACID properties.

For example, suppose that a client includes many transactions, each of which has to increment a common counter. Each transaction can read the counter, compute the new value and then write the counter back. In order to guarantee the ACID properties, each transaction may be required to obtain a write lock on the counter. The transaction would hold the write lock for the duration of the transaction, which means that at most one such transaction can be active at any given time.

Using operations, the system designer can implement a counter increment operation as follows: The operation obtains a write lock on the counter, reads the value of the counter, computes the new value and writes it back. It then releases the write lock on the counter. In addition, another type of lock called an *increment lock* can be used. Increment locks are defined to conflict with read or write locks but not with other increment locks. Before a transaction increments a counter (by calling the increment operation) it obtains an increment lock on the counter.

Two transactions may increment the same counter concurrently, using the counter increment operation, without violating the ACID requirements.

When a transaction is aborted, all its actions must be undone. If the transaction obtained a write lock on the counter it may log the old value of the counter (*value logging*). The transaction can be undone by restoring the old value of the counter. Note that value logging requires the write lock to be held until the transaction is finished (committed or aborted). However, when an application is using operations logging, it may not be correct to simply restore the counters modified to their prior state when undoing an operation. For example, suppose a transaction incremented a counter, whose value was 7. When the transaction is aborted the value of the counter may be 10 and it may be incorrect to restore the counter value to 7. Rather, the value has to be decremented. When system designers define an operation, they must also define an *undo operation* which logically reverses the effect of the operation.

REC allows the client to define arbitrary operations by allowing an operation to call other operations with arbitrary nesting. When an operation, *Op1*, calls another operation, *Op2*, *Op1* is called the *master* of *Op2* and *Op2* a *slave* of *Op1*. Operations are thus organized into a hierarchy with application-level transactions at the top of the hierarchy and page updates at the bottom. An operation hierarchy is illustrated in Figure 20 on page 148.

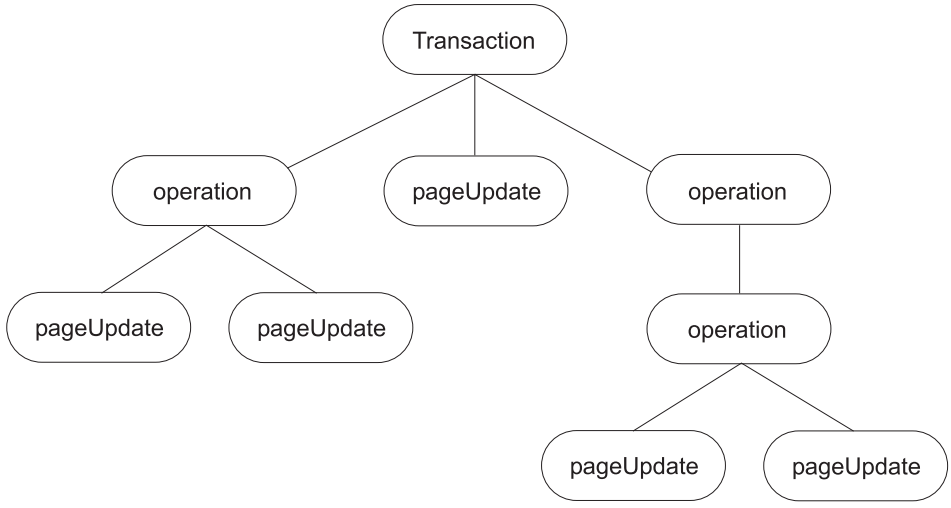

Figure 20. Operation hierarchy

When a client begins an operation, it supplies the identifier of an operation that will be the new operation's master. This specifies the hierarchy (or *family*) to which the new operation belongs and the operation's place in that hierarchy. REC associates with each transaction a special operation called a *root operation*. All the operations executed on behalf of a transaction are descendants of this root operation.

During the execution of an operation the operation may acquire locks on its behalf and on behalf of its ancestors. When the operation commits, it releases all the locks acquired on its behalf (by the operation or by its descendants). When an operation is aborted, REC aborts any of the operations for active slaves and initiates an undo operation for each slave operation that has committed. The undo operation is started as a slave of the master of the original operation (that is, it is the sibling of the operation it is undoing). Undo operations are not allowed to acquire locks on behalf of any of their ancestors. Therefore, the master operation must hold sufficient locks to undo all of its slave operations.

When an operation commits, it provides REC with a lock description and an operation description. The operation description is the description the client will receive from REC when the operation must be undone and should be self contained.

At the bottom of the operation hierarchy lie the page updates that modify the contents of the recovery volumes. A page update is a special type of operation that is restricted to a single page. In addition, REC may call the client to redo pages whose update was lost due to a crash or due to media failure (REC

never attempts to redo operations). Therefore, when the client writes a page, it must provide a description of how to redo the update as well as how to undo the update. This description may be the value of specified bytes on the page or it may be logical operations on the data on the page.

When REC directs the client to redo a page update, it provides the client with the description it receives when the page was updated. For any given page REC directs the client to perform the updates in the same order they were originally done, beginning with the first update that was lost. However, it may interleave updates to different pages in any order. When redoing a page update, the client should only use the description provided by REC and the contents of the page. The client should not read or modify any other pages. (A page update may be undone by updating a different page then the one originally modified.)

Recovery Service operations

Figure 21 on page 150 shows how operations could be used within a transaction in order to increment a counter, *counter1*. It is assumed that the counter resides on a page *counter1page*.

```
tran_Begin(TRAN_TID_NULL, &currentTranId);
rec_GetRootOp(currentTranId, &rootOpId);
lm_Lock(currentTranId,  /* locker */
        "counter1",     /* entity */
        "increment");   /* lock mode */
rec_BeginOp(rootOpId, &currentOpId);

lm_Lock(currentOpId,    /* locker */
        "counter1page", /* entity */
        "exclusive");   /* lock mode */

rec_PinPage("counter1page", /* page Id */
            readWrite,      /* pinMode */
            nonIdempotent,  /* redoMode */
            &pageP,         /* page pointer */ /* out */
            NULL,     /* data associated with page */ /* out */
            NULL);    /* init notification for page */ /* inout */

/* Begin Page Update */
 *rec_BeginPageUpdate(pageP);

/* Increment the counter */
 *((int *)pageP + counter1offset)++;

rec_UpdatePageSet(&pageP,               /* page pointer List */
   1,                                  /* npages */
   currentOpId,                          /* opId */
   "exclusive lock by currentOpId on
   counter1page",                        /* lockDescr */
   "decrement the page counter by one",   /* undo Descr */
   "increment the page counter by one");  /* redo Descr */

/* Note that the order of performing the actual update to the */
/* page and calling rec_UpdatePageSet is unimportant. */

rec_EndPageUpdate(pageP);
rec_UnPinPage(pageP);

rec_CommitOp(currentOpId,                              /* opId */
   "increment counter1 by one",                        /* opDescr */
   "increment lock by currentTranId on counter1 "); /* lockDescr */

tran_End(currentTranId);
```

Figure 21. Recovery Service Operations

Operation locking

An operation should perform updates to recoverable storage only after acquiring the necessary *update locks* and should hold on to these locks until the operation commits. This ensures that an operation's execution or its abort does not interfere with other operations. At the time an operation commits, it

can release its locks provided its master has acquired sufficient locks. The locks held are considered sufficient if the committed operation can be logically undone without the master having to acquire any new locks. One strategy to prevent deadlocks is to order all lockable entities and ensure that all lock requests made within an operation tree conform to that order. While the client of the recovery interface is not required to write deadlock free code, it should ensure that aborting an operation or a transaction never results in a permanent deadlock.

Note: The Lock Service does not provide an update lock mechanism suitable for operation duration locking. Users of operation logging must implement an alternative mutual-exclusion mechanism for operations.

Buffer manager

The buffer manager maintains a write-back cache of volume pages in main memory. It provides primitives to pin a page in main memory and to perform page updates within operations. The buffer manager uses an implementation-dependent policy for replacing pages in the buffer cache. However, it takes "hints" from the client that influence its replacement policy, and it provides the ability for clients to force pages to disk.

Buffers are a scarce resource. The client should pin buffers for a minimal amount of time. In addition, if the client has more than one page pinned at a time, it is up to the client to ensure that there are no deadlocks.

Media recovery

During crash recovery, REC assumes that data that was written to disk is permanent and only data that was in volatile memory and was not written to the disk must be recovered. If a failure occurs that destroys data on the disk, that data can be recovered using REC's media recovery facility.

REC provides three facilities for media recovery: a dump facility, which allows the client to obtain a fuzzy snapshot of the recoverable data and store that snapshot (in a UNIX file, for example); a restore facility, which allows the client to restore the state of recoverable data from a previous dump (this uses the log file as well as the files which store the data obtained in the dump); and a query facility, which allows the client to query the state of the backup to a particular volume or the state of a restore in progress. Media recovery functionality is described below.

Dump: Dumping data from a recoverable volume to a safe medium, such as a tape or disk. REC supports a fuzzy, continuous, recoverable dump:

- *Fuzzy*: The dump of a volume can be performed while the system is running and while the volume is online and available. The dump is divided into segments.
- *Continuous*: The client specifies the size of the segments. After each segment, the contents of that segment together with as many previous

segments as necessary are considered a backup. Therefore, the client is not required to dump the whole volume in every backup session.

– *Recoverable backups*: When a dump is interrupted due to a crash, REC keeps enough information about the dump to be able to continue the dump after the system is restored. In the worst case the contents of the segment in progress may have to be dumped again. In any case, continuing after a crash does not in any way effect the possibility of using the dump for a restore.

Restore: Restoring recoverable data from a previous dump.

– The client can restore the contents of a volume from a previous dump. This requires all the segments comprising the dump, as well as the log file.
– The client can specify a specific dump by providing the name of the segment that ended the dump, or the client may restore from the latest backup.
– In order to save time, the client can restore a number of volumes in the same command.

Query: Querying the state of the backup or restore.

– Querying the state of the backup.
  - The client can obtain information about all backups to a particular volume.
  - The client can obtain the names of all the segments needed in order to restore the volume using this backup, as well as the time and date each dump was begun.
– Querying the state of a restore.
  - The client can obtain information about the state of a restore in progress.
  - The client can obtain a list of all the volumes being restored, the number of backup segments copied, and the total number of log records replayed.

REC supports a *fuzzy* backup taken while the volume is online. In a fuzzy backup, the contents of the backup do not represent a consistent state of the volume. However, using the backup and the log file, REC can redo, for each page of the backup, any of the updates that were made since that page was backed up.

The backup interface includes two functions: **rec_BeginBackupSegment** and **rec_EndBackupSegment**. The first call initiates a dump for a specified volume. In that call the client associates a name with the dump and specifies a size. REC dumps as much of the volume as it can without exceeding the size the client specified. The sizes the client specifies in the **rec_BeginBackupSegment** call divide the backup into segments. In the call to

rec_BeginBackupSegment, the client provides an upcall; REC dumps the segment by making repeated calls to this upcall. In each call REC provides data the client must save. When the call to **rec_BeginBackupSegment** returns, the client must make sure all the data collected by the upcalls is safe. The client then calls **rec_EndBackupSegment** to notify REC that the data is safe. A backup is considered complete after the client makes sufficient calls to **rec_BeginBackupSegment** and **rec_EndBackupSegment** to store the entire volume.

After the client has a complete dump of the volume, the client can then continue to dump additional segments. Each segment completes an additional backup; a backup comprising the smallest number of the latest segments that span the entire volume.

For example, if REC requires ten segments to dump the entire volume, after the tenth call to **rec_EndBackupSegment** the client has a complete backup. The client can then dump additional segments, and at any time the last ten segments comprise a complete backup from which the client can restore the data of the volume.

Each backup command relates to a single volume. There can be at most one backup in progress for any volume. However, the client can dump several volumes concurrently. If a backup (or several backups) is interrupted due to a system crash, the backup can be continued—any segments which were in progress, that is, segments for which an **rec_EndBackupSegment** call was not issued, are aborted and will be dumped again. REC keeps track of all this information.

If the client tries to start a backup while a backup is in progress, the backup in progress will be aborted. (If the client crashes and the server stays up performing a backup, the client can start a new backup simply by calling **rec_BeginBackupSegment**; the previous backup will be aborted.)

If a segment completes a backup, the call to **rec_EndBackupSegment** returns a backup identifier that can be used to identify the backup when querying the backup or when the backup is used to restore a volume. The state of a backup, as well as the list of all segments comprising the backup, can be obtained via the **rec_QueryBackup** function. If the client issues a query providing a backup identifier obtained from the **rec_EndBackupSegment** command, the call provides information about the specified backup. If the caller supplies the special value `REC_NULL_BACKUP_ID`, the call provides information about the most recent complete backup to the specified volume.

The client can restore the contents of a volume by issuing the **rec_Restore** command. The client specifies a volume and a backup identifier. As with the

query function, providing the special value REC_NULL_BACKUP_ID instructs REC to restore the volume from the most recent backup to the volume.

The restore proceeds in two phases. First, REC restores the contents of the volume from the backup segments. Then, REC replays the log file and redoes all necessary updates. Since REC uses only one log file, calling **rec_Restore** for each volume that needs to be restored requires replaying the log file several times. In order to avoid this, **rec_Restore** accepts a list of volumes (and a list of backup identifiers). The client can then restore any number of volumes in a single request. However, the client is not permitted to have more than one outstanding restore request at a time.

REC header files and libraries

Header files

The file **rec/rec.h** contains the REC interface declarations for the C language. This file must be included in any file that uses the REC interface functions or data types.

Libraries

Applications using REC must link with the following libraries, in the order shown:

EncServer

Encina

See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Data types

This section describes the data types and auxiliary functions defined by REC.

General data types and auxiliary functions

REC defines the **rec_status_t** for values returned by REC functions.

The **rec_Free** function frees a pointer allocated by REC (for example, the **rec_backupInfo_t** structure).

Configuration parameters

REC defines the **rec_configParams_t** data type to contain configuration parameters for REC.

Operation data types and related functions

REC defines data types that contain the identifier of a client holding a lock on behalf of operations and transactions and a temporary identifier for an

operation. REC defines two related functions that can be used to determine if two operation identifiers are equal or to represent an operation identifier as a string.

- **rec_hid_t**
- **rec_opId_t**
- **rec_OpIdEqual**
- **rec_OpIdToString**

Buffer manager data types

REC defines data types that contain the unique identifier of a page, enumerate the types of client operations on a page while the page is pinned, and enumerate the properties of the redo information associated with a page update.

- **rec_pageId_t**
- **rec_pinMode_t**
- **rec_redoMode_t**

The **rec_GetPageSize** function can be used to return the remaining space available on the page (in bytes) for use by the client.

Description data types

REC defines data types that contain descriptions of uninterpreted information supplied to the recovery interface by the client, such as the description of an operation or the description of locks held by an operation.

- **rec_descr_t**
- **rec_descrList_t**

Media recovery data types and related functions

REC defines data types that uniquely identify a backup, enumerate the types of backups, contain information about a backup, and contain information returned by the **rec_QueryRestore** function.

- **rec_backupId_t**
- **rec_backupInfo_t**
- **rec_backupType_t**
- **rec_restoreInfo_t**

The **rec_BackupIdEqual** function can be used to determine if two backup identifiers are equal.

Information data types and related functions

REC defines a data type that contains information about the current configuration for REC and a data type that contains information about a logical volume for REC. REC also provides functions to retrieve this information.

- **rec_configInfo_t**
- **rec_volumeInfo_t**
- **rec_QueryConfig**
- **rec_QueryVolume**

Upcall data types

The **rec_Init** function takes several upcalls to handle conditions that occur during the initialization of REC. The types are listed below. For a detailed description of each of the functions, see the **rec_Init** reference page.

- **rec_upcallBuffersExhausted_t**
- **rec_upcallDropLocks_t**
- **rec_upcallPostInit_t**
- **rec_upcallRedoPageUpdate_t**
- **rec_upcallReobtainLocks_t**
- **rec_upcallUndoOp_t**
- **rec_upcallUndoPageUpdateSet_t**

Initialization and restart

REC initialization is a two step process. The first step initializes REC by calling the **rec_Init** function wherein the log file, client upcalls, and REC configuration parameters are specified. The **rec_Init** function must be called before any other function in the REC interface can be invoked.

After the **rec_Init** function is called, REC is in a special mode called admin mode. In this mode, the client may access its restart information and is allowed to perform administrative functions such as enabling and disabling volumes, or restoring the contents of a volume from a backup. However, the client should not attempt to access any of the recoverable volumes in admin mode.

In the second step, the client calls the **rec_RecoverVolumes** function causing REC to read the log file and to redo all the updates that did not make it to disk because of the crash. When the call to the **rec_RecoverVolumes** function returns, all the volumes that were enabled are available for use by the client, and REC is now in normal mode. The client should not perform page updates on volumes, whether or not they were mounted at the time of the last system crash, without first recovering them by calling the **rec_RecoverVolumes** function.

Enabling and disabling volumes

REC uses volumes created by the Toolkit Volume Service. After a volume is created, it may be in one of three states:

- *Enabled*: available for use by the client

- *Disabled*: unavailable for use by the client, but it may be enabled at any time
- *Broken*: unavailable for use and must be repaired before it can be enabled.

The client is responsible for creating volumes using the interface provided by VOL. In order to make the volume accessible through REC, the client must enable the volume by calling **rec_EnableVolume**. Clients may require that a volume be initialized once after the volume is enabled and before it is used. After the volume is enabled, the client reads and writes data on the volume, provided the client already called **rec_RecoverVolumes**.

An enabled volume is re-enabled and auto-recovered by REC during each subsequent system restart when the client invokes **rec_RecoverVolumes**.

If the client wants to prevent data from being written to a volume, the client may disable the volume by invoking the **rec_DisableVolume** function. Clients may wish to disable a volume to perform some administrative operation on it, such as relocating it on a different set of disks. A volume is also automatically disabled by REC when a media error is encountered on the volume, in order to guard against transient media failures.

A volume can only be disabled when REC is in an administrative mode. A server is brought up in an administrative mode by calling **rec_Init**, but not **rec_RecoverVolumes**. In administrative mode, REC makes a limited set of its functions available (**rec_DisableVolume**, **rec_Restore**, **rec_QueryBackup** and **rec_QueryRestore**). Also, REC does not mount any volumes and does not redo or undo any volume updates.

Note that disabling a volume can lead to a subsequent restart failing. This can happen if during that restart, REC needs to abort a transaction which requires writing to the disabled volume or if it needs to redo updates on that volume.

The only reason to disable a volume is to inform REC when a volume is being deleted. Relocating a volume on a different disk can be performed without disabling it, when the server is in the administrative mode.

A disabled volume can be re-enabled by either calling **rec_EnableVolume** or by invoking **rec_Restore**.

In order to reduce the restart time, REC performs a checkpoint periodically and stores its state in the log file. An application can call the **rec_SetCheckpointInterval** function to specify how frequently REC performs checkpoints. An application can call the **rec_ForceCheckpoint** function to force a checkpoint to be performed as soon as possible.

Initialization upcalls

The following upcalls are used to initialize REC:

- Redo-page-update upcall
- Undo-page-update-set upcall
- Undo-operation upcall
- Drop-locks upcall
- Reobtain-locks upcall
- Buffers-exhausted upcall

These upcalls are detailed on the reference page for the **rec_Init** function.

Reading and writing client restart data

REC provides two functions to read and atomically update client restart data: **rec_ReadRestartData** and **rec_WriteRestartData**.

Advanced functions for initialization

Initializing the recovery service with the **rec_Init** function is sufficient in most cases. For those applications that require greater flexibility than the **rec_Init** function supplies, REC also provides two alternative functions for initializing the recovery service. The **rec_PreInit** and **rec_EnableLogFile** functions split the functionality of the **rec_Init** function, making it possible to perform other initialization or application-specific work in between the two. An application that does not require this additional flexibility should use the **rec_Init** function only. Conversely, if an application uses the **rec_Init** function, neither the **rec_PreInit** nor **rec_EnableLogFile** functions should be used.

The **rec_PreInit** function supplies the upcalls and configuration parameters for REC. This function includes an additional upcall not available in the standard **rec_Init** function. The post-initialization upcall allows the client to perform initialization that requires the use of the log file, such as reading or writing restart data, as well as any other initialization that must be done before recovery is begun. This upcall is invoked by the **rec_EnableLogFile** function.

The **rec_EnableLogFile** function specifies the log file for the recovery service to use. The log file must be created before it can be enabled. After the log file is enabled, the client can call the **rec_RecoverVolumes** function to recover volumes. Refer to “Chapter 8. Log Service (LOG)” on page 119 of this manual for additional information on log files.

Operation management

This section describes the interface to the operation manager component of the Recovery Service (REC). This component provides facilities to establish and query the association between transactions and operations and to begin, commit, and abort operations. It also provides the ability to declare a transaction to be *lazy*; that is, its commitment is *not* guaranteed to be *permanent*.

Association between transactions and operations

As mentioned in "Transaction Service overview" on page 11, each transaction is associated with an operation tree and each operation is associated with at most one transaction. Two REC functions provide a means to determine this association: **rec_GetRootOp** and **rec_GetAssociatedTran**.

Beginning and ending operations

REC provides functions to begin, commit, and abort operations: **rec_BeginOp**, **rec_CommitOp**, and **rec_AbortOp**.

Declaring lazy transactions

A transaction's commitment is not guaranteed to be permanent when you declare it to be lazy using the **rec_LazyTran** function.

Forcing the log

REC does not put any constraints on the undo description of an operation. In particular, the client can decide to provide a `NULL` undo description for an operation. This has the effect that the operation is never undone. Therefore, when the commit record of an operation with a `NULL` undo description is written to the log, the effects of the operation are guaranteed to be permanent even if the transaction it is within aborts. The **rec_ForceLog** function guarantees that all prior log records have been written to the log; thus a **rec_ForceLog** call after a **commitOp** call with an `NULL` undo description guarantees the permanence of the committed operation.

Buffer management

This section describes the interface to the buffer manager component of the Recovery Service (REC). The buffer manager provides functions to pin and unpin pages, to update pinned pages as part of an operation, and to force pages to disk. It accepts hints from the client that influence its buffer-replacement policy. The buffer manager also provides primitives to delete pages belonging to a volume from the buffer cache.

Pinning and unpinning pages

REC provides two functions that can be used to pin and unpin pages: **rec_PinPage** and **rec_UnPinPage**. The **rec_AckInitNotification** function can be used to acknowledge the init notification for a pinned page.

Updating pages

Updating a page is a three-step process. The client must call **rec_BeginPageUpdate** before modifying a page. The modifications are "installed" by calling **rec_UpdatePageSet**. The end of the modification is indicated by calling **rec_EndPageUpdate**.

Flushing pages to a volume

Before a clean shutdown of a recoverable server can be performed, data may need to be flushed from the page buffer to the server's volume. REC provides the **rec_FlushVol** function for this purpose.

Forcing pages to disk

Some clients (typically nontransactional) may need to force contents of pages to disk periodically. REC provides a *force* primitive, **rec_ForcePage**, for this purpose.

The contents of a log media archive can be flushed to disk by using the **rec_FlushMediaArchive** function.

Providing hints to the buffer manager

The client can provide *hints* that influence the buffer management's policy for replacing buffers in the buffer pool. If the client knows that a buffer is not going to be accessed for a long time, it can tell the buffer manager to make it the most likely candidate for replacement by using the *preflush* primitive, **rec_PreFlushPage**.

Similarly, the client may know which pages it will access in the near future. The client can indicate this to the buffer manager by using a *prefetch* primitive, **rec_PreFetchPage**.

Initializing a range of pages on a volume

The **rec_InitVolume** function provides the client with the ability to initialize a range of pages belonging to a volume.

Media recovery

This section describes the interface to the media recovery facility of the Recovery Service (REC). REC provides the following calls:

- **rec_BeginBackupSegment**
- **rec_EndBackupSegment**
- **rec_TruncateBackup**
- **rec_RetainLastBackups**
- **rec_Restore**
- **rec_EnableMediaArchiving**
- **rec_DisableMediaArchiving**

- **rec_QueryBackup**
- **rec_QueryRestore**

REC provides three facilities for media recovery: a dump facility, which allows the client to obtain a fuzzy snapshot of the recoverable data and store that snapshot (in a UNIX file, for example); a restore facility, which allows the client to restore the state of recoverable data from a previously taken dump (this uses the log file as well as the data stored during the dump); and a query facility, which allows the client to query the state of the backup to a particular volume or the state of a restore in progress.

A backup is performed by calling **rec_BeginBackupSegment**. The client may also specify the oldest backup needed by calling **rec_TruncateBackup**; this allows REC to free storage used by older backups. These functions are only available in REC's normal mode. The contents of a volume may be restored from a backup using the function **rec_Restore**. This function is only available when REC is in admin mode (see the explanation of REC's modes in "Initialization and restart" on page 156).

Log media archiving can be turned on or off while the system is running. The function **rec_EnableMediaArchiving** enables archiving for the server and **rec_DisableMediaArchiving** disables it. Backups can be performed only when media archiving is enabled. "Chapter 8. Log Service (LOG)" on page 119 of this manual provides additional information on archiving.

The query facility provides two functions:

- The **rec_QueryBackup** function provides the caller with information about the state a backup to a particular volume
- The **rec_QueryRestore** function provides information about the state of a restore.

These functions are available in both admin mode and normal mode.

Diagnostics

This section documents the diagnostic support provided by the Recovery Service. This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal errors) during use of the Recovery Service, informative messages that trace calls on the Recovery Service (tracing), and snapshots of the state of the Recovery Service (state dumps).

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files, or a ring buffer, by use of the relevant Encina Toolkit Trace

Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer**

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

The Recovery Service generates a *fatal run-time error* whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility (see "Trace" on page 165) and the application's execution terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility (see "Directing output" on page 161). This section defines the fatal error messages that may be output by the Recovery Service and briefly describes why they occur and how they can be avoided or remedial action taken.

The Recovery Service provides the following fatal error messages:

- "Attempted to drop locks when no upcall was registered."—REC needs to drop locks but the client has supplied a NULL drop locks function pointer via **rec_Init**. Fix the client program to pass in a valid drop locks function pointer in **rec_Init**.
- "Attempted to redo a page when no upcall was registered."—REC needs to redo a page update but the client has supplied a NULL redo page update function pointer via **rec_Init**. Fix the client program to pass in a valid redo page update function pointer in **rec_Init**.
- "Attempted to reobtain locks when no upcall was registered."—REC needs to reobtain locks but the client has supplied a NULL drop locks function pointer via **rec_Init**. Fix the client program to pass in a valid reobtain locks function pointer in **rec_Init**.
- "Attempted to undo a page when no upcall was registered"—REC needs to undo a page update but the client has supplied a NULL undo page update function pointer via **rec_Init**. Fix the client program to pass in a valid undo page update function pointer in **rec_Init**.
- "Attempted to undo an operation when no upcall was registered."—REC needs to undo a operation but the client has supplied a NULL undo operation function pointer via **rec_Init**. Fix the client program to pass in a valid undo operation function pointer in **rec_Init**.
- "LOG ERROR: *call* failed with *returncode*."—REC's *call* to LOG failed, and returned the string equivalent of the LOG *returncode*. Consult the reference page for the **log_status_t** data type for the meaning of the LOG returncode string, and restart the program after taking remedial action on the log server.

- ″rec_GetPageSize was provided an invalid redo mode.″—Client passed in an invalid value for the ″redoMode″ in the **rec_GetPageSize** function. Fix the client program.
- ″rec_PinPage was provided an invalid pin mode.″—Client passed in an invalid value for the ″pinMode″ in the **rec_PinPage** function. Fix the client program.
- ″REC failed to mount volume *volumename* during restart because: *returncode*.″—REC failed to mount an enabled volume during restart. Fix the broken volume and restart.
- ″REC failed to certify volume *volumename* during restart because: *returncode*.″—REC failed to certify an enabled volume during restart. Fix the broken volume and restart.
- ″REC version is not compatible with data in log file.″—Client attempted to execute an application with an older version of REC after executing with a newer version of REC on the same log file. Relink client with latest version of REC and re-execute.
- ″Undo operation ended with active slaves.″—One of the client-supplied undo routines (the undo-page-update-set or undo-op upcall registered via **rec_Init**) begins operations which are slaves of the undo operations but does not end those operations before returning. Fix the client program.
- ″Undo operation has repeatedly failed.″—The undo of an operation/page update has been aborted repeatedly by the client. Fix the client program.

The following fatal messages indicate an internal error in REC. Contact your Encina support representative with a stack trace/core dump of the program, a copy of the client's data and log volumes, if available, and as much other information about the client program as possible.

- ″Invalid recMedia log record type.″
- ″Invalid buffer state in **bufEvent_PinPage**.″
- ″Invalid buffer state in **bufEvent_UnPinPage**.″
- ″Invalid buffer state in **bufEvent_BeginPageUpdate**.″
- ″Invalid buffer state in **bufEvent_ForcePage**.″
- ″Invalid buffer state in **bufEvent_ForcePageAsync**.″
- ″Invalid buffer state in **bufEvent_CompleteWrite**.″
- ″Invalid buffer state in **bufEvent_IOError**.″
- ″Invalid buffer state in **bufEvent_RedoPageUpdate**.″
- ″Cannot create thread in **bufIO_Init**.″
- ″wrong type of log record in **opEvent_ReplayLogRecord**.″
- ″Wrong type of log record encountered during Op Undo.″
- ″Invalid log record type encountered during Op Undo.″
- ″Operation Identifier Exhaustion.″

- ″Invalid type of log record in routine **VerifyLogRecord**.″
- ″Wrong type of record in procedure **ReplayDPTableChange**.″
- ″Redo Pass: wrong type of log record.″
- ″Error removing an init range during restore.″
- ″Error removing a resume init range during restore.″
- ″Redo Failure redoing pageInit during restore.″
- ″Redo Failure redoing pageUpdate during restore.″
- ″wrong type of log record in restore ReplayLogRecord.″

Warning messages

The Recovery Service generates a *warning* message if it detects a nonfatal error or an exceptional condition. The Encina Trace Facility presents the warning as an output message (see “Trace” on page 165). The Encina Trace Facility has functions that can define the destination of the error output display (see “Directing output” on page 161). This section defines the warning messages that can be displayed by the Recovery Service. The section also briefly describes why the error messages occur, how they can be avoided, or what remedial action can be taken.

The Recovery Service provides the following warning messages:

- ″All buffers in buffer pool pinned.″—There are no free buffers to satisfy the current user request to the buffer manager; the users has all buffers in the buffer pool pinned. Make sure that the buffer pool size is at least equal to the number of threads in the program multiplied by the number of concurrently pinned pages in each thread. Also, make sure that all pinned pages are eventually unpinned.
- ″Allocating last row of operation identifier table.″—REC is about to run out of operation identifiers (a fatal error). The client has too many unresolved operations/transactions. Fix the client program.
- ″Incomplete page write detected on page *pageId*.″—On restart, REC was unable to fix an ″incomplete write″ on the page with the specified id. The volume containing the page is disabled. Restore the volume containing the broken page.
- ″Invalid volume parameters to init page *pageId*.″—An invalid volume id or invalid page number was specified when initializing a page via the **rec_PinPage** function. Fix the client program.
- ″Read Page on vol *volumename* pageNum: *pagenumber* failed with volStatus: *volstatus*.″—An I/O request to a volume failed. Ensure that the volume is enabled; it may be necessary to restore the volume from backups.
- ″ReadPageVersions failed attempting to fix page *pageId*.″—On restart, REC was unable to fix an ″incomplete write″ on the page with the specified id. The volume containing the page is disabled. Restore the volume containing the broken page.

- "Read Segment on vol *volumename* pageNumBegin: *pagenumber* numPages: *number* failed with volStatus: *volstatus*."—An I/O request to a volume failed. Ensure that the volume is enabled; it may be necessary to restore the volume from backups.
- "Re-reading page on vol *volumename* pageNum: *pagenumber* failed with volStatus: *volstatus*."—An I/O request to a volume failed. Ensure that the volume is enabled; it may be necessary to restore the volume from backups.
- "restoreCleanup: Error dismounting volume *volumeid*."—An error was encountered when dismounting a volume at the end of a restore. Fix the broken volume and redo the restore
- "Write Page on vol *volumename* pageNum: *pagenumber* failed with volStatus: *volstatus*."—An I/O request to a volume failed. Ensure that the volume is enabled; it may be necessary to restore the volume from backups.
- "Write Segment on vol *volumename* pageNumBegin: *pagenumber* numPages: *number* failed with volStatus: *volstatus*."—An I/O request to a volume failed. Ensure that the volume is enabled; it may be necessary to restore the volume from backups.
- "VerifyFileHeader: File: # pages from #, expected # from #."—Invalid data was provided to a "read upcall" during restore. Verify whether you provided the correct data to the upcall during restore.
- "VerifyFileHeader: file description found: *actualvalue* expected: *expectedvalue*."—Invalid data was provided to a "read upcall" during restore. Verify whether you provided the correct data to the upcall during restore.
- "VerifyFileHeader: File for volume *actualvalue*, expected *expectedvalue*."—Invalid data was provided to a "read upcall" during restore. Verify whether you provided the correct data to a call to the upcall during restore.
- "VerifyFileHeader: file start lsn is *actualvalue*, expected *expectedvalue*."—Invalid data was provided to a "read upcall" during restore. Verify whether you provided the correct data to the upcall during restore.
- "Volume *volumeid* is not enabled and is not mounted."—The program was restarted with the specified volume disabled. Presence of any unpropagated updates to the volume will result in a fatal error. Restart the program with the specified volume enabled.

Audit messages

The Recovery Service does not emit any audit messages.

Trace

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes calls on the Recovery Service. All Recovery Service interface functions are capable of tracing their entry, parameters, and exit. Clients of the Recovery Service can

enable a specific class of event tracing, such as entry/exit, or a collection of events (entry/exit and parameter tracing).

The level of tracing is controlled by the value of a single variable that is exported by the Recovery Service:

```
unsigned long rec_traceMask
```

The Encina Trace Facility defines a number of ″bit constants″ that when set in the above variable cause a specific form of tracing to occur. For example, the TRACE_ENTRY constant enables entry/exit tracing of Recovery Service interface functions. The Recovery Service also defines Recovery Service-specific bit constants for important classes of internal event.

The following section describes the of tracing supported by the Recovery Service.

Global trace levels

The Recovery Service supports the global trace levels defined by the Encina Trace Facility. These levels are as follows:

- TRACE_ENTRY — traces the entry and exit of functions that make up the Recovery Service interface.
- TRACE_PARAM — traces the parameters passed to the interface functions.
- TRACE_EVENT — traces all events in the Recovery Service. This trace class outputs large amounts of information, and for this reason, is not always recommended.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- TRACE_GLOBAL
- TRACE_NONE

Recovery Service trace levels

The Recovery Service provides no specific trace levels.

State dump

The Recovery Service provides the **rec_DumpState** function to dump the state of the Recovery Service at an application.

Chapter 11. Restart Service

Restart Service overview

The Encina Restart Service provides a general mechanism for storing and retrieving the restart data used by an Encina server. The Restart Service allows Encina servers to store and retrieve the restart data used by the Encina Volume Service. This restart data provides information about the logical and physical volumes used by a server, the location of those volumes, and the physical devices associated with them.

The restart device is a file in the local file system into which the restart data is written. The Restart Service uses multiple copies of the restart device to store the restart information, providing protection against failures by *mirroring* this information. The restart device and mirrors are specified as an argument to the **restart_Initialize** function.

Note: In the restart string used with the **restart_Initialize** function, the names of the restart file and its mirror must be separated by a : (colon) on UNIX platforms but by a ; (semicolon) on Windows NT.

When writing restart information, the **restart_WriteRestartData** function writes sequentially to all restart devices. Maintaining multiple copies of restart data and writing these files sequentially helps guarantee that at least one of these files is valid. When restarting a server, once a valid copy of the restart data is found, the other restart device(s) are updated if their contents are older or invalid. Failing to find a valid copy of the restart data in the restart device or any mirror, or failing to synchronize any devices, is treated as a fatal error.

The Encina restart functions do not maintain any information about the server with which a specific restart device is associated. The user must supply the name of the correct restart devices associated with a specified server.

The **restart_ReadRestartData** function is used to return the restart data written by the last successful invocation of the **restart_WriteRestartData** function. The Encina Restart Service also exports the **restart_mode_t** data type, which is used during initialization to indicate whether a restart is a cold or warm start.

Chapter 12. The Transaction Manager-XA (TM-XA) Service

TM-XA overview

This section describes the interface to the Encina Transaction Manager-XA (TM-XA) Service. The TM-XA Service implements the transaction manager side of the X/Open XA interface. The TM-XA Service chapter is organized as follows:

- The remainder of this section describes the basic functionality of the XA interface and how it is coordinated with applications using TRAN.
- "Initialization and restart" on page 156 describes how to use the application programmer's interface to initialize and close down the TM-XA Service.
- "Transaction context" on page 194 describes how a programmer can control the use of the TM-XA Service.
- "Use of XA by the TM-XA Service" on page 195 describes how features of XA are or are not used by the TM-XA Service.
- "Trace and state dump information" on page 133 describes the diagnostic support provided by the TM-XA Service.

The TM-XA Service is used by transactional applications (such as a transaction processing monitor) to coordinate distributed transactions in X/Open XA-compliant resource managers. A transactional application is depicted in Figure 22 on page 170.

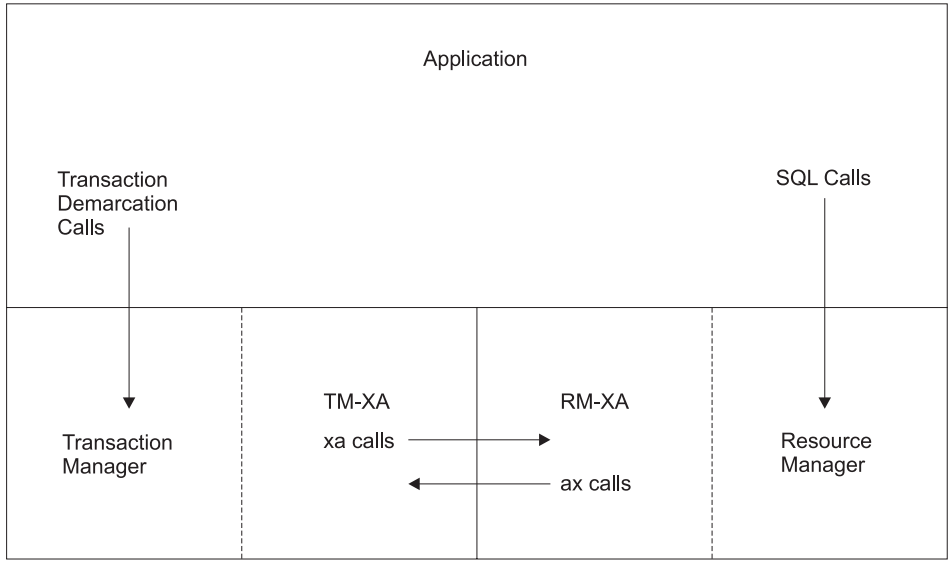

Figure 22. An application using the TM-XA Service

The TM-XA Service is built on two components of the Encina Toolkit Executive: the Distributed Transaction Service (TRAN) and the Thread-to-Tid Mapping Service (ThreadTid). Applications using the TM-XA Service provide the list of resource managers that use the XA interface and set the transaction context before calling these resource managers. The service handles commit coordination and recovery of the resource managers automatically from the application's point of view. The TM-XA Service uses TRAN to implement these features.

It is assumed that the reader is familiar with the X/Open XA interface as specified in recent drafts from X/Open. This document supports the version of XA described in the Draft XA Specification of September 1991 (company review draft).

The information documented in this chapter serves several purposes. First, it describes the interface that an application programmer can use for controlling TM-XA Service. Second, it describes the transaction manager (TM) functions that are provided to a resource manager (RM). This information is needed by people who implement resource managers that work with the TM-XA Service. Third, it describes how the TM-XA Service uses the XA standard to coordinate global transactions.

The relationship between XA and TRAN

The XA protocol is used by transaction managers to communicate transaction context to resource managers. Each time the transaction manager executes some kind of transaction function, such as beginning a transaction, the TM informs the RM of the function.

The primary strategy used by the TM-XA Service is to map TRAN events to appropriate XA calls. In particular, whenever a transaction is associated with a thread, the resource managers accessible from that thread are also informed of the association. When the association with a thread is relinquished, the resource managers are also told to relinquish the association between native resource manager requests and a transaction. Similarly, transaction prepares, aborts, and commits are communicated to RMs through the corresponding parts of the XA protocol.

Dissociation between a transaction and resource managers can be subtle. Two kinds of dissociation can occur. *Suspension* implies a temporary dissociation, where most state should be retained for later use. *Successful completion* implies that a collection of RM work on behalf of this transaction has been completed and only a modest amount of state must be retained about the transaction. Typically, a resource manager is assumed to preserve both cursor and lock information in a suspended association, making them available when the association is resumed. When an association is successfully completed, only lock information is assumed to be maintained for any other associations that may join the transaction.

The TM-XA Service suspends an association when it is reasonable for a transaction to continue work in the same context where it was suspended. For example, an association between a transaction and resource manager would be suspended when a thread starts working on a nested transaction. When the nested transaction completed, the parent transaction would resume with the state it had before the nested transaction began.

An example of a successful completion is provided by an application server. When a server starts work on behalf of a transaction, the server makes an association between the resource manager and the transaction. When the server finishes a request, it would normally dissociate the transaction as a successful completion. Although the server might later perform further work on behalf of the transaction, there is no assumption that cursor state from the first association remains. Hence, the successful completion dissociation was made.

Under ordinary circumstances, Transactional-C and the Monitor maintain the association between threads and transactions. The TM-XA Service coordinates with these services to maintain the communication between TRAN events and resource managers. However, the XA protocol may impose an excessive

amount of processing and network traffic to coordinate TRAN and resource managers when no use of the resource manager by a transaction is intended. Therefore, the TM-XA Service also provides a way for an application to selectively disable the TM-XA Service for a collection of transactions.

General usage of TM-XA Service

Most Transactional-C applications have to perform only a modest amount of work to exploit XA services:

- Each resource manager instance that is to be used by the application is registered with the TM-XA Service using the **tmxa_RegisterRMI** function. (See "Initialization and restart" on page 156.)
- The TM-XA Service is initialized by calling the **tmxa_Init** function. (See "Initialization and restart" on page 156.)

The processing performed by the **tmxa_Init** function connects the TM-XA Service with other facilities in the Distributed Transaction Service (TRAN) and the Thread-to-Tid Mapping Service (ThreadTid). This eliminates the need for application programs written in Transactional-C or the Encina Monitor to perform any other work to support XA interoperability. However, if selective control of TM-XA Services is desired, then appropriate use of the **tmxa_OpenRMIs** and **tmxa_CloseRMIs** functions (see Section "Serializing access to RMs" on page 190), and the **tmxa_SetAutomaticXaAssociation** function (see "Transaction context" on page 194) may be required. Programs that use TRAN directly to start and end applications can also exercise fine control over the TM-XA Services through the use of the **tmxa_AssociateTid** and **tmxa_DissociateTid** functions (see "Transaction context" on page 194).

In order for the TM-XA Service to operate correctly, applications using the TM-XA Service require that a recovery service be registered with TRAN. A recovery service must be provided regardless of whether the application directly manages recoverable data or not. See "Providing a recovery service" on page 194 for further information.

CAUTION:
The TM-XA Service cannot tolerate thread cancellation inside of its routines. TM-XA calls should be made only from threads with thread cancellation disabled.

The flow of XA calls in typical APIs

The XA communication between resource manager instances and the transaction manager has two stages. In the first stage, threads and resource manager instances are associated with transactions while the resource managers perform work on behalf of the application. In the second stage, the transaction manager and the resource manager communicate in order to coordinate their decisions about the outcomes of transactions. The first stage is implemented by the routines that associate and dissociate transactions with

resource manager instances: **tmxa_AssociateTid** and **tmxa_DissociateTid**. The second stage is implemented by a collection of routines that are invoked automatically when TRAN determines the outcome of a transaction.

XA and Transactional-C

Transactional-C is integrated with XA primarily through ThreadTid. Whenever Transactional-C calls ThreadTid to associate a transaction with a thread, a callback is made to the TM-XA Service that makes a corresponding association with a resource manager instance via the **xa_start** procedure. When a transaction identifier is seen by the TM-XA Service for the first time, the TM-XA Service registers callbacks for coordinating the commitment protocol stage; therefore Transactional-C programs need not perform any additional functions. Figure 23 shows a block diagram of communication in a Transactional-C application using various Encina components and TM-XA.

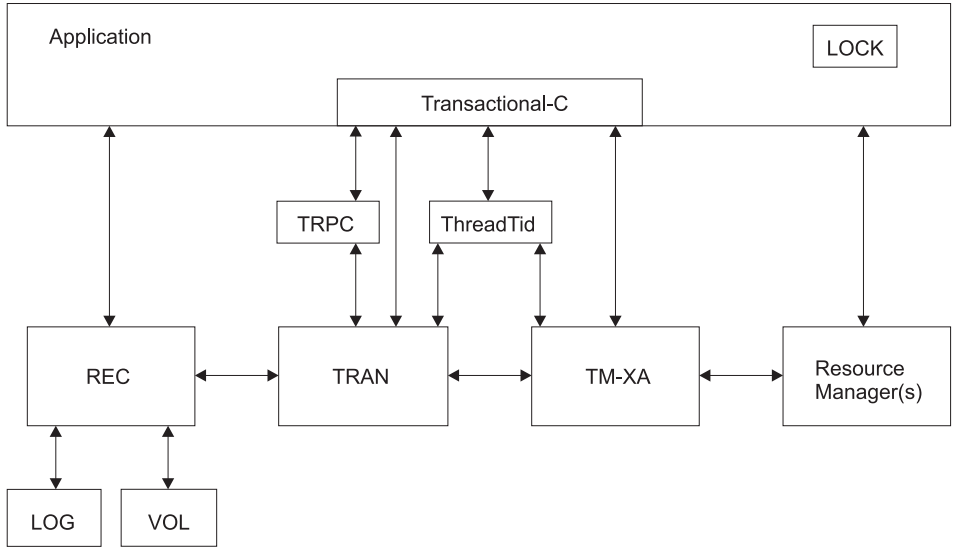

Figure 23. Tran-C application communications with Encina and TM-XA

To illustrate the underlying process, each set of XA calls associated with each Transactional-C construct is considered in turn. Each example displays a representation of the Transactional-C construct on the left. To the right is shown the chain of calls that is made leading to an appropriate XA call. Note that the maximal number of calls are shown. Some calls are not made in certain circumstances. For example, calls of **xa_forget** are made only when heuristic decisions are made by the resource managers.

For purposes of illustration, the examples assume that each nested transaction uses a different XID. See "Nested transactions" on page 187 for a discussion of nested transactions in XA. These examples also assume each transaction

construct (even **subTran**) creates a top-level transaction. When the transaction created is a nested transaction, the prepare and commit flows do not occur until the top-level transaction completes. This is because the XA prepare and commit calls happen when the TRAN callbacks are invoked.

The transaction construct

The **transaction** construct is used to start a (possibly nested) transaction. When a new transaction is started, any association that the thread had with a previous transaction is removed and a new association is made with the new transaction. In Figure 24, the previous (outer) transaction is denoted `To` and the new (inner) transaction is denoted `Ti`.

```
transaction
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
                   |-> AssociateTid -> xa_start(Ti,noflags)
               ...transaction code...
          -> threadTid interaction -> DissociateTid ->
             xa_end(Ti,success)
          -> prepare callback -> xa_prepare(Ti)
onCommit
          -> resolution callback -> xa_commit(Ti)
               ...onCommit code...
onAbort
          -> abort callback -> xa_end(Ti,success); xa_rollback(Ti)
               ...onAbort code...
          -> finish callback -> xa_forget(Ti)
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 24. The transaction construct

When the nested transaction starts, the call by Transactional-C to the ThreadTid Service invokes the callback to the TM-XA Service which in turn first calls **tmxa_DissociateTid** to suspend the previous association (if one exists), and then calls **tmxa_AssociateTid** to create the new association. The **tmxa_AssociateTid** call will create the two phase commit (2PC) callbacks for this transaction. If the transaction is a top-level transaction, when the transaction ends, TRAN invokes a TM-XA Service callback to attempt to prepare the RMs through **xa_prepare** calls. If the transaction commits, the after-resolution callback will be generated by TRAN to the TM-XA Service to commit the RMs (through **xa_commit** calls). Should the transaction abort, the before-abort callback will be made by TRAN. This will possibly call **xa_end** (as required by the XA specification for transactions that have not yet been ended successfully) and then inform RMs that the transaction has aborted by calling **xa_rollback**. During 2PC, an RM might have returned some heuristic decision, which the RM will remember until told to forget it. When TRAN informs the TM-XA Service that it is done with a transaction (through the after-finished callback), the TM-XA Service informs RMs to forget any

heuristic decisions. Finally, any association with resource manager instances that existed before the **transaction** construct is resumed.

The topLevel construct

The XA processing for the **topLevel** construct is essentially the same as for the **transaction** construct described in "The transaction construct" on page 174.

The suspend clause

The use of a **suspend** clause in a **transaction** construct (Figure 25) causes the same XA behavior as for the **transaction** construct except that no prepare callback is made. That is, the association with the inner transaction is ended successfully (by passing the flag `TMSUCCESS` to the **xa_end** call) rather than suspended. The value of the flag passed to the **xa_end** call is provided to the TM-XA Service by the callback from the ThreadTid Service.

```
transaction
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
                   |-> AssociateTid -> xa_start(Ti,noflags)
               ...transaction code...
          -> threadTid interaction -> DissociateTid ->
             xa_end(Ti,success)
suspend
               ...suspend code...
onAbort
          -> abort callback -> xa_end(Ti,success); xa_rollback(Ti)
               ...onAbort code...
          -> finish callback -> xa_forget(Ti)
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 25. The suspend clause

As in the **transaction** construct, an aborted transaction may first have to call the **xa_end** function to meet the requirements of the XA specification. When the **transaction** construct has completed, any previously established association is resumed. "Maintaining transaction context" on page 180 provides a description of how to suspend the association between a transaction and resource managers in order to retain the cursor context over repeated associations.

The resume construct

The **resumeTran** construct is similar to the transaction construct, except that a transaction already exists. In order to execute this construct, the transaction must have been suspended by Tran-C, so the TM-XA Service resumes the association with the inner transaction rather than starting it.

The TM-XA Service determines how to start an association from its internal state. Hence, unlike dissociation, the TM-XA Service does not rely on

parameters passed during the ThreadTid Service callback to determine whether to resume or start an association.

The **suspend** clause in a **resumeTran** construct (Figure 26) is handled the same way as in a **transaction** construct.

```
resumeTran
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
                   |-> AssociateTid -> xa_start(Ti,join)
               ...resumeTran code...
          -> threadTid interaction -> DissociateTid ->
             xa_end(Ti,success)
suspend
               ...suspend code...
onAbort
          -> abort callback -> xa_end(Ti,success); xa_rollback(Ti)
               ...onAbort code...
          -> finish callback -> xa_forget(Ti)
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 26. The resume construct

The onCommit clause in a resume construct

The presence of an **onCommit** and **onAbort** clause (Figure 27) in a **resumeTran** construct is handled the same way as in a **transaction** construct.

```
resumeTran
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
                   |-> AssociateTid -> xa_start(Ti,join)
               ...resumeTran code...
          -> threadTid interaction -> DissociateTid ->
             xa_end(Ti,success)
          -> prepare callback -> xa_prepare(Ti)
onCommit
          -> resolution callback -> xa_commit(Ti)
               ...onCommit code...
onAbort
          -> abort callback -> xa_end(success); xa_rollback(Ti)
               ...onAbort code...
          -> finish callback -> xa_forget(Ti)
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 27. The onCommit clause in a resume construct

The concurrent construct

The **concurrent** construct (Figure 28 on page 177) is used to create multiple threads and/or multiple transactions. Before any subclauses are executed, the

association between any outer transaction and resource manager instances is suspended. Each subclause performs its own XA work as required. When the construction is completed, the associations with the outer transaction are resumed.

Note that the suspension is requested with migration. This is done to accommodate new subthreads, which should inherit the outer transaction's state.

```
concurrent
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
     subtran or subthread 1
      ...
     subtran or subthread n
}
onCommit
               ...onCommit code...
onAbort
               ...onAbort code...
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
concurrent
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
     subtran or subthread 1
      ...
     subtran or subthread n
}
coEnd
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 28. The concurrent construct

There is no substantive difference between using the **coEnd** clause or not. The **onCommit** and **onAbort** clauses do not cause any XA work since they are not the result of the outermost transaction committing or aborting. The inner transactions (subtransactions) have their callbacks attached to their **subTran** clause.

The subtTran clause

A **subTran** clause (Figure 29 on page 178) is handled the same way as the **transaction** construct.

```
subTran()
<subthread/subtransaction starts>
          -> threadTid interaction -> AssociateTid ->
             xa_start(Ti,noflags)
<subthread/subtransaction ends>
          -> threadTid interaction -> DissociateTid ->
             xa_end(Ti,success)
          -> prepare callback -> xa_prepare(Ti)
onCommit
          -> resolution callback -> xa_commit(Ti)
               ...onCommit code...
onAbort
          -> abort callback -> xa_end(success); xa_rollback(Ti)
               ...onAbort code...
          -> finish callback -> xa_forget(Ti)
```

Figure 29. A subTran clause

If nested transactions share XIDs with their ancestors, their initiation in a **subTran** clause will cause the **xa_start** call to resume an association and the **xa_end** call to suspend (with migration) the association.

The subtThread clause

A **subThread** clause (Figure 30) acts as if it is part of the containing transaction, and therefore resumes any association that might have been pending when the **concurrent** construct was encountered.

```
subThread()
<subthread starts>
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
<subthread ends>
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
```

Figure 30. subThread clause

The cofor construct

The **cofor** construct (Figure 31 on page 179) is handled like the **concurrent** construct.

```
cofor(int)
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
subTran
}
onCommit
               ...onCommit code...
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
cofor(int)
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
     subthread
}
onCommit
               ...onCommit code...
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
cofor(int)
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
     subTran
}
coEnd
          -> threadTid interaction -> AssociateTid ->
              xa_start(To,resume)
cofor(int)
          -> threadTid interaction -> DissociateTid ->
             xa_end(To,suspend|migrate)
{
     subthread
}
coEnd
          -> threadTid interaction -> AssociateTid ->
             xa_start(To,resume)
```

Figure 31. cofor construct

The concThread construct

The **concThread** construct (Figure 32) does not inherit any transaction context, and therefore no XA coordination is needed.

```
concThread()
          -> no transactions, so no xa calls
```

Figure 32. The concThread construct

Maintaining transaction context

For some applications, retaining transaction context in the resource manager over subsequent associations of the same XID is desired. An example of such a case is an Encina Monitor client that accesses a DBMS through a server and wishes to retain SQL cursor state over multiple TRPCs to the server. Resource manager transaction context can be retained by using the `TMSUSPEND` flag in the **xa_end** call to the RM.

The example in "The suspend clause" on page 175 shows that the initial setting in TM-XA causes the `TMSUCCESS` flag to be used when associations are ended using the **xa_end** call. TM-XA offers an interface to change the flag passed to the **xa_end** function. The **tmxa_SetAutomaticXaAssociation** function can be used to specify the end flag. The following pseudo-code examples show how the cursor state can be retained using the **tmxa_SetAutomaticXaAssociation** function.

In the **client_procedure** routine (Figure 33), the Monitor client makes three TRPCs to the server within a transaction, after which it simply commits the transaction.

```
client_procedure() {
   transaction {
       InitStateInServer();
       for ( some_iterations ) {
          DoSomeWorkInServer();
       }
       CleanupStateInServer();
   } onAbort {
     ...
   }
}
```

Figure 33. Client example code

The **server_Init** routine (Figure Figure 34 on page 181) called by the Encina Monitor sets the flag to be used when TM-XA calls **xa_end** to dissociate any new transactions.

```
void server_Init(argc, argv)
   int argc;
   char *argv[];
{
  ...
  tmxaStatus = tmxa_SetAutomaticXaAssociation(TRAN_TID_NULL,
                     TMXA_NEW_TOP_LEVEL_TIDS, TMSUSPEND|TMMIGRATE);
  ...
}
```

Figure 34. Initializing the server to retain cursor state

In the **server_Init** routine, the TMSUSPEND flag indicates that associations should be suspended so that resource manager transaction context is retained. The TMMIGRATE flag indicates that TM-XA intends, but is not required, to resume the association in a different thread from the one that suspended the association.

The **InitStateInServer** routine (Figure 35) sets up the DBMS cursor state for access in subsequent calls to the server, along with other initialization. (A cursor in SQL is a handle or name for an area in memory in which a parsed SQL statement and other information for processing the statement are kept. This helps in avoiding reparsing of SQL statements that could be issued repeatedly, thereby speeding up database access.)

```
InitStateInServer()
{
  threadTid interaction -> AssociateTid -> xa_start(XID1, TMNOFLAGS)
   ...
  /* define the cursor and open it */
  EXEC SQL DECLARE my_statement STATEMENT;
  EXEC SQL AT my_db DECLARE my_cursor CURSOR FOR my_statement;
  EXEC SQL PREPARE my_statement FROM :my_string;
  EXEC SQL OPEN my_cursor;
   ...
  threadTid interaction -> DissociateTid ->
              xa_end(XID1, TMSUSPEND|TMMIGRATE)
}
```

Figure 35. Setting up the cursor state

The **DoSomeWorkInServer** routine (Figure 36 on page 182) uses the open cursor retained by the resource manager to retrieve data.

```
DoSomeWorkInServer()
{
  threadTid interaction -> AssociateTid -> xa_start(XID1, TMNOFLAGS)
   ...
  /* fetch data using the open cursor */
  EXEC SQL FETCH my_cursor INTO :column1, :column2;
  EXEC SQL UPDATE my_table
       SET field1 = :my_val
       WHERE CURRENT OF my_cursor;
  ...
  threadTid interaction -> DissociateTid ->
              xa_end(XID1, TMSUSPEND|TMMIGRATE)
}
```

Figure 36. Using the open cursor to retrieve data

The **CleanupStateInServer** routine (Figure 37) could be called by the client after issuing all other TRPCs so that server state could be released. Note that the dissociation is still done with the TMSUSPEND flag thereby retaining resource manager state for the transaction.

```
CleanupStateInServer()
{
  threadTid interaction -> AssociateTid -> xa_start(XID1, TMNOFLAGS)
   ...
  /* cleanup any application server state */
   ...
  threadTid interaction -> DissociateTid ->
               xa_end(XID1, TMSUSPEND|TMMIGRATE)
}
```

Figure 37. Releasing the server state

When the client reaches the **onAbort** clause after all the TRPCs in the transaction have completed successfully, TRAN and TRPC cooperate to initiate commitment in the server. As shown in Figure 38 on page 183, the server executes prepare, resolution, and finish callbacks in a manner similar to the previous examples. TM-XA must terminate the suspended association via a call to the **xa_end** function with the TMSUCCESS flag before the transaction can be committed. This **xa_end** call releases the cursor state that was retained earlier.

```
onAbort in client -> initiate commit in server ->
    prepare callback -> xa_end(XID1, TMSUCCESS)
         xa_prepare(XID1, TMNOFLAGS)
    resolution callback -> xa_commit(XID1, TMNOFLAGS)
    /* if heuristic outcomes */
    finish callback -> xa_forget(XID1, TMNOFLAGS)
```

Figure 38. onAbort initiates commitment in the server

If the TMNOFLAGS flag is included in the *endFlag* parameter of the **tmxa_SetAutomaticXaAssociation** function, automatic XA calls are disabled during transaction association and dissociation. The server procedures shown in the preceding example would have to make explicit calls to the **tmxa_AssociateTid** and **tmxa_DissociateTid** functions. In this case, the flag passed to the **xa_end** function would be specified on the call to the **tmxa_DissociateTid** function.

XA flow in distributed TMs

In a distributed online transaction processing (OLTP) system, many hosts can be working on behalf of a transaction. As the transaction moves from machine to machine, threads and resource managers become associated with transactions and execute work on their behalf. When the work is finished, the associations are broken.

To coordinate a transaction among distributed transaction managers, a global identification of the transaction is necessary. XA identifies transactions through the use of the XA transaction identifier (XID). An XID consists of two parts, a global transaction identifier (gtrid) and a branch qualifier (bqual). The TM-XA Service assumes that global transaction identifiers are shared among all participating applications working on the same transaction, but that each application has its own branch qualifier. Therefore, XA transaction identifiers (XIDs) are local to an application. It will never be the case that two different applications (servers) will be manipulating the same XID. Therefore, any XA processing for an XID will be within a single application. When an application associates a thread with the transaction, the application-specific XID is associated with the RMs attached to that application; when the transaction association with a thread is removed, so is the XID's association with the RMs.

In a distributed environment, changes in associations can take place when a transaction enters an application (through reception of an RPC) and when a transaction leaves an application (through transmission of an RPC). There are four distinct situations:

- Sending a request: the current association can be suspended.
- Receiving a request: an association is started.
- Sending a reply: the current association is ended.

- Receiving a reply: the suspended association can be resumed.

These actions are illustrated in Figure 39.

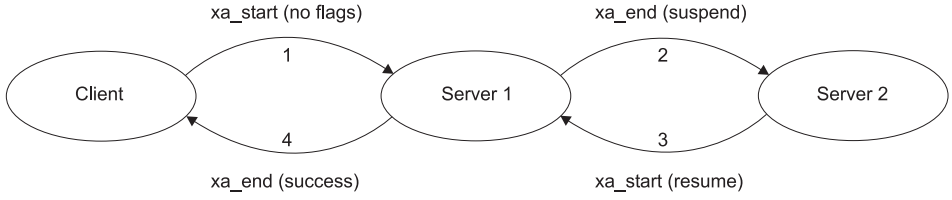

Figure 39. Initial server contact

In the diagram, the client makes transactional RPC calls to Server1, which in turn makes further transactional RPC calls to Server2. The numbers on the arcs show the order of communication.

Because the TM-XA Service relies on a recovery service that is registered with the Transaction Service, the XA processing must take place in a server. Therefore, we consider the XA calls made within Server1 during each of the four situations. When Server1 receives a request from the client, it starts working on a transaction, and therefore starts an association by calling **xa_start** with `TMNOFLAGS` as a parameter.

During its processing, Server1 sends a request to Server2. Normally, the TM-XA Service does not suspend the association during this call. If no other threads in Server1 are contending for the XA resources in Server1, there is no need to pay the overhead of relinquishing and then reestablishing the transaction association with the XA resources. When the reply to Server1's request to Server2 is received, Server1 continues using the resource manager on behalf of the same transaction. Therefore, the XA calls depicted on arcs 2 and 3 would not be made.

However, one might want to free the resource managers in Server1 while it is making a remote call on Server2. This is accomplished by suspending the current association before making the remote procedure call (shown on arc 2) and resuming the association after the call (shown on arc 3). If one wants to achieve this effect for only one call, one can use the **threadTid_Suspend** routine (before the remote procedure call) and the **threadTid_Resume** routine (after the remote procedure call) to implement this behavior. If one wants this behavior on all remote procedure calls made by Server1, then one would register a callback for the outgoing request with **trpc_CallBeforeSendingRequest** and for the incoming reply with **trpc_CallAfterReceivingReply**. In a Transactional-C environment the registered callbacks would call ThreadTid to change the state of the

association between the thread and the transaction. For example, the callback procedure to register for the outgoing request could be the following:

```
static void ThreadTidSuspend(handle, argP, tranInfoP, ifSpecP, messageP)
     rpc_binding_handle_t handle;
     void *argP;
     trpc_tranInfo_t *tranInfoP;
     trpc_ifSpec_t *ifSpecP;
     tran_message_t *messageP
{
     threadTid_Suspend();
}
```

The corresponding callback procedure for the incoming reply could be as follows:

```
static void ThreadTidResume(handle, argP, tranInfoP, ifSpecP, message)
     rpc_binding_handle_t handle;
     void *argP;
     trpc_tranInfo_t *tranInfoP;
     trpc_ifSpec_t *ifSpecP;
     tran_message_t message
{
     (void) threadTid_Resume();
}
```

These callbacks could be registered using the following calls:

```
trpcStatus = trpc_CallBeforeSendingRequest(
     ThreadTidSuspend,
     (void *) NULL,
     TRPC_NULL_CALLBACK_DATA_ID );
trpcStatus = trpc_CallAfterReceivingReply(
     ThreadTidResume,
     (void *) NULL,
     TRPC_NULL_CALLBACK_DATA_ID);
```

When these callbacks are registered, Server1 will suspend the association between the transaction and the RMs until Server1 receives an answer from Server2. (This frees the RMs in Server1 to perform work on other transactions.) On reception of an answer from Server2, Server1 resumes the association of the transaction with the RMs and finishes the work on the transaction. When Server1 returns its answer back to the client, it ends the association with the transaction by calling **xa_end** with the parameter TMSUCCESS.

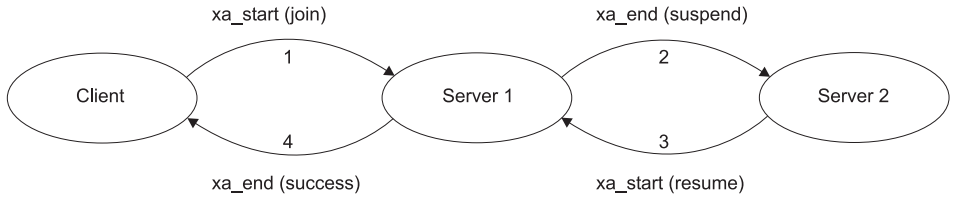

Figure 40. Repeated server contact

Figure 40 shows what would happen when the client makes a second call to Server1 on behalf of the same transaction. Everything is the same except for the parameter used for the **xa_start** call during request reception. Because the transaction has already been seen by Server1, it joins the in-progress transaction.

XA's thread of control

The XA specification is defined in terms of an application's thread of control. There is no requirement within the XA specification that cooperating threads of control share resources or that threads of control must isolate themselves from one another. The only requirement of the XA specification is that a resource manager, an application, and a transaction manager agree on the concept of a thread of control.

Encina applications may consist of several threads. Unlike the XA specification of thread of control, a thread in an Encina application shares all of a program's resources with all other threads, including address space, memory, and communication connections. Typically, a resource manager is available to an Encina application as a resource manager client library that is linked into the application. Therefore, this library is shared among all threads of an application. For the Encina application and the library to work correctly, there must be an agreement on the relationship between an applications's thread and the XA-specified thread of control assumed by the resource manager.

One interpretation is that a thread of control is the same as an application's thread. In this case, each application thread must open the resource manager and each application thread may have its own association with the resource manager. Another interpretation is that the application is a single XA-specified thread of control which is shared among all the application's threads. In this case, the resource manager needs to be opened only once for access by all threads, and only a single association may be made between all of an application's threads and the resource manager. The interface provided by the TM-XA Service uses the former assumption for applications, but supports resource managers that use either interpretation. The interpretation assumed by a resource manager is specified as part of its registration with the TM-XA Service, as described in "Serializing access to RMs" on page 190.

Migration

Although XA permits an XID to be migrated among applications, the TM-XA Service will only perform migration among threads in the same application.

In general, a request to suspend and migrate an XID in a call to **tmxa_DissociateTid** will be honored if the RM has declared itself to support migration, otherwise the association will be ended successfully. If an RM declares itself to support migration and then refuses to migrate when asked, the TM-XA Service will end the association with a `TMSUCCESS` flag and record an error to the standard error stream.

Note that migration is necessary only when a resource manager equates an XA-specified thread-of-control with each of the application's threads. Normally, a resource manager equates the XA-defined thread of control with an application program, and therefore, the application threads share a single XA-defined thread of control. In the normal case, all requests for `(TMSUSPEND | TMMIGRATE)` will be transformed into a "local" suspension (`TMSUSPEND`), thereby allowing applications to act as if the XID is migrating from one thread to another, but allowing the RM library to perform the more efficient local suspension.

Nested transactions

The Transaction Service supports nested transactions. However, the XA specification does not describe any nesting semantics. Therefore, it is necessary to provide an interpretation of the Encina nesting model onto the XA model.

The current design specifies four alternatives. One may map all nested transactions into the same XID, allow each subtransaction to have its own *gtrid* or allow each subtransaction to share the *gtrid* component of the XID but to have its own branch (*bqual*). In the last case, one may choose to determine the outcome of each branch independently or not. Note that transaction outcome (commit or abort) is linked differently than heuristic outcomes (damage). The implications of each choice are shown in Table 6.

Table 6. Semantics of nesting transactions using XA

Model	TID	XID	isolation (locks)	abort linkage	damage linkage	excess starts
TMXA_DIFFERENT_GTRID I	1.1 1.2	B.0 C.0	too much	no	no	yes
TMXA_SAME_XID II	1.1 1.2	A.0 A.0	too little	yes	yes	no
TMXA_DIFFERENT_BQUAL_ INDEPENDENT III	1.1 1.2	A.1 A.2	unclear	maybe	no	yes

Table 6. Semantics of nesting transactions using XA (continued)

Model	TID	XID	isolation (locks)	abort linkage	damage linkage	excess starts
TMXA_DIFFERENT_BQUAL _LINKED IV	1.1 1.2	A.1 A.2	unclear	yes	no	yes

The notation used in Table 6 on page 187 is as follows. For each model, two nested transactions are shown, labeled 1.1 and 1.2. The parent transaction 1 is assumed to be mapped to an XID that has a global transaction identifier (*gtrid*) of A and a branch qualifier (*bqual*) of 0, denoted A.0. Model I asserts that each nested transaction gets its own global transaction identifier (*gtrid*) (branch qualifiers not meaningful here). Model II states that nested transactions use the same *gtrid* and *bqual* as the top most ancestor of a transaction tree. Model III states that all nested transactions within a transaction tree share the same *gtrid*, but get unique *bqual* values. Model IV uses the same encoding of the transaction as model III, but applies different semantics when aborting a nested transaction. In model III, nested transactions (XIDs with different *bqual* values) may be aborted independently. In model IV, aborting an XID will cause all XIDs with the same *gtrid* to be aborted.

To accommodate all four models and the need of the recovery system to identify which XIDs belong to a particular application, the constructed XIDs will use the format shown in Table 7 (when encoded using the Encina format):

Table 7. XID encodings for nested transactions

Model	gtrid	bqual	
I	GTID	1 || APPLID	
II	TLGTID	2 || APPLID	
III	TLGTID	3 || APPLID	GTID
IV	TLGTID	4 || APPLID	GTID

In Table 7, GTID refers to the (Encina-generated) universally unique identification of transaction. This value is available as the value of the `TRAN_PROPERTY_KEY_GLOBAL_IDENTIFIER` property. TLGTID refers to the (Encina-generated) universally unique identification of the top-level ancestor of transaction tree. APPLID refers to the universally unique identification of an application, and `||` indicates concatenation.

XID assignment

XA identifies transactions through the use of an XID. In Encina, the format of the XID assigned to a transaction depends on the nesting model being used to

represent nested transactions. When the TM-XA Service encounters a top-level transaction, the default nesting model (Model 1) is assumed; a subtransaction assumes the nesting model of its parent. The nesting model can be changed for top-level transactions with the **tmxa_SetNestingModel** function.

The TM-XA Service does not assign an XID until an association between a transaction and a resource manager is made or until an XID is explicitly requested with the **tmxa_GetXidForTid** function. If a transaction does not use a resource manager, or if no XID is requested, an XID is not created. Once an XID is assigned for a transaction, the nesting model for that transaction cannot be changed.

Rollback linkage

When a transaction aborts, calls of **xa_rollback** are made on its corresponding XID. Further, the TM-XA Service will attempt to abort all linked transactions as per table 1. However, three circumstances complicate the abort linkage. First, the nesting model used for nested transactions may change. Therefore, the model that would be used for a particular subtransaction may not be known when a potentially linked transaction aborts. Second, not all transactions perform XA work. Some nested transactions may perform work with non-XA resource managers and therefore have no XA semantics (or XIDs) associated with them. Third a transaction may not be eligible to be aborted at the site where a linked transaction aborted. The Transaction Service limits the ability of applications to abort a transaction. Applications may only abort transactions that have performed work for them, which is defined as transactions begun by the application or transactions received in an RPC request. For example, assume a client starts a transaction (T1) and a nested transaction (T2), and then makes an RPC on behalf of T2 to the server. The server receives T2 and starts another nested transaction T3. At this point the server may abort T2 or T3, but it is not permitted to abort T1. However, if all three transactions are sharing the same XID (model TMXA_SAME_XID) then aborting one should abort all three.

When the need to abort a transaction arises, the TM-XA Service will examine each transaction in a family, and each transaction that would be linked if it had been assigned an XID will be marked for rollback. Transactions that already have XIDs associated with them will be aborted immediately; all other linked transactions will be aborted by the TM-XA Service only if they attempt to perform XA work, defined as making an association between the transaction and resource managers. If the transactions marked for abortion never do any XA work, they will not be explicitly aborted by the TM-XA Service.

Heuristic damage linking

Linking transactions that are heuristically damaged has the same complications as linking aborted transactions, and the same algorithms are used. However, damage is linked only among subtransactions sharing the same XID (model `TMXA_SAME_XID`).

Header files and libraries

Header files

The file **tmxa/tmxa.h** contains the TM-XA interface declarations for the C language. This file must be included in any file that uses TM-XA interface functions or data types.

Libraries

The Encina TM-XA functions are in the **EncServer** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Initialization and termination

The TM-XA Service provides the ability to register resource manager instances (RMI) that require transaction management via the X/Open XA interface. All such resource manager instances are registered using the **tmxa_RegisterRMI** function.

After all the resource manager instances have been registered, you should initialize the TM-XA Service using the **tmxa_Init** function. Normally, the TM-XA Service initializes and terminates a resource manager (RM) automatically through the use of **xa_open** and **xa_close** calls, but an application can control the opening and closing of RMs directly by using the **tmxa_OpenRMIs**, **tmxa_CloseRMIs**, and **tmxa_SetRMIOptions** functions. An application can query the options set for a resource manager with the **tmxa_GetRMIOptions** function.

Serializing access to RMs

"XA's thread of control" on page 186 describes two interpretations of XA-specified thread-of-control: either an application thread or an application. The value of the *threadSupport* parameter of the **tmxa_RegisterRMI** function is used to specify which of the two interpretations is used by the resource manager.

If the value of the *threadSupport* parameter is `TMXA_SERIALIZE_ALL_XA_OPERATIONS`, the TM-XA Service permits at most one thread to make an XA call to the resource manager client library at a time. In the absence of local transactions, this value also serializes native operations to

the resource manager client library. This is the value to be specified if a resource manager client library interprets an application to be a single XA-specified thread of control.

The XA specification describes a resource manager implementation where simultaneous requests of several threads of control are sent to a common back-end process. The back-end process may choose to serialize some of these requests and handle others in parallel. Because the TM-XA Service interprets each application thread as an XA-specified thread of control, a shared resource manager client library can assume some of the roles normally relegated to a back-end process while still providing the interpretation that an application is a single XA-specified thread of control. Two such approaches may be specified by the *threadSupport* parameter.

If the value of the *threadSupport* parameter is `TMXA_SERIALIZE_START_END`, the TM-XA Service assumes that the resource manager client library permits simultaneous calls of all XA operations by the transaction manager except for transaction association (**xa_start**, for example). Thus, the TM-XA Service ensures that only one association with the resource manager client library is attempted at a time. However, the TM-XA Service permits simultaneous calls of other XA operations to be made from other threads. For example, while one thread is calling the **xa_start** function, another could be calling the **xa_prepare** function. The RM client library must coordinate these calls as appropriate (by serializing them, for example).

If the value of the *threadSupport* parameter is `TMXA_SINGLE_ASSOCIATION`, the TM-XA Service does not prevent multiple threads from attempting different associations at the same time, but it assumes that the RM client library accepts only one association at a time. This might be accomplished, for example, by the RM client library blocking an **xa_start** function in one thread if an association in another thread has been made. However, the RM client library is acting as though a single XA-specified thread is using it. Thus, for example, an association suspended (without migration) in one thread would have to be resumed in another thread rather than joined.

If the value of the *threadSupport* parameter is `TMXA_MULTIPLE_ASSOCIATIONS`, the TM-XA Service assumes that the resource manager is treating each application thread as an XA-specified thread of control. Therefore, the TM-XA Service performs no serialization of XA operations between threads. Some implications of non-serialized XA operations include the following:

- Each thread can have its own association.
- A suspended association must be migrated to be resumed in another thread.
- An association suspended (without migration) in one thread can be joined in another thread.

- An association in one thread can be rolled back by another thread before the first thread ends the association.
- Any thread can perform the completion protocol for a transaction (but only one thread actually does so).

Note, however, that the XA specification explicitly permits the RM back end to serialize associations made by multiple processes. Therefore, a legal implementation for a client library might be to serialize the **xa_start** calls from different threads.

In the case where the value of the *threadSupport* parameter of the **tmxa_RegisterRMI** function is equivalent to `TMXA_SERIALIZE_ALL_XA_OPERATIONS`, the TM-XA Service serializes access to the RM client library between threads. The operations to be serialized are **xa_open**, **xa_close**, **xa_prepare**, **xa_commit**, **xa_rollback**, **xa_forget**, **xa_recover**, **xa_start** and **xa_end** pairs, **ax_reg** and **xa_end** pairs, and **ax_reg** and **ax_unreg** pairs.

Note: The **ax** routines are exported by a transaction manager for use by dynamic resource managers. The **ax_reg** and **ax_unreg** functions are specified as part of the X/Open XA standard.

If serialization is provided, it is enforced between threads and not within a thread. For example, once a thread issues an **xa_start**, no other thread can issue an XA call until the first thread finishes by calling **xa_end**. However, a call to **xa_rollback** can be performed in the same thread between calls to **xa_start** and **xa_end**. (The XID specified in the call to **xa_rollback** is not the same as the XID specified in the call to **xa_start**.)

There are several implications of this serialization strategy. First, if a thread blocks on an RM client library call in the midst of an **xa_start** and **xa_end** pair (for example, while waiting for a lock that another transaction in another thread holds), no other XA work is performed by that application until the thread attempting to get the lock unblocks. For example, if another thread tries to commit (call to **xa_commit**) or abort (call to **xa_rollback**) a transaction (in an attempt to release the lock), the second thread blocks waiting for the first thread to execute an **xa_end** call, resulting in deadlock.

Second, native calls to the resource manager client library on behalf of global transactions are serialized. A global transaction is initiated by the call to **xa_start**, which results in an initial internal TM-XA Service lock being acquired by the thread. No other thread can obtain this lock for performing the **xa_start**, so no other thread is able to perform native calls to the resource manager on behalf of a global transaction. An application might attempt to make native calls to the resource manager in another thread without making an association. Technically, this is a local transaction and is not permitted to

coexist with a global transaction, but the TM-XA Service does not prevent such calls. Only the thread that successfully started is permitted to continue and make native RM calls.

Third, to accommodate dynamic resource managers, the internal TM-XA Service locks are taken on the potential of an association being made. When an association is specified by a call to **tmxa_AssociateTid**, the TM-XA Service assumes that dynamic resource managers register themselves via calls to **ax_reg**. Therefore, the TM-XA Service acquires the locks to serialize access to the resource manager by the thread, but this supposition correctly serializes access only for calls that are part of global transactions.

Fourth, because access is only serialized correctly for calls that are part of global transactions, there is a small window of contention for dynamic RMs when executing *local* transactions. This is because the serialization of local transactions starts in the call to **ax_reg**, not when the potential for a call to **ax_reg** begins. Thus, it is possible that one thread may start a native call to the RM, which will eventually call **ax_reg**, while another thread is making another call to the RM (either on behalf of XA, such as an **xa_prepare** call, or simply another local transaction). Two threads are using the RM at the same time, since there was no implied serialization between the thread making a native call (which cannot be intercepted by the TM-XA Service) and any other thread. Applications that use threads, dynamic resource managers, and local transactions must take responsibility for synchronizing the use of the RM client library properly.

Because different resource managers might provide different interpretations of XA's thread of control, the TM-XA Service enforces the most restrictive requirement of the registered resource manager instances. Thus, if any resource manager instance registers itself using the `TMXA_SERIALIZE_ALL_XA_OPERATIONS` value, then all XA calls to all resource manager client libraries are serialized and at most one association is attempted in any thread. Similarly, if no resource manager instance registers itself using the `TMXA_SERIALIZE_ALL_XA_OPERATIONS` value, but at least one registers using `TMXA_SERIALIZE_START_END`, then the TM-XA Service serializes all associations for all resource manager client libraries.

If all resource manager instances register using either `TMXA_SINGLE_ASSOCIATION` or `TMXA_MULTIPLE_ASSOCIATIONS` (and at least one registers using `TMXA_SINGLE_ASSOCIATION`), then the TM-XA Service does not serialize associations. To permit multiple associations to be made simultaneously, all resource manager instances must register using `TMXA_MULTIPLE_ASSOCIATIONS`.

Providing a recovery service

For correct operation of the TM-XA Service, applications must provide a recovery service for the Transaction Service to use. The Encina Recovery Service (REC) can be used for this purpose. Figure 41 shows a code fragment for initializing the Recovery Service when an application manages no recoverable data directly (that is, all recoverable data is managed by resource managers).

```
/* Begin toolkit initialization, and initialize TRPC */
preInitTC();
tc_InitTRPC();
/* Initialize TM-XA */
status = tmxa_RegisterRMI(OPEN_STRING, CLOSE_STRING,
            &infx_xa_switch, THREAD_SUPPORT, &rmid);
status = tmxa_Init();
/* Initialize the recovery service */
rec_Init(configParams.logFile, configParams.logAuthnLevel, NULL,
            NULL, NULL, NULL, NULL, NULL, NULL, NULL);
rec_RecoverVolumes();
/* Finish toolkit initialization */
postInitTC();
```

Figure 41. Simple Recovery Service initialization

If an application manages recoverable data, then the NULL arguments in the **rec_Init** call shown in the figure must be replaced with more meaningful values. Refer to "Chapter 8. Log Service (LOG)" on page 119 and "Chapter 10. Recovery Service (REC)" on page 143 for details.

In the Encina Monitor environment, the initialization of TM-XA, REC, and LOG is handled automatically when the **mon_ServerRecoverable** function is called. The Monitor also uses **tmxa_SetRMIAutoClose**(TMXA_CLOSE_ON_THREAD_EXIT) to cause RM instances to close automatically on thread exit. See the *Encina Monitor Programming Guide* for more information.

Transaction context

Associations between a transaction and resource manager instances are controlled by the use of **tmxa_AssociateTid** and **tmxa_DissociateTid**. These calls are used to make and break, respectively, associations. The TM-XA Service also provides the following related functions:

- **tmxa_GetXidForTid**
- **tmxa_SetAutomaticXaAssociation**
- **tmxa_SetNestingModel**
- **tmxa_SetUsesOnlyLocalXaWork**

Most programmers need not use these calls directly. Typically, calls to these procedures are performed during callbacks of other internal functions in the TM-XA Service. However, the programmer is free to disable the automatic association provided by these callbacks and perform the associations explicitly. Care should be taken to follow the restrictions of the use of these calls.

Use of XA by the TM-XA Service

The TM-XA Service does not exploit all possible facilities in the XA specification. This section discusses how some features in the XA standard are used in this implementation, along with their potential use in the future.

1. The flag `TMASYNC` will not be used. No asynchronous calls are used or foreseen. Because Encina is a threads-based system, it is assumed that other threads are available to run if a call to an XA procedure blocks.
2. The flag `TMFAIL` will not be used in the first release. Resource managers will be informed of TM-initiated aborts by calls to **xa_rollback**. If needed, `TMFAIL` may be used in future releases.
3. The flag `TMENDRSCAN` will not be used. Once a scan is started, the TM will continually invoke **xa_recover** until the RMI indicates that no more transactions are available (by returning fewer XIDs than the available space provided by the TM).
4. The flag `TMMULTIPLE` will not be used. No asynchronous calls are used.
5. The procedure **xa_complete** will not be called. Because no asynchronous calls will be made, there is no need to check for their completion.
6. The TM-XA Service assumes it can join transactions within an application program, but will not join a transaction across an application boundary (an application boundary is defined as the scope of the *applID* value). However, different branches of a transaction may be used in different (cooperating) application programs.
7. Migration is desired to be absolute: if an RM promises to migrate, it should honor that request on demand, and not return `NOMIGRATE` when asked to suspend/migrate. (See "Transaction context" on page 194 for a discussion of what will happen in this case.)
8. The TM-XA Service assumes that it need not close a resource manager instance for the RMI to operate correctly. If asked, the TM-XA Service will attempt to close each RMI at most once (per successful open).
9. The TM-XA Service assumes that it may open an RMI any number of times without ill effect. It will, however, attempt to open each RMI exactly once.
10. The TM-XA Service uses branch identifiers, and under certain circumstances (specifically, model III – see section "Nested transactions" on page 187), assumes that branches may be independently committed or rolled back.

11. Heuristic outcomes that match the desired outcome will be immediately forgotten.
12. An RMI may be asked to prepare, commit, rollback or forget one transaction in the midst of a start/end pair of a different transaction.
13. An RMI may be asked to end successfully a suspend association in the midst of a start/end pair of a different transaction.
14. The TM-XA Service may call **xa_rollback** within a call to **ax_reg**. The call to **xa_rollback** may use the `XID` to be returned by **ax_reg** call. When **ax_reg** calls **xa_rollback**, the return from **ax_reg** will be `TMER_TMERR`.
15. The TM-XA Service may have multiple transactions that are suspended (without migration). It resumes and suspends these transactions in any order and any number of times.
16. An RMI may be asked to prepare, commit, rollback or forget one transaction after suspending (without migration) another transaction.
17. If two branches are being used, each branch is expected to participate fully in the 2PC protocol. In particular, even if one branch prepares and commits, the TM-XA Service expects the RM to return `XA_OK` to requests to prepare and commit the other branch.
18. The Encina XID format ID is 113577.

Diagnostics

This section documents the diagnostic support provided by the TM-XA Service. This support takes the form of recoverable or unrecoverable erroneous events (warnings or fatal-errors) during the use of the TM-XA Service, informative messages that trace calls on the TM-XA Service (tracing) and snapshots of the state of the TM-XA Service (state dumps).

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring-buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

The TM-XA Service generates a *fatal run-time error* whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility, and the application's execution terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility. This section defines the fatal error

messages that may be output by the TM-XA Service, and briefly describes why they occur, and how they may be avoided or remedial action taken.

The TM-XA Service has the following fatal error messages:

- `Could not calculate XIDs of prepared transactions during recovery.`—The TM-XA Service was unable to calculate the XIDs of in-flight (prepared) transactions returned from the Transaction Service. Therefore, there is no way to compare the XIDs of transactions known by the Transaction Service with the XIDs known by the resource managers. No action will be taken on the prepared transactions pending in the resource managers until the application program is started again. The program exits since recovery by the Transaction Service of its in-flight transactions without corresponding recovery by the TM-XA Service may result in erroneous rollbacks of resource manager in-flight transactions. The application should be rerun. If the problem recurs, contact Encina Product Support.
- `Could not create an XID for this transaction: transaction identifier`—An internal inconsistency within the TM-XA Service has resulted in its inability to calculate a XA-compliant XID for this transaction. Because the inconsistency implies program corruption, it exits. There is nothing an application programmer can do to fix this problem. A possible work around is to specify a smaller application identifier to the Transaction Service. However, this work around will cause all current in-flight (prepared) transactions to not be recovered. This error should be reported to Encina Product Support.
- `Could not obtain application identifier.`—The TM-XA Service was unable to obtain the application identifier from the Transaction Service. This should never happen since the application identifier is established before the TM-XA Service requires it. Therefore, this error reflects a corruption of the application, and it exits. There is nothing an application programmer can do to fix this problem. A possible work around is to specify a smaller application identifier to the Transaction Service. However, this work around will cause all current in-flight (prepared) transactions to not be recovered. This error should be reported to Encina Product Support.
- `Could not obtain from TRAN a list of prepared transactions during recovery. Recovery aborted.`—The TM-XA Service was unable to get the list of in-flight (prepared) transactions from the Transaction Service during recovery processing. Therefore, there is no way to coordinate the list of prepared transactions known by the Transaction Service with the lists of prepared transactions known by the resource managers. No action will be taken on the prepared transactions pending in the resource managers until the application program is started again. The program exits since recovery by the Transaction Service of its in-flight transactions without corresponding recovery by the TM-XA Service may result in erroneous

rollbacks of resource manager in-flight transactions. The application should be rerun. If the problem recurs, contact Encina Product Support.

- `Internal error: unknown last start flag` *`number`*.—Internal data maintained by the TM-XA Service appears to have been corrupted. Contact Product Support.
- `Internal error: unknown last xa end flag` *`number`*.—Internal data maintained by the TM-XA Service appears to have been corrupted. Contact Encina Product Support.
- `RM` *`resource manager id`* `returned` *`number`* `XIDs (which is more recovery information that requested) overwriting storage.`—A resource manager provided more recovery information than the TM-XA Service requested. The TM-XA Service allocates only enough space for the recovery information that it is prepared to handle. Therefore, the resource manager has overwritten some part of the data or program of the application server. Because the application has been potentially corrupted, it exits. The behavior reflects a defect in the resource manager and should be reported to the resource manager vendor.
- `ThreadTid callback event inconsistent with internal state (callback event number` *`event number`*`)`—An internal inconsistency between the TM-XA Service and the ThreadTid Service has been detected. This should never happen and indicates either a version skew between the two services or corruption of a service. Because the application has been potentially corrupted, it exits. There is nothing an application programmer can do to fix this problem. A work around is to disable the automatic XA association facilities of the TM-XA Service and use the **tmxa_AssociateTid** and **tmxa_DissociateTid** calls directly (which will avoid use of the ThreadTid Service altogether). This error should be reported to Encina Product Support.

Warning messages

Warning messages describe recoverable events that cannot be communicated by ordinary programming means, such as return codes. The TM-XA Service generates five different warning messages:

- `Gtrid too long to encode: recovery compromised` *`length`*.—The TM-XA Service was unable to generate a preliminary-XA compliant GTRID that it could use to identify transactions and be recognizable after a failure. The TM-XA Service uses an algorithm that is a function of the nesting model and the XA-compliant XID structure to create a preliminary-XA GTRID. If the XID is too long, it cannot be compressed into a GTRID and this warning is printed. The transaction is not aborted; should the system fail, the in-flight transactions in the preliminary-XA compliant resource manager will be aborted since the GTRIDs will not match any in-flight transactions from the Transaction Service. The application should specify a smaller

application identifier for the application that is starting the transaction (not necessarily the application communicating with the preliminary-XA compliant resource manager.)

- `Heuristic outcome performed on transaction XID xid`.—An in-flight transaction from a resource manager has taken a heuristic decision during recovery. Specifically, the transaction is not recognized by the Transaction Service, and is therefore rolled back as required by the presumed-abort protocol. The resource manager has responded to the rollback request by returning a heuristic commit, heuristic mixed or heuristic hazard return code. Because the Transaction Service has no corresponding transaction to associate with the heuristic result, the warning message is printed. The application programmer should perform whatever maintenance procedures are required when resource managers perform heuristic decisions.
- `Resource manager could not abort XID xid`.—A resource manager has refused to rollback a transaction during recovery. Specifically, the transaction is not recognized by the Transaction Service, and is therefore rolled back as required by the presumed-abort protocol. The resource manager has responded to the rollback request by refusing to rollback the transaction. Because the Transaction Service has no corresponding transaction to associate with the rollback refusal, the warning message is printed. The application should be rerun. If the problem persists, the application programmer should perform whatever maintenance procedures are required when resource managers perform heuristic decisions.
- `Resource manager could not forget heuristic decision for XID xid`.—A resource manager has refused to forget a heuristic decision it took for a transaction during recovery. Specifically, the transaction is not recognized by the Transaction Service, and is therefore rolled back as required by the presumed-abort protocol. The resource manager has responded to the rollback request by returning a heuristic rollback, heuristic commit, heuristic mixed or heuristic hazard return code. In response, the TM-XA Service has requested the resource manager to forget the heuristic decision; the resource manager refuses. Because the Transaction Service has no corresponding transaction to associate with a prolonged transaction completion for retrying the forget request, the warning message is printed. The application programmer should perform whatever maintenance procedures are required when resource managers perform heuristic decisions.
- `Truncating application identifier when constructing XID -- recovery compromised.`—The TM-XA Service was unable to generate an XA-compliant XID that it could use to identify an transactions and be recognizable after a failure. The TM-XA Service uses an algorithm that is a function of the transaction's Transaction Service identifier and the application's application identifier to create XIDs. If the identifiers are too large, the calculated XID will not fit into an XA-specified XID structure. The transaction will be aborted immediately and this warning printed. The

application should specify a smaller application identifier for both the application starting the transaction and the application communicating with the XA-compliant resource manager.

Audit messages

The TM-XA Service does not emit any audit messages.

Abort reasons

To abort transactions, the TM-XA Service uses the Encina Abort Facility, which is documented in "Chapter 5. Abort Facility" on page 91. The Encina Abort Facility associates each abort reason with an abort code and a brief description, which are specified in the **tmxa_abort_t** data type. TM-XA uses the following abort reasons:

`TMXA_PROTO_ERROR_CODE`: "The TM-XA protocol was violated."

`TMXA_XA_ERROR_CODE`: "A TM-XA call failed."

`TMXA_MODEL_LINKED_CODE`: "A transaction linked with this one failed."

`TMXA_INTERNAL_ERROR_CODE`: "An error was detected in TM-XA internal data structure."

`TMXA_TRAN_ERROR_CODE`: "A Transaction Service call failed."

`TMXA_MISSING_RECOVERY_SERVICE_CODE`: "Application is not recoverable."

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as calls are made on the TM-XA Service. All TM-XA Service interface functions are capable of tracing their entry, parameters, and exit. In addition, important events during the execution of a TM-XA function, for example, XA calls, may be traced. Clients of the TM-XA Service can enable a specific class of event tracing, such as entry/exit, or a collection of events (entry/exit and parameter tracing).

The level of tracing is controlled by the value of a single variable that is exported by the TM-XA Service:

```
unsigned long tmxa_traceMask
```

The Encina Toolkit Trace Facility defines a number of ″bit constants″ that when set in the above variable cause a specific form of tracing to occur. For example, the TRACE_ENTRY constant enables entry/exit tracing of TM-XA Service interface functions. The TM-XA Service also defines TM-XA Service-specific bit constants for important classes of internal events.

The following section describes the different methods for setting the value of the above variable to obtain the required level of tracing, and is followed by a description of the levels of tracing supported by the TM-XA Service.

Global trace levels

The TM-XA Service supports the global trace levels defined by the Encina Toolkit Trace Facility. These levels are as follows:

- TRACE_ENTRY — traces the entry and exit of functions that make up the TM-XA Service interface.
- TRACE_PARAM — traces the parameters passed to the interface functions.
- TRACE_EVENT — traces all events in the TM-XA Service. This trace class outputs large amounts of information, and for this reason, is not always recommended.

In addition, the following two trace constants are provided for enabling all, or no, tracing (respectively):

- TRACE_GLOBAL
- TRACE_NONE

TM-XA Service internal events

The TM-XA Service provides three internal trace levels.

Tracing XA activities

The level TMXA_TRACE_XA can be used to enable tracing of XA and AX calls. When enabled, every XA call made by the TM-XA Service to a resource manager will be traced. Similarly, all AX calls from a resource manager to the TM-XA Service will be traced. Tracing is provided before the calls to output the parameters to be used in the call, and tracing is provided after the calls to report the return codes and values of output parameters.

Tracing locking activities

The level TMXA_TRACE_LOCK can be used to enable tracing of internal TM-XA Service locks. These locks are used to serialize access to resource managers that do not support multi-threaded operations. Other locks are used to coordinate local transactions registered by dynamic resource managers and to coordinate access to global data structures within the TM-XA Service.

Tracing callback activities

The level TMXA_TRACE_CALLBACK can be used to enable entry, exit and parameter tracing of callback procedures that the TM-XA Service has registered with the Transaction Service, ThreadTid Service and threading system.

State dump

The TM-XA Service provides the **tmxa_DumpState** function to dump the state of the TM-XA Service at an application.

The dump provides the following information:

- A snapshot of the global variables used by the TM-XA Service

- A description of all known resource managers
- A description of the state of all known XIDs
- A description of the state of all known transactions
- A description of the state of all resource managers
- A description of the state of all known transaction associations between resource managers and the TM-XA Service

An annotated example of a dump follows:

```
17 D   Starting state dump for TM-XA Service
```

The first table describes the available resource managers.

```
17 D   Switch Table (2 entries)
17 D   rmid      threadSupport      switchP      openInfo/closeInfo
17 D   0         0                 000904A8     000BB088/000A0208
17 D   1         3                 000904F8     000BB588/000BB208
```

The global variables used by the TM-XA Service are listed next.

```
17 D  tmxa_traceMask: 0, tmxa_serializeFlag: 1, tmxa_closeFlag: 0
17 D  tmxa_defaultXaUse:1,tmxa_allRmsShareState:0,tmxa_allRmsThreadBased:0
17 D  tmxa_defaultNestingModel: 1, tmxa_defaultTidUsesOnlyLocalXaWork: 0,tmxa\
                   _tmxaSlot: 150
17 D  tmxa_applIdDataP: 00107D68, tmxa_applIdLen: 1
17 D  tmxa_abortKeyValue length 29 tmxa_abortKeyValue data 00107E28
17 D  numAssocCreations 4, numAssocDeletions 0
17 D  numXidCreations 2, numXidDeletions 0
17 D  numTidCreations 2, numTidDeletions 0
17 D  numThreadSlotCreations 2, numThreadSlotDeletions 0
```

Figure 42. Global variables used by the TM-XA Service

The state of every transaction known by the TM-XA Service is recorded in the transaction identification table.

```
17 D  Tid Table Entries
17 D  EntryP  Tid    Top tid     Reference Count
17 D  |       |      |           | Abort Callback executed
17 D  |       |      |           | | Abort requested
17 D  |       |      |           | | | Damaged
17 D  |       |      |           | | | | Damage Type
17 D  |       |      |           | | | | | Damage Requested
17 D  |       |      |           | | | | | | XA Enabled
17 D  |       |      |           | | | | | | | Children XA Enabled
17 D  |       |      |           | | | | | | | | Local XA Work Only
17 D  |       |      |           | | | | | | | | | Children Local XA Work Only
17 D  |       |      |           | | | | | | | | | | Nesting model
17 D  |       |      |           | | | | | | | | | | | Children Nesting Model
17 D  |       |      |           | | | | | | | | | | | | Dead Entry
17 D  |       |      |           | | | | | | | | | | | | | Recovery Pending
17 D  |       |      |           | | | | | | | | | | | | | |Last Starter Thread
17 D  |       |      |           | | | | | | | | | | | | | | |LastStrtThreadCnt
17 D  |       |      |           | | | | | | | | | | | | | |         |XIDTblPt
17 D  |       |      |           | | | | | | | | | | | | | |         |    |
17 D 00185A08: 00010000 00010000 1 0 0 0 0 0 1 1 0 0 1 1 0 0 00000000 000 00185
17 D 00347108: 00000001 00000001 1 0 0 0 0 0 1 1 0 0 1 1 0 0 00000000 000 00347
```

Figure 43. TM-XA transaction states

The state of every XID known by the TM-XA Service is recorded in the XID table, which is referenced by the transaction identification table. Note that some XIDs may be multiply referenced because of transaction nesting semantics. Each XID entry has an array of resource manager state that describes the global relationship between a resource manager instance and the XID. An example global relationship is whether the XID has been prepared or not. Each resource manager state also has a list of association entries that describe the state of the association between a specific resource manager instance and the XID. For resource managers that registered themselves with `TMXA_MULTIPLE_ASSOCIATIONS`, each Encina application thread is another XA-specified thread-of-control, and therefore association state is maintained on a thread basis.

```
17 D  Xid Table Entries
17 D  EntryP   FormatID   GtridLen   BqualLen   tidsUsingXid   localXaCount
17 D  Gtrid
17 D  Bqual
17 D  00347408:    113577       7              2         1            0
17 D  00000001010158
17 D  0158
17 D  RmXid Entries
17 D rmid rmXidEntryP EverStarted Prepare Commit Rollback Forget  Migrating
17 D    0     00347308         1    FFFFFF35  FFFFFF35  FFFFFF35  FFFFFF35 0
17 D     Association entries
17 D     00347388
17 D     1   00347324         1    FFFFFF35  FFFFFF35  FFFFFF35  FFFFFF35 0
17 D     Association entries
17 D     00347588
17 D  00185B08:    113577       7              2         1            0
17 D  00010000010158
17 D  0158
17 D  RmXid Entries
17 D rmid rmXidEntryP EverStarted Prepare Commit RollbackForget Migrating
17 D    0   00185608         1       FFFFFF35  FFFFFF35  FFFFFF35  FFFFFF35 0
17 D       Association entries
17 D       00185988
17 D    1  00185624         1       FFFFFF35  FFFFFF35  FFFFFF35  FFFFFF35 0
17 D      Association entries
17 D      00185E88
```

Figure 44. Thread association between a specific resource manager instance and the XID

The association table gives the state of the association between an XID and a thread-specific resource manager instance. Resource managers that are not thread-specific, such as those registered with `TMXA_SERIALIZE_ALL_OPERATIONS`, share a single association record. The *rmXidOwner* field specifies which global XID and RM state this association is part of, and the *rmThreadOwner* specifies which thread-based RM this association is part of.

```
17 D  Association Table Entries
17 D  EntryP Alive rmThreadOwner rmXidOwner StartFlag StartResult EndFlag  \
                  EndResult refCount
17 D      suspending thread   suspending tid     nextRmP     nextXidP
17 D  00185E88: 1  000CE108      00185624   00000000  00000000    02000000 \
                  00000000     2
17 D      1                   00010000           00000000    00000000
17 D  00347588: 1  00387E08      00347324   00000000  00000000    FFFFFF35 \
                   FFFFFF35     2
17 D      0                   00000000           00000000    00000000
17 D  00347388: 1  00094EE8      00347308   00000000  00000000    FFFFFF35 \
                   FFFFFF35     1
17 D      0                   00000000           00000000    00000000
17 D  00185988: 1  00094EE8      00185608   00000000  00000000    02000000 \
                   00000000     1
17 D      1                   00010000           00000000    00000000
```

Figure 45. Association Table Entries

The XID-unrelated state of a resource manager is kept in an RM state table. This includes information as to whether the resource manager is open. Also, all associations that this resource manager is involved in is linked together. There is a global resource manager state table and a thread-specific resource manger state table. Although both tables contain entries for all resource managers, the thread-specific table contains information for the thread-specific resource managers and the global table contains accurate information for global resource managers. To find the correct table, examine the *StubPP* field of a thread-specific resource manager table. It will refer to either the resource manager state of the thread (for a thread-specific resource manager) or to the corresponding entry in the global resource manager table (for global resource managers).

```
17 D  Global/Shared Rm State Table
17 D  Entry     assocP    &curLocalState  curLocalStateP  clsP->numOutOfTran \
                  clsP->oftLock
17 D  00094EE8: 00000000  00094EE8        00094EE8        0                  \
                  <unimpl>
17 D      Association entries
17 D  RM Stub state
17 D  rmid  &StubP[rmid] StubPP[rmid]  rmOpened  rmStarted  lastOpenResult \
                  lastCloseResult
17 D  0     00107A08     00107A08      1         00347408   00000000       \
                  FFFFFF35
17 D  1     00107A18     00107A18      0         00000000   FFFFFF35       \
                  FFFFFF35
```

Figure 46. Global/Shared Resource Manager State Table

The thread-specific state has a list of all associations made in that thread for all resource managers, keeps state about local transactions within that thread,

and has a thread-specific resource manager state table. The thread-specific state is linked together across threads so that any thread may dump the contents of all threads.

```
17 D  Thread-specific state
17 D  00387C88: TranId 1  Thread 17  Next   00107F88  Prev  00000000  \
                 Rm State Table
17 D  Entry     assocP    &curLocalState  curLocalStateP  clsP->numOutOfTran \
                  clsP->oftLock
17 D  00387E08: 00347588  00387E08        00094EE8        0                  \
                  <unimpl>
17 D      Association entries
17 D      00347588
17 D  RM Stub state
17 D  rmid  &StubP[rmid]  StubPP[rmid] rmOpened  rmStarted  lastOpenResult \
                lastCloseResult
17 D  0     00387708      00107A08     1         00347408   00000000       \
                FFFFFF35
17 D  1     00387718      00387718     1         00347408   00000000        \
                FFFFFF35
17 D  00107F88:  TranId 0   Thread 1  Next  00000000  Prev  00387C88 \
                Rm State Table
17 D  Entry     assocP    &curLocalState  curLocalStateP  clsP->numOutOfTran\
                clsP->oftLock
17 D  000CE108: 00185E88  000CE108        00094EE8        0                   \
                <unimpl>
17 D      Association entries
17 D      00185E88
17 D  RM Stub state
17 D  rmid  &StubP[rmid]  StubPP[rmid] rmOpened  rmStarted  lastOpenResult \
              lastCloseResult
17 D  0     000CE188      00107A08     1         00347408   00000000     \
              FFFFFF35
17 D  1     000CE198      000CE198     1         00000000   00000000       \
              FFFFFF35
17 D  Ending state dump for TM-XA Service
```

Figure 47. Thread-specific states across threads

Chapter 13. Volume Service (VOL)

Volume Service overview

The Encina Toolkit Volume Service (VOL) reads and writes data to secondary storage and provides an administrative interface to random-access secondary storage. It is built on the underlying operating system's synchronous (write-through rather than buffered) I/O interface. VOL administers secondary storage in units called volumes.

The Volume Service chapter is organized as follows:

- The remainder of this section explains basic concepts, Volume Service objectives, important abstractions, and the storage model used.
- "Data types and auxiliary functions" on page 220 describes the Volume Service data types.
- "Initialization" on page 214 specifies how to initialize the Volume Service.
- "Volume I/O" on page 223 describes VOL's page I/O functions.
- "Volume administration" on page 223 describes the administrative functions.
- "Volume Service interoperability" on page 224 describes functions needed for interoperating with other volume services, such as IBM's logical volume manager, AIX/LVM.
- "Diagnostics" on page 226 describes the diagnostic support provided by the Volume Service.

Volume Service

The Volume Service is the secondary storage access and administration facility for Toolkit applications, managing large quantities of disk space and data. The Volume Service provides interfaces for administering random-access secondary storage (disk) so that the physical boundaries of these devices are transparent to the client.

VOL is solely a storage administration and access facility. VOL relies on its clients to define disk allocation and data storage management and to take responsibility for crash recovery. As such, the Volume Service is intended to accomplish the following:

- **Administer Disk Space**: Provide a simple interface for the administration of large quantities of disk space, and the necessary maintenance and diagnostic facilities.
- **Store Data**: VOL reads and writes data to multiple physical devices.

- **Increase Data Availability**: In addition to VOL's administration interface, VOL provides client-invisible mirroring. Mirroring, where two or more copies of a volume are available online, provides increased availability and limits the negative effects of hardware failures on the availability of physically-accessible data.
- **Remove Physical Boundaries**: Storage areas from a number of disks can be made to look as one. This allows data to transparently span physical device boundaries, or allows physical devices to be transparently replaced by other physical devices of different type and capacity.
- **Tailor Physical Boundaries to Performance Goals**: VOL allows data to be placed within certain physical device boundaries to achieve desired performance characteristics, such as throughput and failure isolation.
- **Efficiently Use Disk Space**: Disk space can be managed in smaller units, allowing more efficient management.

The storage abilities provided by VOL are suitable for long-lasting, important data, and therefore VOL sometimes sacrifices speed for increased reliability. As a result, VOL may be inappropriate for some applications, such as virtual memory paging.

Implementations

The Volume Service supports two mutually-exclusive interfaces: the Volume Service on UNIX (VOL/UNIX) and the Volume Service interoperability interface.

VOL/UNIX is intended for clients running on UNIX systems that do not already have some type of logical volume service. It includes functions for creating logical volumes and maintaining them, as well as functions for doing I/O. This implementation of the Volume Service also supports mirroring.

The Volume Service interoperability interface is intended for systems that provide some type of logical volume abstraction, such as IBM's AIX Logical Volume Manager. This implementation of the interface does not support all the abstractions defined in the VOL/UNIX interface. It acts as a mapping layer to the vendor supplied volume service. It contains some functions for maintenance and supports most of the I/O routines. The Volume Service interoperability interface does not support mirroring or volume creation and destruction.

It is worth noting that in most cases the VOL/UNIX interface could be used even on UNIX systems that provide a logical volume abstraction. Under this circumstance volume mirroring would be handled by the VOL/UNIX code and not the underlying vendor supplied volume service.

Volume Service storage model

VOL administers physical disks using the following storage model abstractions. The storage abstractions are illustrated in Figure 48 on page 210, Figure 49 on page 210, and Figure 50 on page 211.

Storage abstractions

Disk: The Volume Service defines a disk as a byte-addressable chunk of random access secondary storage that, when accessed via a suitable I/O interface, provides synchronous I/O. VOL uses synchronous I/O instead of the file system buffer pool so that the data does not go through operating system buffering, since buffering would lead to inconsistencies in the presence of system crashes.

A disk is identified by a system-specific, printable character string called a *disk name*. Under most versions of UNIX, a disk is a disk partition. VOL/UNIX also supports pre-allocated UNIX files as disk abstractions. For such files, VOL/UNIX always sets the synchronous write bit to ensure the reliability of I/O.

The first page of each disk is reserved by the Volume Service to store access information. This page is the control block for the disk. Data types and functions that return page sizes return the number of usable pages, which does not include the disk control block.

Page: A page is the smallest addressable unit of space administered by VOL. The number of bytes in a page (the page size) is an implementation-specific constant (typically page size is defined to be 4096 bytes). The smallest unit of I/O that VOL uses is a page.

Chunk: A chunk is an allocation unit used by VOL for rounding down region sizes. A chunk is specified as a number of pages; the number must be a power of 2. For instance, the following are valid chunk sizes: 32, 64. The chunk size is often set to the number of VOL pages that are guaranteed to be physically contiguous on a disk. Using that chunk size ensures good performance for volumes that are used for sequential data access.

Region: A region is a collection of pages with contiguous addresses (one or more chunks) on a single disk. A region cannot be smaller than a chunk. A region size that is not a multiple of the chunk size will be rounded down to the nearest multiple of the chunk size. For example, if the chunk size is 32 pages, a region size of 72 pages will be rounded down to 64 pages. To make the most efficient use of disk space, it is recommended that the region size be set to a multiple of the chunk size.

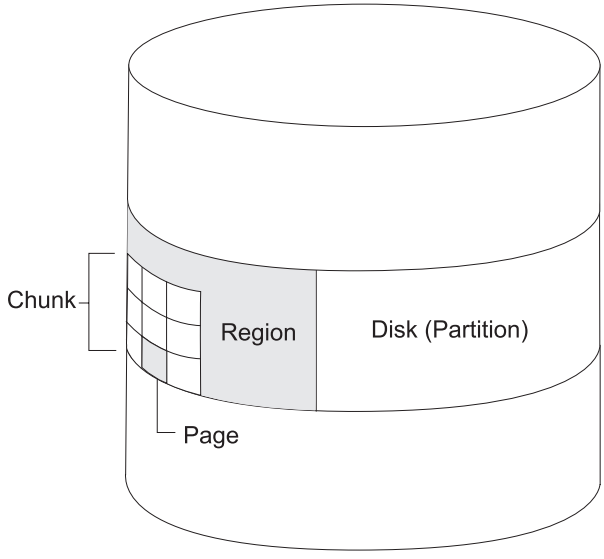

Figure 48. Physical storage abstractions

Physical volume: A physical volume is a collection of regions on one or more disks. Data is stored in physical volumes and administrative operations are performed on physical volumes. Physical volumes are referred to by a *physical volume identifier*, that is assigned when the volume is created.

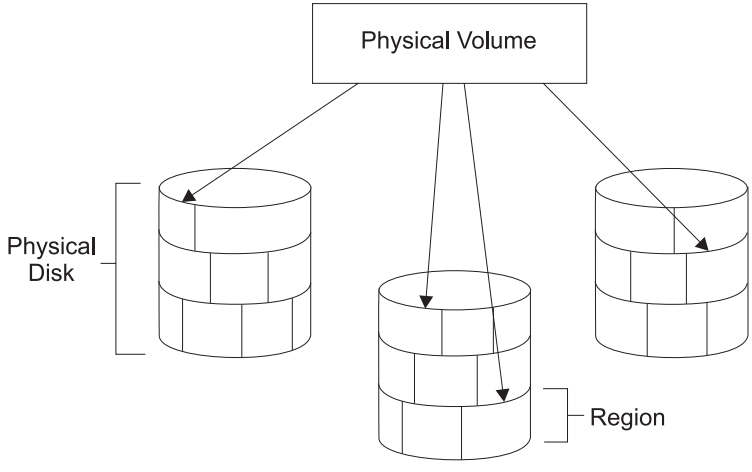

Figure 49. Physical volume

Logical volume: A logical volume presents the user with a contiguous address space of pages (0 .. N-1 pages in a volume with N pages) on secondary storage. Page I/O, mounting, and dismounting volumes is

performed by applications on logical volumes. A logical volume is mapped to one or more physical volumes. All user I/O occurs through a logical volume. Logical volumes are referred to by a *logical volume identifier*. Note that Encina's logical volumes should never be shared (for example, by a file system), or loss of data may occur.

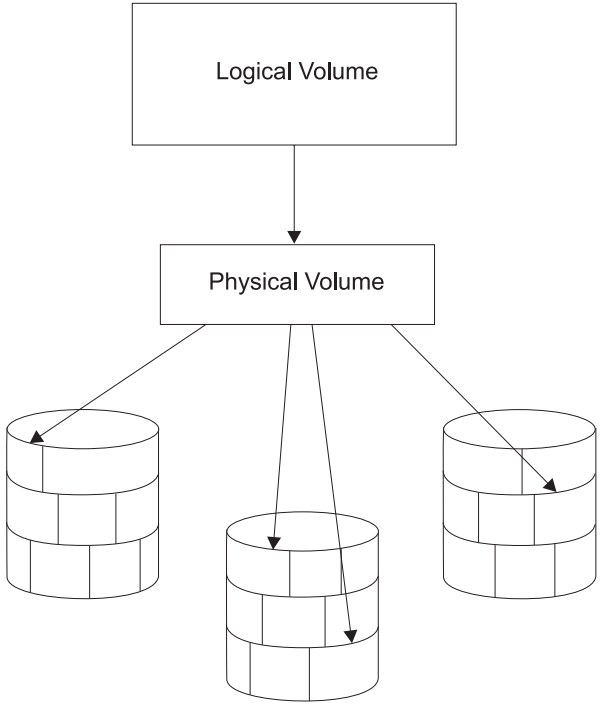

Figure 50. Logical volume

Storage abstraction states

The VOL storage abstractions (disks, physical volumes, logical volumes) are associated with various states; the state determines which operations can be performed on the storage abstraction. This section defines the states and movement between states for each storage abstraction.

Disk states:

Initialized: A disk is initialized when a client call to the **vol_InitializeDisk** function has succeeded.

Online: A disk is online when a physical volume has regions on the disk, and that physical volume is in the mounted state.

Physical volume states:

Online: A physical volume is considered online if all of the disks containing its constituent regions are online. An online volume may be mounted. Under some circumstances, a volume need not be fully online (have all of its regions available) to be mounted. See "Logical volume states" for further information.

Mounted: A physical volume is mounted when a call to the **vol_MountLogicalVol** function has succeeded against a logical volume that the physical volume backs. I/O requests made against the logical volume will result in I/O to the disk(s) holding the regions of the mounted physical volume.

Dismounted: A physical volume is dismounted when the logical volume is dismounted by a call to the **vol_DismountLogicalVol** function or when the system crashes.

Clean: A physical volume is clean if the data on it is valid. Only clean volumes can be mounted in normal mode. A clean volume automatically becomes dirty if it is mounted when the system is shut down.

Stale: A physical volume is stale if its contents reflect a version of the logical volume that is known to VOL to be earlier than the current version.

Since VOL does not examine every page of the physical volume, it is conservative about declaring physical volumes as stale; if a user mounts a logical volume while one of two physical volumes backing it is unavailable, but immediately dismounts it, VOL will declare the unavailable physical volume as stale.

Dirty: A physical volume is dirty if it was mounted while the system crashed. Since the client may have modified in-memory copies of some pages without writing them back to the volume, or the system may have crashed while VOL was in the process of writing a page, the client must be prepared to examine and certify the contents of the physical volume. Dirty volumes cannot be mounted in normal mode.

Corrupted: A physical volume is corrupted if its contents do not reflect the result of user I/O requests. A physical volume with a hard write error is marked as corrupted. A volume can also be marked as corrupted by calling the **vol_CorruptedLogicalVol** function.

The only operations permitted on corrupted physical volumes are repair and reconstruction. Corrupted volumes cannot be mounted. See "Logical volume states" for further information.

Indeterminate: This state is equivalent to corrupted. The indeterminate state will not be supported in future releases of Encina.

Logical volume states:

Mounted: A logical volume is mounted if a call to the **vol_MountLogicalVol** function has succeeded against it. Normally, all physical volumes backing a logical volume must be fully online for the

logical volume to mount successfully, but the client may request that VOL allow mounting of physical volumes that are not fully online via the `VOL_MOUNT_PARTIAL` flag.

Dismounted: A logical volume is dismounted if no call to the **vol_MountLogicalVol** function has succeeded against it, or if a call to the **vol_DismountLogicalVol** function has succeeded against it.

Corrupted: A logical volume is marked as corrupted when all of the physical volumes backing the logical volume have become corrupted. The only operations permitted on an corrupted logical volume are repair operations. A VOL client can mark a logical volume as corrupted by calling the **vol_CorruptedLogicalVol** function.

Indeterminate: This state is equivalent to corrupted. The indeterminate state will not be supported in future releases of Encina.

Volume Service usage model

In general, there are three stages of use for the Volume Service.

1. Initialize the Volume Service and make volumes available for normal operations.
2. Carry out normal operations. This includes both I/O and administrative functions.
3. Make volumes unavailable for normal operations and cease processing via an orderly shutdown.

Changes to VOL abstractions during these three stages are recorded in the restart data. The restart data and each of the stages noted above will be discussed in the following sections.

CAUTION:
The DCE threading model permits the cancellation of threads. The Volume Service assumes its work will not be interrupted by such a cancellation. Should a thread executing a VOL function be cancelled, the results are undefined.

Restart data

The Volume Service maintains *restart data* for the logical volumes, physical volumes, and disks that are in use by an application. VOL requires that the application save and restore this restart data. When a client application initializes VOL, it provides the most recent restart data and a function that VOL can call to store new versions of the restart data. When volumes or disks change state, VOL stores an updated copy of the restart data. VOL assumes that the client application will successfully save this restart data in one atomic action.

If the client cannot save the restart data, the application must crash, so that the previous logical and physical volumes can be reestablished. There are

many ways that the management of the restart data can be implemented. The most common approach is to save and restore the restart data in a log managed by a log service. See "Chapter 8. Log Service (LOG)" on page 119 for more information about the Encina Toolkit Log Service.

Initialization

There are two types of Volume Service initialization: cold start and warm start. Cold start implies that there is no preexisting restart data for the Volume Service, so no volumes exist. Usually this will only occur once when the client application is first run. Warm start implies that restart data for the Volume Service does exist, and therefore so do volumes. The steps necessary for each type of initialization are described below. If you are making use of the Volume Service in conjunction with the Encina Recovery Service, see Chapter "Chapter 10. Recovery Service (REC)" on page 143 for further information.

Cold start:

1. Initialize the Volume Service, passing a NULL pointer and zero for restart data and its length.
2. Create logical volumes. Under the VOL/UNIX interface this includes initializing disks via the **vol_InitializeDisk** function, creating physical volumes via the **vol_CreatePhysicalVol** function, and finally creating logical volumes via the **vol_CreateLogicalVol** function. Under the VOL interoperability interface, the client only needs to invoke the **vol_MapLogicalVol** function to create a logical volume.
3. Mount the logical volumes so that normal administrative and I/O operations can begin. This is done by invoking the **vol_MountLogicalVol** function, with the VOL_MOUNT_NO_FLAGS flags argument.

Warm start: The steps described here assume that the client application had an orderly shutdown. For instructions concerning a warm start after a nonorderly client shutdown (a crash), see "Crash restart operations" on page 219.

1. Initialize the Volume Service, passing in a pointer to the Volume Service restart data and its length.
2. Mount the logical volumes so that normal administrative and I/O operations can begin. This is done by invoking the **vol_MountLogicalVol** function, with `VOL_MOUNT_NO_FLAGS` as the flags argument.

Normal operations

Normal Volume Service operations consist of both I/O operations to logical volumes, as well as administrative operations. Normal I/O operations include functions such as **vol_ReadPages**, **vol_WritePages**, and **vol_FillVol**. Normal administrative operations cover a variety of topics, which are discussed below. If I/O errors occur while the Volume Service is attempting to complete these or other Volume Service functions, the error-advice upcall is invoked and the

operation is unconditionally aborted. The error-advice upcall is for informational purposes only. It must not invoke any administrative functions. For more information on the error-advice upcall, see the reference page for the **vol_Init** function.

VOL does not lock data to serialize I/O. I/O requests to the same page from different threads will occur in an arbitrary order. If serialized I/O must be guaranteed, it is the client's responsibility to provide additional support.

Many of the VOL query functions allocate memory to return information to the caller. Clients must not modify this information in any way before returning it to VOL via the **vol_Free** facility.

Mirroring a logical volume:

Note: This information applies only to the VOL/UNIX interface, since mirroring is not supported under the VOL interoperability interface.

Once a logical volume has been created, it can be mirrored (replicated) to provide additional data availability (see Figure 51 on page 216). Mirrors (versions) are physical volumes and are added to a logical volume by invoking the **vol_AddMirror** function. A logical volume can have any number of mirrors. A physical volume that is assigned to a logical volume is considered to be *mapped*. All physical volumes mapped to a given logical volume are said to *back* the logical volume. All physical volumes backing a logical volume must have the same chunk size.

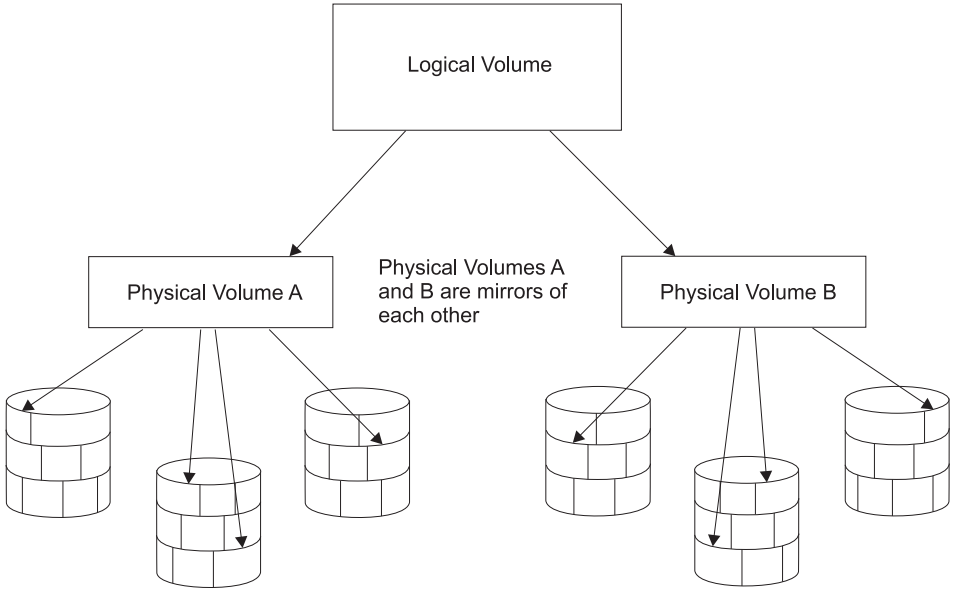

Figure 51. Mirrored volumes

If a mirror is added to a logical volume prior to any I/O being issued to that logical volume, then no further action is required beyond the call to the **vol_AddMirror** function. However, if I/O has occurred to the logical volume, then after the call to **vol_AddMirror** succeeds, the newly added physical volume is marked as stale (`VOL_STATE_STALE`). This means that particular physical volume is out of sync with respect to the other physical volumes backing the logical volume. To remedy this situation the client must invoke the **vol_SyncLogicalVol** function. This causes all physical volumes backing the logical volume to contain the same data and clears the stale state. The data is copied from a non-stale physical volume, to the stale physical volume(s). No I/O is permitted to stale physical volume(s) backing a logical volume.

Expanding volumes: Both logical and physical volumes can be expanded to increase the size of the volumes. The expanded data space is added to the end of the volume. Once a volume is expanded it cannot be shrunk.

Under the VOL/UNIX interface, logical volumes are expanded by first expanding each of the physical volumes backing the logical volume via the **vol_ExpandPhysicalVol** function. The additional data space created by the expansion of the physical volume(s) is then made usable by invoking the **vol_ExpandLogicalVol** function.

Under the VOL interoperability interface, the client only needs to expand the logical volume, since no notion of a physical volume exists in this interface.

Prior to invoking the **vol_ExpandLogicalVol** function to expand the logical volume, the client must have taken any necessary steps to expand the vendor volume to which the logical volume is mapped.

Corrupted volumes: Both logical and physical volumes can be marked as corrupted (`VOL_STATE_CORRUPTED`). Physical volumes are marked as such whenever a write error occurs on that physical volume. The Volume Service will not allow I/O to be issued against a corrupted physical volume. Should all physical volumes backing a logical volume become corrupted, then the logical volume is marked as corrupted and will be automatically unmounted. This ceases all I/O operations to that logical volume.

To repair a corrupted physical volume that was a mirror for a logical volume, it is suggested that the following steps be taken. (These actions are only valid under the VOL/UNIX interface since there is no physical volume abstraction or mirroring supported under the VOL interoperability interface.)

1. Remove the physical volume as a mirror from the logical volume by invoking the **vol_RemoveMirror** function (optional).
2. Relocate the region in which the write error occurred by invoking the **vol_RelocatePhysicalVol** function.
3. Change the state of the physical volume from corrupted to stale by calling the **vol_CertifyPhysicalVol** function.
4. Add the physical volume as a mirror to the logical volume by invoking the **vol_AddMirror** function (optional).
5. Invoke the **vol_SyncLogicalVol** function on the logical volume. This will update the data on the physical volume and clear the stale bit.

In order for VOL to repair a corrupted logical volume under the VOL/UNIX interface, an uncorrupted (either stale or clean) physical volume must be available to the logical volume. It is recommended that clients always have at least two good physical volumes backing a logical volume.

To repair a logical volume that has become corrupted, the client should take the following steps:

1. Mount the corrupted logical volume in restore mode by invoking the **vol_MountLogicalVol** function with the `VOL_MOUNT_RESTORE` flag set.
2. Perform media recovery (for example, copy data from a tape).
3. Invoke the **vol_CertifyLogicalVol** function to clear the corrupted bit on the logical volume. This will also change the mount status of the logical volume from restore mode to normal mode.

Under the VOL interoperability interface, since mirroring is not supported, any write error will cause the logical volume to be marked as corrupted. The Volume Service assumes the underlying vendor volume service will provide

the client with the means to repair the damaged volume. Once the vendor volume has been repaired, the client should follow the steps for repairing a corrupted logical volume.

Destroying volumes: Both physical and logical volumes can be destroyed. To destroy a physical volume (under the VOL/UNIX interface only) do the following:

1. If the physical volume is mapped to a logical volume unmap it by invoking the **vol_RemoveMirror** function. Note that if this physical volume is the only one mapped to the logical volume, you need to destroy the logical volume. See instructions below.
2. Invoke the **vol_DestroyPhysicalVol** function passing in the physical volume identifier.

Once a physical volume is destroyed, the Volume Service no longer retains any information about that physical volume or its identifier. Any attempt to use that identifier in Volume Service interface calls will result in the error `VOL_INVALID_PHYSICAL_ID`.

To destroy a logical volume take the following steps:

1. If the logical volume is mounted, then unmount it by invoking the **vol_DismountLogicalVol** function.
2. If you are using the VOL/UNIX interface and the logical volume is mirrored, you must first unmap all but one of the physical volumes backing the logical volume.
3. Destroy the logical volume by invoking the **vol_DestroyLogicalVol** function under VOL/UNIX or the **vol_UnmapLogicalVol** function under the VOL interoperability interface.

Destroying a logical volume means the associated logical volume identifier will be invalid for most Volume Service interface calls. The logical volume identifier itself is not destroyed and remains in the Volume Service's restart data; it can be reused by passing it to the **vol_RecreateLogicalVol** function under the VOL/UNIX interface or the **vol_RemapLogicalVol** function under the VOL interoperability interface. Thus, logical volume identifiers are eternal. It is also valid to pass a destroyed logical volume identifier to the **vol_GetLogicalVolInfo** function.

Orderly shutdown

The only requirement for an orderly shutdown of the Volume Service is that all logical volumes that are mounted be unmounted by invoking the **vol_DismountLogicalVol** function.

Crash restart operations

Should a Volume Service client application fail to execute an orderly shutdown due to a system or application failure, the Volume Service will require maintenance on those volumes that were mounted at the time of the failure. All such volumes will be marked as dirty (`VOL_STATE_DIRTY`) during Volume Service initialization, and no I/O will be permitted to such volumes. Two of the reasons why maintenance is required are, one, if the client application crashed during a Volume Service write operation, data could have gotten to one mirror of a logical volume but not to all of them, and two, since the Volume Service page size is larger than a disk sector, the write could complete to one sector of a page but not all. Also, clients such as the Encina Recovery Service may be logging updates to pages without actually issuing I/O's. To perform maintenance on dirty logical volumes do the following:

1. Mount the dirty logical volume(s) in maintenance mode by invoking the **vol_MountLogicalVol** function with the `VOL_MOUNT_MAINTENANCE` flag set.
2. For each page that the client believes may not contain the correct data, invoke the **vol_ReadPageVersions** function. If there is more than one version of the page, then select the correct version and invoke the **vol_ChoosePageVersion** function to make all versions of the page like the chosen one. This sequence of invocations to the **vol_ReadPageVersions** and **vol_ChoosePageVersion** functions should be executed for all pages the client believes has incorrect data. If only one version of the page exists for the logical volume (that is, it is not mirrored), then the client should mark the logical volume as corrupt by invoking the **vol_CorruptedLogicalVol** function. This invokes the error-advice upcall to indicate this volume is being marked as corrupt (`VOL_STATE_CORRUPTED`) and unmounts the logical volume.
3. After all pages are corrected, and there was no need to mark the logical volume as corrupt, then the client should invoke the **vol_CertifyLogicalVol** function. This clears the dirty state bit and changes the mount mode from maintenance to normal.

VOL header files and libraries

Header files

Applications that link with the Volume Service must include the header file **vol/vol.h** in their C program.

Libraries

The Encina Volume Service functions are in the **EncServer** library. See the *Writing Encina Applications* manual for further information on the libraries used during compilation.

Data types and auxiliary functions

This section describes the data types and related functions exported by the Volume Service (VOL) interface.

Status data type

Most VOL functions return a status code of type **vol_status_t**. Any function that is successful returns `VOL_SUCCESS`. Status codes can be converted into a string using the **encina_StatusToString** function. See the reference page for the **vol_status_t** data type for a complete list of VOL status codes.

Action on error

The **vol_errorAction_t** data type is used by the Volume Service to indicate how to respond to a volume error.

Disk space

The following VOL data types are used to represent information about a region on a disk:

- **vol_physLayout_t**
- **vol_region_t**
- **vol_regionDescr_t**
- **vol_regionUse_t**

VOL also exports the `VOL_PAGE_SIZE` constant, which defines the system-wide page size.

Volume identifiers and related functions

The following VOL data types are used to represent logical and physical volume identifiers:

- **vol_logicalVolId_t**
- **vol_physicalVolId_t**

The following VOL functions are used to compare volume identifiers:

- **vol_LogicalVolIdCmp**
- **vol_PhysicalVolIdCmp**

Names and labels

A client of VOL may find it useful to associate a small amount of naming information with a logical or physical volume. For example, a logical volume could be designated a file system's root volume by naming it "root", or a physical volume could be named "sales mirror 1" to remind administrators of its purpose.

VOL will map from logical and physical volume names to logical and physical volume identifiers with the **vol_GetPhysicalByName** and **vol_GetLogicalByName** functions.

VOL clients can associate a small amount of distinguishing information with disks managed by VOL (through the use of disk labels). A client could use this to ensure that the contents of the volumes on that disk are compatible with the current software version.

Shown below are the C data structures used by VOL to track textual information about the volumes under its control. All string lengths include the NULL terminator.

```
char physicalVolName[VOL_PHYSICAL_NAME_SIZE];
char logicalVolName[VOL_LOGICAL_NAME_SIZE];
char diskLabel[VOL_DISK_LABEL_SIZE];
char diskName[VOL_DISK_NAME_SIZE];
```

A disk name is a null-terminated byte string representing the OS-specific name of a disk (such as **/dev/rhp0a**). VOL assumes that disk names are printable and accessible via a file system "open" call.

CAUTION:
The use of links for disk names and vendor volume names is discouraged since the Volume Service does not traverse them. A link and the file/device it points to could be used separately as

- The *name* argument for the **vol_InitializeDisk** function
- The *logicalVolName* argument for the **vol_MapLogicalVol** and **vol_RemapLogicalVol** functions

However, this would create multiple means of access to the same physical storage area. This will surely result in the loss of data.

Query result types

VOL defines the following constants describing storage abstraction states:

- `VOL_STATE_PARTIAL` — volume is partially but not fully online.
- `VOL_STATE_ONLINE` — volume is completely online.
- `VOL_STATE_MOUNTED` — physical volume is backing a mounted logical volume.
- `VOL_STATE_MIRRORED` — physical volume is not the only one backing the indicated logical volume.
- `VOL_STATE_MAPPED` — physical volume is mapped to a logical volume.
- `VOL_STATE_INDETERMINATE` — volume contents are indeterminate.
- `VOL_STATE_CORRUPTED` — volume contents are corrupted.
- `VOL_STATE_STALE` — physical volume contents are out of date.
- `VOL_STATE_CLEAN` — physical volume contents are up to date.
- `VOL_STATE_DIRTY` — physical volume needs some repair. If a storage abstraction is not in the "dirty" state, it is "clean".

- VOL_STATE_DESTROYED — logical volume has been destroyed and no longer has any physical volumes backing it.
- VOL_STATE_ANY — a specific state is not defined, and all information is returned.

Note: The VOL_STATE_INDETERMINATE state will not be supported in future releases of Encina. It is equivalent to the VOL_STATE_CORRUPTED state, which should be used instead.

Possible states are set in the *flags* field of the **vol_physicalVolInfo_t** and **vol_logicalVolInfo_t** data types. States are represented by a bit-vector. The VOL_STATE constants define the bits that correspond to each state. VOL provides functions, such as **vol_GetLogicalVolList**, that return disks or volumes in a specified state(s). If a match on more than one state is required, the required states can be combined with a bitwise **OR**.

Not all states are valid for all storage abstractions. Valid states for VOL storage abstractions are presented in Table 8.

Table 8. Valid volume and disk flags describing states

Storage state flags	Disk	Physical volume	Logical volume
VOL_STATE_CORRUPTED		X	X
VOL_STATE_DESTROYED			X
VOL_STATE_DIRTY		X	
VOL_STATE_INDETERMINATE		X	
VOL_STATE_MAPPED		X	
VOL_STATE_MIRRORED			X
VOL_STATE_MOUNTED		X	X
VOL_STATE_MOUNTED_MAINT			X
VOL_STATE_ONLINE	X	X	
VOL_STATE_PARTIAL		X	
VOL_STATE_STALE		X	

Logical volume locks

The following VOL data types are used in defining locks for logical volumes:

- **vol_logicalVolLock_t**
- **vol_logicalVolLockInfo_t**
- **vol_logicalVolLockType_t**
- **vol_pageList_t**

Initializing the Volume Service

The Volume Service must be initialized before calling any other VOL function. At initialization, the error-advice upcall and the save-restart-data upcall are set to handle disk error notification and saving and updating restart data. VOL supplies the **vol_Init** function for initialization.

Volume I/O

VOL provides the following page I/O functions:

- **vol_ReadPages**
- **vol_WritePages**
- **vol_ReadPageVersions**
- **vol_ChoosePageVersion**
- **vol_FillVol**
- **vol_ValidatePage**

Volume administration

Administrative interfaces for the VOL module

Volume Service (VOL) provides the following functions for VOL administration:

Disk Operations

vol_InitializeDisk

vol_VerifyDisk

vol_RelabelDisk

vol_DestroyDisk

Creating and Destroying Volumes

vol_CreatePhysicalVol

vol_CreateLogicalVol

vol_RecreateLogicalVol

vol_DestroyPhysicalVol

vol_DestroyLogicalVol

Renaming Volumes

vol_RenameLogicalVol

vol_RenamePhysicalVol

Manipulating Volumes

vol_ExpandPhysicalVol

vol_ExpandLogicalVol

vol_RelocatePhysicalVol

Mounting and Dismounting Volumes

vol_MountLogicalVol

vol_DismountLogicalVol

vol_CertifyLogicalVol

vol_CertifyPhysicalVol

vol_CorruptedLogicalVol

Mirroring Volumes

vol_AddMirror

vol_RemoveMirror

vol_SyncLogicalVol

Querying Disks and Volumes

vol_GetDiskInfo

vol_GetPhysicalVolInfo

vol_GetLogicalVolInfo

vol_GetDiskList

vol_GetPhysicalVolList

vol_GetLogicalVolList

vol_GetPhysicalByName

vol_GetLogicalByName

vol_Free

Locking Logical Volumes

vol_LockLogicalVol

vol_UnlockLogicalVol

vol_GetLogicalVolLockInfo

vol_GetLogicalVolLockList

Volume Service interoperability

VOL provides the following interoperability functions. When using VOL without the interoperability interface, these interoperability functions return `VOL_NOT_IMPLEMENTED`.

- **vol_MapLogicalVol**
- **vol_UnmapLogicalVol**
- **vol_RemapLogicalVol**

In addition to the interoperability functions, the following VOL functions can be used with other vendor-supplied volume services.

- **vol_CertifyLogicalVol**
- **vol_CorruptedLogicalVol**

- **vol_DismountLogicalVol**
- **vol_DumpState**
- **vol_ExpandLogicalVol**
- **vol_FillVol**
- **vol_Free**
- **vol_GetLogicalByName**
- **vol_GetLogicalVolInfo**
- **vol_GetLogicalVolList**
- **vol_Init**
- **vol_LogicalVolIdCmp**
- **vol_MountLogicalVol**
- **vol_ReadPages**
- **vol_ReadPageVersions**
- **vol_RenameLogicalVol**
- **vol_ValidatePage**
- **vol_WritePages**

When compiled for the VOL interoperability interface, the following functions are not available and will return the `VOL_NOT_IMPLEMENTED` status code:

- **vol_AddMirror**
- **vol_ChoosePageVersion**
- **vol_CreatePhysicalVol**
- **vol_CreateLogicalVol**
- **vol_DestroyDisk**
- **vol_DestroyLogicalVol**
- **vol_DestroyPhysicalVol**
- **vol_ExpandPhysicalVol**
- **vol_GetDiskInfo**
- **vol_GetDiskList**
- **vol_GetPhysicalByName**
- **vol_GetPhysicalVolInfo**
- **vol_GetPhysicalVolList**
- **vol_InitializeDisk**
- **vol_PhysicalVolIdCmp**
- **vol_RecreateLogicalVol**
- **vol_RelabelDisk**
- **vol_RelocatePhysicalVol**

- **vol_RemoveMirror**
- **vol_RenamePhysicalVol**
- **vol_SyncLogicalVol**
- **vol_VerifyDisk**

Diagnostics

This section documents the diagnostic support provided by the Volume Service. This includes event tracing, fatal messages, warning messages, and snapshots of the Volume Service state.

Directing output

All trace, dump, warning, and fatal error output may be directed to user-specified files or a ring-buffer, by use of the relevant Encina Toolkit Trace Facility functions. By default this output will be sent to the ring buffer. The ring buffer may be dumped by calling the **trace_DumpRingBuffer** function.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or the standard error or standard output stream. See "Appendix A. Tracing and debugging Encina Toolkit applications" on page 231 for more details on the Encina Trace Facility.

Fatal error messages

The Volume Service generates a fatal runtime error whenever an unrecoverable event occurs. The run-time error is presented as an output message through the Encina Trace Facility (see "Trace and state dump information" on page 227) and the application's execution terminated. The destination of the error message can be defined by using the functions provided by the Encina Trace Facility (see "Directing output"). This section defines the fatal error messages that may be output by the Volume Service, and briefly describes why they occur, and how they may be avoided or remedial action taken.

- ″Release version numbers in VOL restart data are invalid.″—The release and version numbers stored in VOL's restart data are invalid. If an earlier version of the restart data is available try it, or possibly restore a version from backup media.
- ″The VOL restart data is corrupted.″—The restart data passed to VOL is corrupted, i.e. unreadable by VOL. If an earlier version of the restart data is available try it, or possibly restore a version from backup media.

Warning messages

The Volume Service returns the following warning message.

- ″volPhysical_Expand: Failed to bring disk, *disk_name*, online.″ and ″volPhysical_Relocate: Failed to bring disk, *disk_name*, online.″—An attempt

to "open" the disk, *disk_name*, for I/O operations failed. Validate that the disk, *disk_name*, exists and has read/write permissions for the client who is attempting to access it via VOL.

Audit messages

The Volume Service does not produce any audit messages.

Trace and state dump information

Tracing

The Volume Service exports the *vol_traceMask* variable to provide clients with the means of turning on or off specific tracing within the Volume Service. The variable is defined as follows:

```
unsigned long vol_traceMask
```

The following event trace bits are defined by the Volume Service. They may be OR'ed in any combination.

Note: Currently the Volume Service does not support any internal entry/exit or parameter tracing.

- VOL_TRACE_ADMIN — is the equivalent of OR'ing the following trace bits:
  - VOL_TRACE_CREATE_DESTROY
  - VOL_TRACE_EXPAND
  - VOL_TRACE_MAP_UNMAP
  - VOL_TRACE_MOUNT_UNMOUNT
- VOL_TRACE_ALL_DISK — traces all events that occur in conjunction with disks.
- VOL_TRACE_ALL_IO — traces all events that occur in conjunction with I/O issued via the Volume Service.
- VOL_TRACE_ALL_LOGICAL — traces all events that occur in conjunction with logical volumes.
- VOL_TRACE_ALL_PHYSICAL — traces all events that occur in conjunction with physical volumes.
- VOL_TRACE_CREATE_DESTROY — traces the creation and destruction of exported objects such as logical and physical volumes.
- VOL_TRACE_EXPAND — traces the expansion of physical and logical volumes.
- VOL_TRACE_LOCKS — traces events associated with the setting and releasing of locks on logical volumes.
- VOL_TRACE_MAP_UNMAP — traces the mapping and unmapping of logical volumes via the **vol_MapLogicalVol** and **vol_UnmapLogicalVol** functions.
- VOL_TRACE_MOUNT_UNMOUNT — traces the mounting and unmounting of logical and physical volumes.
- VOL_TRACE_RESTART — traces events in the restart code.

- VOL_TRACE_SIG_DISK — traces only the significant events that occur in conjunction with disks.
- VOL_TRACE_SIG_IO — traces only the significant events that occur in conjunction with I/O issued via the Volume Service.
- VOL_TRACE_SIG_PHYSICAL — traces only the significant events that occur in conjunction with physical volumes.

State dump

The Volume Service provides the **vol_DumpState** function to dump the state of the Volume Service at an application.

Diagnostic RPC interface functions for VOL

The VOL interface includes RPC interfaces for use in diagnostic and service tools. These functions are not intended for use in applications.

The functions in this section are defined in the IDL (Interface Definition Language) file **diag_vol.idl**.

The diagnostic interfaces exported by the Toolkit Volume Service (VOL) are the following:

- **diag_vol_PageSize** This diagnostic RPC function returns the value of the VOL_PAGE_SIZE constant for a specified logical volume.
- **diag_vol_ReadPages**
- **diag_vol_WritePages**
- **diag_vol_ReadPageVersions** The diagnostic RPC interface to this function does not define the *statusesPP* and *buffersPP* parameters returned in the standard VOL function.
- **diag_vol_ChoosePageVersion**
- **diag_vol_CorruptedLogicalVol**
- **diag_vol_ValidatePage**

Part 4. Appendixes

Appendix A. Tracing and debugging Encina Toolkit applications

This appendix describes the tracing facilities provided by the modules of the Encina Toolkit. These facilities enable users and developers to monitor and observe the execution of those modules when called within an application. Sections of this appendix introduce the Encina Toolkit Trace facility, discuss its organization into multiple levels of tracing control, and discuss how to configure an application to use these facilities. Subsequent sections explain how to enable and disable Toolkit tracing, discuss how to selectively enable or disable tracing within specific modules of the Encina Toolkit, explain the different levels of tracing provided by specific Encina Toolkit modules, discuss how to activate or deactivate specific levels of tracing, and explain how to direct tracing output to specific locations, depending on the application that is producing the output.

For more information on tracing in Encina, see *Encina Administration Guide Volume 1: Basic Administration*.

General information about tracing

Tracing allows a developer or system administrator to obtain information about the execution path of an application, the parameters with which functions and procedures are called, and tracks significant, developer-defined events in the execution of an application. This type of information can be very useful when debugging user-level applications or when monitoring system performance and module interaction. The ability to obtain tracing output from tools that provide an enabling technology, such as the modules of the Encina Toolkit, can also provide substantial insights into how the software is being used, and therefore how it can be tuned to provide increased performance.

Most tracing facilities, such as those provided by the Encina Toolkit, can be enabled or disabled within an application. In the case of server programs, tracing is generally compiled in, but tracing output is not produced unless tracing is enabled (activated) in the server program. The most common types of tracing information gathered from applications or underlying services are the following:

- Tracking the values of the parameters passed to functions or procedures within a component or product. This is generally known as *parameter* tracing.
- Tracking the calling sequence of functions or procedures within a component or product. This is frequently known as *entry-exit* tracing.

- Tracking the occurrence of specific events within a product or component. This is generally known as *event* tracing.

Each of these types of tracing produces a different amount of output. For example, tracing both the calling sequence of internal functions and changes in the parameters passed to them can produce a great deal of output. This type of output may only be useful in certain circumstances, such as when attempting to resolve certain types of problems or when using tracing to obtain performance statistics. For this reason, the tracing facilities provided by Encina support different levels of tracing, which means that you can selectively identify the type of tracing output that you want to obtain.

In general, tracing should only be enabled when absolutely necessary. Tracing is best suited to detecting logical problems in the design or implementation of a system. Generating trace output is often counterproductive when attempting to diagnose timing problems, because the resources required to generate and write tracing information may obscure the timing problem. The best argument for only enabling tracing when absolutely necessary is the sheer volume of output which it generates. In general, tracing should also only be enabled selectively—if you suspect a problem in a specific module, you should only enable tracing for that module of component, expanding the tracing scope only as you discover that it is necessary to do so.

Overview of tracing in the Encina Toolkit

Encina software consists of products and components, where a product is a collection of components. Encina products also generally make use of supporting services provided by Encina components. For example, the Encina Structured File Server is a product that uses the transactional support services provided by the Encina Toolkit. The Encina Toolkit itself is composed of two products, the Encina Toolkit Executive and the Encina Toolkit Server Core, while each of these products is composed of a number of different components, such as the Encina Toolkit Recovery Service and the Encina Toolkit Transaction Service.

Because of the open, modular nature of Encina, its tracing facilities are similarly modular. Tracing can be controlled and separately enabled in both Encina products and Encina components. These are referred to as *standard* and *component* tracing. Standard tracing activates tracing information throughout all Encina products. Component tracing activates tracing information that has special significance to each component, and only affects tracing output generated by calls to that component.

Each Encina product and component defines a *trace mask*, which is a 32-bit global variable that controls the type of tracing that is currently active for that

product or component. This variable is a logical OR of all of the types of tracing that are currently active for the component. A trace mask is of the following form:

```
extern unsigned long component-name
```

For example, the trace mask for the Encina Toolkit Executive Log Service component would be named *log_traceMask*, while the trace mask for the Encina Toolkit Server Core Lock Service would be named *lock_traceMask*.

Encina products do not have explicit trace masks. Tracing within Encina products must be enabled or disabled within an application or by using an administrative tool such as the Encina Toolkit administrative tool, **tkadmin**. The **tkadmin** tool has access to internal information about the mappings between Encina products and their associated components. When the **tkadmin** tool is used to enable tracing for an Encina product, it modifies the trace masks, at the specified levels, for all of the components that make up that product. For information on using the Encina Toolkit administrative tools see the *Encina Administration Guide Volume 1: Basic Administration*.

The Encina tracing facilities support generation of the following types of information during the execution of a product or component:

- For exported functions, those documented and available to users of a component or product:
  - The calling sequence of exported functions within the product or component. This is activated by setting the `TRACE_ENTRY` bit of the component's trace mask to the value 1 (a logical OR of `TRACE_ENTRY` and the current value of the trace mask).
  - The values of the parameters with which exported functions in that product or component are called. This is activated by setting the `TRACE_PARAM` bit of the component's trace mask to 1 (a logical OR of `TRACE_PARAM` and the current value of the trace mask.
- For internal functions, those used internally by a component or product that are unavailable to users:
  - The calling sequence of internal functions within the product or component. This is activated by setting the `TRACE_INTERNAL_ENTRY` bit of the component's trace mask to 1 (a logical OR of `TRACE_INTERNAL_ENTRY` and the component's trace mask).
  - The values of the parameters with which internal functions in that product or component are called. This is done by setting the `TRACE_INTERNAL_PARAM` bit of the component's trace mask to 1 (a logical OR of `TRACE_INTERNAL_PARAM` and the component's trace mask).
- The occurrence of specific events within a product or component. This is activated by setting the `TRACE_EVENT` bit of the component's trace mask to 1 (a logical OR of `TRACE_EVENT` and the component's trace mask).

To simplify enabling tracing when attempting to resolve a problem where performance during debugging is not an issue, Encina also provides two mask values which simultaneously either enable all standard tracing (TRACE_GLOBAL) or disable all tracing (TRACE_NONE).

Requirements for using Encina tracing

This section describes the additions to your application code and working environment that are required to use the Encina Tracing facility, and to display tracing messages in the correct language.

Include files for compiling applications with tracing

In order to compile applications that provide tracing output, the source code for your application must specify the include file that defines constants and data structures used by the Encina tracing facility. This is the file **trace.h**, which can be specified in your application source code by using a line like the following:

```
#include <utils/trace.h>
```

Environment variables for generating trace output

During the execution of Encina Toolkit component routines within an application, the Encina tracing mechanism uses a component-specific trace database to identify and interpret the events for which tracing has been activated. Each Encina Toolkit component's trace database has a name of the form **trace**_*componentName*.**tpp**. These database are located by the Encina tracing run-time environment by checking the directories listed in the NLSPATH environment variable. This environment variable must be set to enable National Language Support (NLS), and defines a search path (a collection of directory names separated by colons) that will are searched in order by the NLS support system to locate NLS message catalogs. For tracing purposes, the NLSPATH variable usually contains an entry of the form *installationDirectory***/msg/%L/%N**, where *installationDirectory* is the name of the directory in which Encina has been installed on your system, **%L** represents the language in which you want those messages to be displayed, and **%N** is the name of the file containing the messages.

The language used for NLS-aware messages from the Encina trace facility is **C**, because this is the programming language in which the Encina Toolkit is written. The name of the catalog of messages used by the tracing facility for a specific component is formed from this abbreviation. For example, the abbreviation for the Toolkit Distributed Transaction Service is **tran**, which means that the name of the message catalog containing Toolkit Distributed Transaction Service trace messages for use by applications running in the United States is *installationDirectory***/msg/C/trace_tran.cat**.

If the trace database cannot be found for a component in which tracing has been enabled, a warning message will be displayed for each trace statement whenever tracing is enabled. If you receive such messages, check to make sure that this variable has been set, and correctly includes the name of the directory in which the Encina Toolkit trace message catalogs have been installed.

Enabling tracing in Encina applications

Once you know a component's trace mask and the level of tracing that you want to activate, tracing can be enabled by setting the bit corresponding to the required level in the trace mask. This can be done in several different ways. This section discusses the mechanisms that application developers can use to enable tracing while they are working on or testing an application. Tracing for Encina components or products can also be enabled by a system administrator by using Enconsole or administrative commands, such as **tkadmin**. Using the **tkadmin** command is described in the *Encina Administration Guide Volume 1: Basic Administration*.

Activating component-level tracing in Encina

The level of tracing within a component can be controlled in one of the following ways:

- **From within a program**: This option is typically used during the development of an application to verify its execution path. The trace mask can either be explicitly set, or you can implement an application-specific mechanism for getting trace levels from the environment. For example, if you needed to do a great deal of tracing at multiple levels, without having to recompile your application, you might consider using an environment variable to hold a numeric value that your application reads and converts to a trace mask value.
- **By using a debugger to set trace masks while an application is running**: This can either be done by starting the application inside a debugger, or by attaching to a running process using the debugger. These options are typically used when tracking down problems in an application that is under development but is undergoing testing.
- **By using the Encina Toolkit administration tool, tkadmin**: This option is typically used to trace problems or obtain execution information in a server written with the Encina Toolkit. For performance reasons, enable tracing in a server only when it is under development or when you are trying to track down a problem and need to collect execution information.
- **By using the administrative RPC interfaces provided by Encina Toolkit components**: This option is typically used if you are writing your own administrative tool, or if you write a server application that you want to be able to contact by using an interface of your own design.

The mechanism that you use to enable tracing depends largely on the type of application you are developing and the stage of the development of your application. For example, the first and second options are typically used during the development of an application. The third and fourth options are used when developing and testing server applications that use the Toolkit services. The first and fourth options require the application (or client application, for option four) to be recompiled if you need to change the tracing level or types of tracing events. For debugging or analyzing server applications in a production environment, the third option (using **tkadmin**, the Encina Toolkit Administration tool) is preferable, since this both provides the most flexible solution and eliminates the need to recompile your server.

The next few sections provide examples of using each of these mechanisms to enable entry/exit and parameter tracing for the Toolkit Lock Service (LOCK) module.

Enabling component tracing within an application

When a program enables tracing, it can set and reset trace bits explicitly. For example, required trace bits can be added to the *lock_traceMask* variable with an **OR** operation:

```
lock_traceMask |= TRACE_ENTRY | TRACE_PARAM;
```

To disable this tracing, the program uses an **AND** operation with the bitwise negation of the trace bits:

```
lock_traceMask &= ~(TRACE_ENTRY | TRACE_PARAM)
```

Enabling component tracing by using a debugger

Tracing can be enabled using a debugger in two different ways: either by attaching to a running process, and setting the desired trace mask while the process is active, or by starting the application within a debugger, and enabling the trace masks when certain breakpoints are reached.

For example, when running an application within a debugger, tracing can be enabled by assigning the required trace bits to the appropriate trace mask variable. Typically, you would do this by setting a breakpoint in the main routine of the application, assigning the appropriate trace mask value to the internal location where the that variable is stored, disabling the breakpoint, and then continuing the execution of the application. The numeric value assigned to the location of the trace mask variable would be the numeric **OR** of the different types of tracing that you want to enable.

In the following two examples, the value 0x06 is the logical **OR** of the values of the `TRACE_ENTRY` and `TRACE_PARAM` trace bits.

For example, the commands necessary to use the traditional UNIX debugger, **dbx**, to do this would look something like the following:

```
(dbx) stop in main
(dbx) run ...
(stopped in main)
 ...
(dbx) assign lock_traceMask = 0x06
(dbx) continue
```

The commands to do the same thing are present in most debuggers. For example, the commands to set these values using the GNU debugger, **gcc**, would look something like the following:

```
gdb> break main
gdb> run ...
 ...
gdb> set lock_traceMask = 0x06
gdb> continue
```

Enabling component tracing by using the Encina administration tool tkadmin

The Encina Toolkit administration tool **tkadmin** simplifies enabling tracing masks in running servers. To specify that entry/exit and parameter tracing be enabled in the LOCK module, a system administrator (or someone with administrative privileges for the Encina Toolkit modules) can execute a command like the following:

```
% tkadmin trace component lock entry param
```

The **entry** and **param** arguments are the names used by the **tkadmin** command to identify entry and parameter tracing.

Alternately, the **tkadmin trace specification** command uses a different syntax that enables both product and component tracing to be specified at the same time. For more information about using the **tkadmin** commands, see the *Encina Administration Guide Volume 1: Basic Administration*.

Enabling component tracing through administrative RPC interfaces

The Encina Toolkit provides direct access to all of the administrative interface functions used to implement the **tkadmin** tool. For example, a client application can directly issue the following RPC to a server that uses the LOCK module:

```
admin_trace_SetComponentMask(rpcHandle, "lock",
          TRACE_ENTRY | TRACE_PARAM);
```

Making this call would require that *rpcHandle* be a DCE-RPC binding handle for the Lock Service application.

Specific information about the administrative RPC interfaces exported by the various modules of the Encina Toolkit, and about how to use those interfaces, is provided with the documentation of each module.

Enabling product-level tracing in Encina

The Encina Toolkit does not provide any productwide trace masks. There are still several different mechanisms for enabling tracing within an entire Encina product. The next few sections show three different ways to enable entry/exit and parameter tracing throughout the modules of the Encina Toolkit Executive.

Enabling product tracing in an application

In an application, productwide tracing can be enabled by using the **TRACE_PRODUCT** macro to specify the trace mask for the product. The parameters to this macro are the name of the product, passed a string value, and the trace mask values that you want to enable for that product. An example of using this construct to specify entry/exit and parameter tracing in the Encina Toolkit Executive product:

```
TRACE_PRODUCT("Encina Executive", TRACE_ENTRY | TRACE_PARAM);
```

Enabling product-level tracing by using the Encina administration tool tkadmin

The Encina Toolkit administration tool **tkadmin** can enable or disable tracing masks for a product in running servers. To specify that entry/exit and parameter tracing be enabled throughout the Encina Toolkit Executive, a system administrator (or someone with administrative privileges for the Encina Toolkit) can execute a command like the following:

```
% tkadmin trace product "Encina Executive" entry param
```

The **entry** and **param** arguments are the names used by the **tkadmin** application to identify entry and parameter tracing.

Alternately, the **tkadmin trace specification** command uses a different syntax that allows both product and component tracing to be specified at the same time. For more information about using the **tkadmin** commands, see the *Encina Administration Guide Volume 1: Basic Administration*.

Enabling product tracing through RPC interfaces

The Encina Toolkit provides direct access to all of the administrative interface functions used to implement the **tkadmin** tool. For example, a client application can directly issue the following RPC to a server that uses the services provided by the Encina Toolkit Executive:

```
admin_trace_SetProductMask(rpcHandle, "Encina Executive",
            TRACE_ENTRY | TRACE_PARAM);
```

Making this call requires that *rpcHandle* be a DCE-RPC binding handle for the Lock Service application.

Specific information about the administrative RPC interfaces exported by the various modules of the Encina Toolkit, and about how to use those interfaces, is provided with the documentation of each module.

Collecting and interpreting Encina tracing output

The utility of a tracing facility depends upon the granularity, volume, and clarity of the information it displays about the program you are trying to trace. The ability to specify the location to which trace output is being written is very important, so that this information can be captured to files or removable media and subsequently analyzed. This is especially important when large amounts of tracing output is being generated (such as when tracing has been enabled in frequently accessed modules such as the Encina Toolkit TRPC or Transaction Services).

This section describes the format of the output produced by the Encina tracing facility, and also describes how that output can be obtained and directed into specific output files for subsequent examination.

Summary of traceable events

Each level of Encina tracing is a member of an associated trace class. The trace facility provides functions to unregister an existing destination for trace output by class, and to register a new destination. These registration and unregistration functions use the trace output classes defined in the tracing facility header file, *install-directory*/**include/utils/trace.h**. This file defines constant names used to refer to the possible events with which tracing can be associated. These classes are shown in Figure 52.

```
TRACE_CLASS_ENTRY   /* internal/external entry/exit */
TRACE_CLASS_EVENT   /* events */
TRACE_CLASS_PARAM   /* internal/external parameters */
TRACE_CLASS_FATAL   /* fatal error messages */
TRACE_CLASS_WARNING /* warning messages */
TRACE_CLASS_AUDIT   /* audit messages */
TRACE_CLASS_DUMP    /* state dump output */
```

Figure 52. Classes of traceable events

Using these classes, trace output can be redirected in one of three ways: from within a program, by using the Encina Toolkit Administration tool (**tkadmin**), or from within a remote application by using the administrative RPC interfaces directly. Each of these methods is discussed in this appendix.

Formatting Encina trace output

The Encina Toolkit Trace Facility produces formatted trace output that can be used for error reporting and problem diagnosis. Figure 53 on page 240 shows

sample trace output for event, entry/exit, parameter, and state dump messages.

```
 6 22644 94/05/26-19:40:18.557693 50400415 E sfs: Hello from SERVER
 1 01324 94/08/18-00:27:18.905458 28040c00 > trpc_UseWkEndpoints
 1 01324 94/08/18-00:27:18.905520 28040c23 P bindingVectorP:\
 360d30; count 0.
 1 01324 94/08/18-00:27:18.905840 28040c01 <R trpc_UseWkEndpoints\
 trpc.c trpc -> 00000000
 1 01324 94/08/18-00:27:18.905441 28043c02 < trpcReady trpc.c trpc

22 09515 94/08/19-12:35:10.987124 10043819 D threadTid state dump\
 for thread id: 22
```

Figure 53. Sample Encina tracing output

Each line of tracing output is a trace statement. The different fields in each trace statement using the default format are the following:

- **Thread identifier**: This field contains the thread identifier for the thread within the application process that generated the trace output. The thread identifier is displayed in decimal format and is at least four digits (6, 1, and 22 in the sample output in Figure 53).
- **Process identifier**: This field contains the system identifier for the process that generated the trace output.
- **Timestamp**: This field contains the date and time that the trace statement occurred. The timestamp is displayed in the form *YY*/*MM*/*DD*-*hh*:*mm*:*ss*.*uuuuuu*, where *YY* is the last two digits of the year, *MM* is the month, *DD* is the day, *hh* is the hour (in 24-hour format), *mm* is the minutes, *ss* is the seconds, and *uuuuuu* is the number of microseconds. By default, the time is given in local time.
- **Trace identifier**: This field contains the trace identifier for the trace message. The trace identifier is displayed as an eight-digit number in hexadecimal format. Trace identifiers are unique across Encina; they are provided for customer support purposes or for use when error message catalogs are not available.
- **Trace class code**: This field contains a one- or two-character code indicating the trace class (or subclass) for the trace statement. The trace class code is displayed as one of the following:

  > Indicates the entry to a function.

  < Indicates the exit from a function.

  <R Indicates the exit from a function with a return value.

  A Indicates an audit message.

  D Indicates a state dump message.

  E Indicates an event.

F Indicates a fatal error.

P Indicates a parameter trace.

W Indicates a warning (error) message.

- **Message**: This field contains a message string that describes the event that generated the message and includes any event-specific data. If the message is a fatal, warning (error), or audit message, the field contains an NLS message string.

  When a function is entered (**>**), the name of that function is displayed. If parameter tracing is enabled (**P**), subsequent lines typically list the values of the parameters to that function when it was called.

  When a function returns (**<**), the name of the function that generated the trace statement and the name of the source file in which this function is present are listed. If the returning function has a return value (**<R**), the function and source file names are followed by the hexadecimal value of the return value.

  When a fatal error occurs (**F**), two lines of trace output are displayed: the first line of trace output describes the error condition, and the second line identifies the name and line number of the Encina source file in which the error was detected.

The generation of audit, warning (error), and fatal messages is always enabled when the appropriate conditions are encountered. Examples of the format for Encina audit, warning, and fatal messages is shown in Figure 54.

```
31 22644 94/05/26-19:43:43.724026 30349826 F  Undo operation has\
 repeatedly failed
31 22644 94/05/26-19:43:43.744425 00000006 F /project/oltp112\
/build/prod/oltp/source/src/server/rec/opevent.c 4492
 1 22753 94/03/28-08:17:51.003038 a0040437 W  Unable to bind to\
 server /.:/encina/sfs/server1 (DCE-rpc-0214: not registered in\
 endpoint map)

 2 01324 94/08/18-00:43:47.419541 6084e018 A  sfs: Initializing ...\
 Thu Aug 18 00:43:47 1994
```

Figure 54. Example error messages

Functions for obtaining trace output in applications

Within applications, all trace, dump, warning, and fatal error output produced by the Encina tracing facility is stored in a circular, internal buffer (ring buffer). The size and type of information stored in the ring buffer is of defined in the trace buffer header, which is pointed to by an instance of the type **trace_buffer_t**. See the reference page for this function for a description of this data type.

Tracing information obtained using the Toolkit administrative tool, or by calling the administrative RPC interfaces to the Toolkit, is automatically extracted from the ring buffer and formatted. Within applications, you must call the **trace_Register** function to register an upcall for all of the classes of tracing events you want to obtain. The limit on the number of upcalls that can be registered for each class of tracing output is specified in the tracing constant `TRACE_MAX_UPCALLS`.

The Encina Toolkit tracing facility provides the **trace_FileUpcall** upcall to implement Encina tracing to the **stderr** or **stdout** devices via the **trace_Register** function. This upcall should be used unless you are integrating Encina tracing with existing tracing mechanisms such as the AIX trace mechanism. An alternate upcall is provided to for this sort of integration. See "Using the AIX trace and log facilities" on page 243 for more information on integrating Encina and AIX tracing.

The following is a typical example of using the **trace_FileUpcall** function to register tracing output:

```
trace_Register(TRACE_CLASS_ENTRY, trace_FileUpcall, stderr);
```

This call directs the output of entry tracing to the error output stream, `stderr`. Substituting another file pointer for `stderr` in this example would write all entry tracing output to that file.

Registering upcalls for certain tracing events eliminates the need to explicitly display or monitor the contents of the ring buffer where trace output is stored. However, in case this is necessary, or for better performance, the Encina tracing facility provides the **trace_DumpRingBuffer** function.

For example, a call to this function to write the contents of the tracing ring buffer to the standard error output stream, in formatted fashion, would be the following:

```
trace_DumpRingBuffer((void *) stderr, TRUE);
```

As mentioned previously, it is usually unnecessary to explicitly call the **trace_DumpRingBuffer** function. In most cases, trace output from Encina can be obtained using the upcalls discussed earlier in this section.

To format tracing information collected from the ring buffer, the Encina tracing facility provides the **trace_FormatBuffer** function. This function takes three parameters; a pointer to a buffer holding the tracing information, the length of the buffer supplied to hold the formatted output, and a pointer to a buffer used to hold the formatted output. This function returns the length of the formatted event description written in the supplied buffer. If the buffer provided to the **trace_FormatBuffer** function is too small to hold the entire formatted trace event string, the string is truncated to fit in the buffer.

Redirecting tracing output in an application

The trace facility functions can be used at any time to direct trace output to a new output stream. The following example shows the redirection of trace output to the file */tmp/trace.out*, appending entry and parameter tracing information to any existing contents of the file:

```
FILE *newOutputStream;
if ((newOutputStream = fopen("/tmp/trace.out", "a+")) != NULL) {
    trace_Register(TRACE_CLASS_ENTRY | TRACE_CLASS_PARAM,
         trace_FileUpcall, (void *) newOutputStream); }
```

In this example, **trace_Register** and **trace_FileUpcall** are the functions provided by the trace facility to register and unregister destinations for specific trace output classes. The **trace_FileUpcall** function is provided by the trace facility to interpret the trace output and send the result to a previously opened file (in this case, **/tmp/trace.out**). These functions are discussed in "Functions for obtaining trace output in applications" on page 241.

Disabling specific trace class output

The **trace_Unregister** function is used to programmatically unregister upcalls for the specified class(es) of events.

Unregistering a single upcall for a specific class of tracing event does not affect any other upcalls registered for that event. Each upcall for specific events must be registered and unregistered separately.

Using the AIX trace and log facilities

AIX provides integrated tracing and error logging facilities as consistent mechanisms for obtaining tracing and error information about different applications and packages. When tracing is enabled for an application or package, a system trace daemon collects and logs all traceable events that correspond to the specified level of tracing. Similarly, error reports from packages that are integrated with the AIX error logging facility write error messages to a central error log. The applications or products that generated specific trace messages are identified through unique, predefined numbers that are assigned by IBM to specific applications and software vendors. These unique identifiers are referred to as *hook IDs*. The hook ID for Encina products is **294**. The hook ID is only meaningful for trace events written to the AIX trace stream. Tracing messages written to the central error log are identified using a string identifier, rather than a hook ID.

Encina provides the **trace_AixErrlogUpcall** and **trace_AixTraceUpcall** upcalls to enable developers to integrate Encina tracing and audit messages with the AIX trace and error logging mechanisms. Like the **trace_FileUpcall**, these upcalls are supplied as an argument to the **trace_Register** function. The **trace_AixTraceUpcall** upcall directs the specified class of trace output to the AIX trace stream. The **trace_AixErrlogUpcall** upcall directs the specified class of trace output to the AIX error logging facility. Having two separate upcalls

enables Encina applications to direct, for example, fatal error messages (messages for trace events of class TRACE_CLASS_FATAL) to the AIX trace facility, while directing all other messages for other trace classes to the AIX error logging facility.

Once tracing information for an Encina application has been collected, it must be formatted for display. AIX systems provide the **trcrpt** utility to process and format the tracing events that have been logged in a specific file. AIX systems also provide the **errpt** utility to process and format any fatal error messages that have been received from a specific application. To use these utilities, the format templates for these types of messages must have been installed on the system. These files define the format of the messages that have been collected or logged. For any fatal error messages that may be generated by Encina applications, this is the file **aix_err_template**, which can be installed using the AIX **errupdate** utility. All other registered levels of tracing events are formatted using the template contained in the file **aix_trcfmt**, which can be installed using the AIX **trcupdate** utility.

For more information about any of the commands used for collecting, installing, or displaying trace or error information on AIX system, see the AIX system documentation for the appropriate command.

Redirecting tracing output by using the Encina administration tool tkadmin

The Encina Toolkit administration tool **tkadmin** simplifies redirecting tracing output in running servers. To specify that the output from entry/exit and parameter tracing be redirected to the file **/tmp/trace.out**, a system administrator (or someone with administrative privileges for the Encina Toolkit, would execute a command like the following:

```
% tkadmin redirect trace entry param -filename "/tmp/trace.out"
```

The **entry** and **param** arguments are the names used by the **tkadmin** application to identify entry and parameter tracing.

For more information about using the **tkadmin** command, see the document entitled *Encina Administration Guide Volume 1: Basic Administration*.

Redirecting tracing output through administrative RPC interfaces

The Encina Toolkit provides direct access to all of the administrative interface functions used to implement the **tkadmin** tool. For example, to redirect trace output to the file **/tmp/trace.out**, a client application can directly issue the following RPC to a server that uses the services provided by the Encina Toolkit Executive:

```
admin_trace_RedirectTrace(rpcHandle,
              TRACE_CLASS_ENTRY | TRACE_CLASS_PARAM,
              "/tmp/trace.out");
```

Making this call requires that *rpcHandle* be a DCE RPC binding handle for the server.

Specific information about the administrative RPC interfaces exported by the various modules of the Encina Toolkit, and about how to use those interfaces, is provided with the documentation for each module.

Obtaining tracing information after application failures

If an application terminates abnormally while tracing is enabled, the Encina Toolkit saves all the information found in the tracing ring buffer for that application. The information is saved automatically and stored in a file located in the directory from which the application was started. The file is named **EncinaTraceBuffer.***PID*, where *PID* is the process ID of the application. This file, commonly referred to as the *ring buffer dump* file, contains the contents of the tracing ring buffer in binary format.

The file containing the ring buffer dump is created only if an application terminates abnormally while tracing is enabled. To interpret the contents of this binary file, the Encina Toolkit provides the **interpretTrace** command. This program reads from standard input, taking a binary ring buffer file as input, and produces an ASCII translation of its input on standard output. On operating systems such as UNIX that support output redirection, you can redirect the output of this program to a file. The following is an example of using this program:

```
% interpretTrace -f EncinaTraceBuffer.17123
```

If the tracing output produced by this program does not reveal the reason that your application terminated, please see your Encina site contact.

Obtaining tracing information from aborted transactions

After a transaction is aborted, the **encina_abortReason_t** data structure that contains the abort reason for the transaction may also contain trace information. Only abort reasons that follow the format used by Encina status codes (that is, have `ENCINA_STANDARD_FORMAT_UUID` in the *formatUuid* field of the abort reason data structure) can contain the trace buffer. This trace buffer can provide additional information about the cause of the abort, such as an XA return code or a DCE status code. (See “Chapter 5. Abort Facility” on page 91 for more information on abort reasons.)

The Trace Facility exports the **trace_IdentifyAbortData** function, which finds the trace buffer (if one exists) contained in an abort reason. Application programs normally do not need to call this function to access the trace buffer data stored in an abort reason as many of the higher-level Encina functions that retrieve abort reasons do so automatically.

Data types and functions for Encina tracing

The following data types are exported by the Encina Trace Facility:

- **trace_buffer_t**
- **trace_uid_t**

The following functions are exported by the Encina Trace Facility:

- **trace_AixErrlogUpcall**
- **trace_AixTraceUpcall**
- **trace_AixUpcall**
- **trace_DumpRingBuffer**
- **trace_FileUpcall**
- **trace_FormatBuffer**
- **trace_Register**
- **trace_Unregister**

Appendix B. Error messages from other components

Encina uses distributed services provided by the Open Software Foundation Distributed Computing Environment (DCE). In addition to using DCE, Encina components are built by using the Base Development Environment (BDE), Encina's internal portability layer, and the Encina utilities package. The BDE and Encina utilities are internal components whose interfaces are not exported by Encina. Although the interfaces to these components are not available to Encina users, error messages generated by Encina servers and applications can include messages from such internal components. These messages are processed like other Encina messages.

By default, any error messages generated by Encina servers and applications are placed in a ring buffer in the server and sent to the server's **stderr** device. The administrator can optionally redirect fatal or warning messages to a file using the **tkadmin redirect trace** command.

This appendix lists the error messages returned by the Encina BDE and the Encina utilities package and suggests appropriate actions to take to recover from the errors.

BDE messages

Encina components are built using the BDE, Encina's internal portability layer. The BDE and Encina utilities are internal components whose interfaces are not exported by Encina. Although the interfaces to these components are not available to Encina users, error messages generated by Encina servers and applications can include messages from the BDE. These messages are processed like other Encina messages.

BDE fatal error messages

The BDE generates a *fatal runtime error* whenever an unrecoverable event occurs. The runtime error is written by default to **stderr**. Applications can redirect the output to an alternative destination by using the **trace_Register**, **trace_Unregister**, or **trace_FileUpcall** function. This section defines the fatal error messages that can be out put by the BDE and briefly describes why they occur. If a BDE fatal message occurs, please make a note of the message and report it to your local Encina service representative.

The types of fatal errors in the BDE are listed below. Since the same error type is used from multiple places in the BDE, each message is preceded by the file and line number of the BDE source where the error actually occurred. In addition, each message is followed by one word of data in brackets. This data

is additional information that can aid BDE developers in tracing the cause of the defect. BDE fatal error messages have the following format:
FileName:LineNumber:"messageString" [0x#]

The BDE supplies some usage errors when the BDE interface is used in an invalid manner. However, in some cases, an invalid usage is not detected. For example, **bde_ThreadGetData** is a performance-critical routine, and typically assumes that the slot parameter is valid.

BDE provides the following fatal error messages:

- "Out of resource: *resource*"—No more objects of the specified type are available. Examples include **bde_alarm_t**, **bde_thread_t,** and **bde_threadSlot_t.** For **bde_thread_t**, for instance, this does not mean that another thread could not be created, it means that the possible values for the **bde_thread_t** type have been exhausted. This error is rare.
- "System call failure: *call*, errno *errno-#* (*errno-string*)"—A system call has unexpectedly failed. Check system documentation for specified system call and report the error to your local Encina service representative.
- "Internal call failure: *call*, status *status-#* (*status-string*)"—An internal routine unexpectedly failed. Report the error to your local Encina service representative.
- "Memory failure: Could not allocate memory for *variable* of size *size* bytes."—An internal attempt to allocate a block of memory of the specified size failed. Review program's use of memory. Make sure that allocated memory is appropriately freed.
- "Call to *call* returned unexpectedly."—An internal call unexpectedly returned. For example, an internal call to **longjmp** or **bde_ThreadExit** should not return. Report the error to your local Encina service representative.
- "Invalid value for parameter *parameter* (*hex-value*>=) provided to *call*."—A parameter or variable contained an unexpected value; for example, an invalid thread identifier was passed to **bde_ThreadDetach**. The name of the variable or parameter, along with the unexpected value, are printed as part of the failure message. Report the error to your local Encina service representative.
- "Call to unsupported routine (*routine*)."—A call has been made to a routine not supported by the BDE. For example, **bde_uxio_Fork** is not supported in the Multiple-process BDE. Calling this routine would raise this error. Report the error to your local Encina service representative.
- "Usage error: Attempted to detach thread (*thread-id*) that is currently being joined."—An attempt was made to detach the specified thread, but the thread is currently being joined. Report the error to your local Encina service representative.

- ″Usage error: Attempted to join with thread (*thread-id*) that is detached.″—An attempt was made to join with the specified thread, but the thread is detached. Report the error to your local Encina service representative.
- ″Usage error: bde_ThreadSleep called within alarm handler.″—It is illegal to call **bde_ThreadSleep** from within an alarm handler. Report the error to your local Encina service representative.

BDE warning messages

The BDE generates a *warning message* whenever an exceptional event occurs. The warning is written by default to **stderr**. Applications can redirect the output to an alternative destination by using the **trace_Register**, **trace_Unregister**, or**trace_FileUpcall** function. This section defines the warning messages that may be output by the BDE and briefly describes why they occur. If a BDE fatal message occurs, please make a note of the message, and report it to your local Encina service representative.

- ″Thread *thread-id* is exiting due to uncaught exception.″—Either the exception occurred when it was not supposed to occur, or the exception was not handled by the application program. In the first case, correct the application so this exception is not raised. In the second case, provide a TRY/CATCH clause for this exception in the application.
- ″Program is exiting due to uncaught exception in main thread.″—The main thread received an exception. Because it is the main thread, it will exit with **bde_Exit**. See suggested recourses listed above.

In the coroutine and multiple-process BDE, the following two warnings are printed to the registered file descriptor, rather than to **stderr** as documented. In the DCE implementation, these warnings call the underlying DCE functions, over which the BDE has no control. Therefore, output is written to **stderr**.

- ″Exception: *exception exception#*″—The uncaught exception has been identified by name and number.
- ″Exception: unknown exception (*state information*)″—The uncaught exception did not have an exception name associated with it, so state information will be printed instead.

Encina utilities messages

The Encina utilities package returns the following fatal runtime error message when an internal attempt is made to allocate a block of memory. Review the program's use of memory and make sure that the allocated memory is appropriately freed.

```
Message_____________________________________
     Action_____________________________________
"*** Memory error error ***
File: filename
Line: line number"
```

DCE messages

Check your DCE documentation or with your DCE supplier for information about DCE error messages and how to recover from them.

Appendix C. Administrative RPC interfaces for the Encina Toolkit

This appendix provides general information about the remote procedure call (RPC) interfaces to the administrative functions exported by the Encina Toolkit. These RPC interfaces are provided to aid in the development of online facilities for administering servers built using the Encina Toolkit.

Servers constructed by using the Encina Toolkit rely on the services provided by various modules of the Toolkit. For example, transactional servers typically use the Toolkit Lock (LOCK) Service to obtain and mediate locks, the Toolkit Volume (VOL) Service to provide a logical interface to physical disk storage, and the Toolkit Recovery (REC) Module to protect against system or media failures. All servers that use these Toolkit modules also share a set of facilities to administer these components. Each server, however, also has its own specific administrative needs that are defined by the resources it manages. For example, the Encina Structured File Server (SFS) has to provide means to reorganize the indices it uses to provide access into the structured files it manages. Each of these administrative facilities has two components: an RPC interface and an interactive program for accessing this interface.

Each server projects two RPC interfaces for administration: one for performing server-specific administration and the other for administering constituent Toolkit components. The second interface is the same for all servers. In addition, an interactive tool is provided for administering the toolkit components—this tool binds to the appropriate server by using the name service. Figure 55 illustrates this architecture.

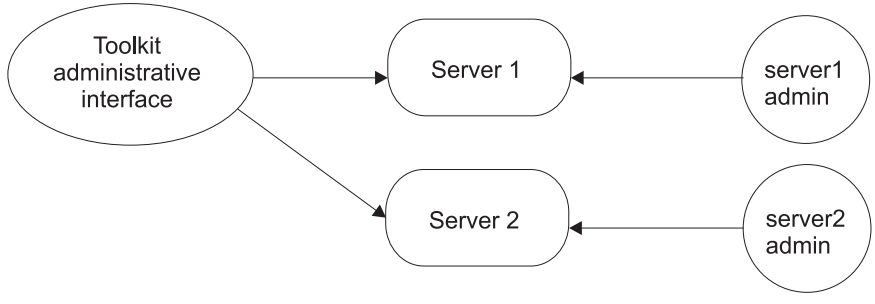

Figure 55. The Encina Toolkit administration model

This appendix discusses the RPC interfaces provided to the administrative functions exported by the modules of the Encina Toolkit. Integrating these

administrative RPC interface definitions with your applications provides remote access to the administrative functions of the underlying Toolkit modules. Additional information about other aspects of Encina administration can be found in the programming documentation for specific servers and in the Encina administration guides.

General information

This section provides general information about the RPC interfaces provided for remote access to the administrative functions of the Encina Toolkit, which includes information about the naming conventions for these functions, parameter profiles for these functions, and error handling for calling these functions. The section concludes by explaining how to incorporate these RPC interfaces into clients.

All of the administrative macros and functions discussed in this portion of this document are either defined in administrative header files (C-language definition files with the **.h** extension) or in Interface Definition Language (IDL) files. IDL is the language used to define interfaces from which client and server stubs can be generated for applications that communicate using the Distributed Computing Environment (DCE) RPC communication mechanism.

Naming conventions for administrative RPC functions

The administrative RPC interfaces exported by the Encina Toolkit are all of the form **admin***Toolkit-component_admin-function-name*. The **admin** prefix indicates that this is an administrative function provided for general Toolkit administration. The next portion of the name, *Toolkit-component*, identifies the Toolkit module that exports the administrative function corresponding to this function. The portion of the name referred to as *admin-function-name* is usually the name of an administrative function exported by the interface of a particular component.

For example, the RPC interface function provided to contact the Transaction Service (TRAN) function **tran_TidTopAncestor** is the **admin_tran_TidTopAncestor** function. The syntax of these two functions follows:

```
tran_status_t tran_TidTopAncestor(
    IN tran_tid_t tid,
    OUT tran_tid_t *topAncestorP)
admin_wireStatus_t admin_tran_TidTopAncestor(
    IN handle_t h,
    IN admin_tran_tid_t tid,
    OUT admin_tran_tid_t *topAncestorP,
    OUT admin_tran_status_t *status)
```

As shown in this example, the RPC interface provided for this function simply exchanges the native Transaction Service data types for the equivalent IDL data types and provides two additional parameters. The first parameter is the **handle_t** parameter, which identifies the server to which the administrative RPC is issued; this is the server on which the appropriate Transaction Service function is actually executed. The second parameter that appears in the IDL administrative function, but not in the actual Toolkit module function, is the **admin_tran_status_t** parameter, which returns (in this case) the actual Transaction Service status code. This parameter is really not new, as it returns the status code returned by the underlying administrative Toolkit function. This last parameter is discussed in "Errors returned by administrative RPC functions". The **admin_wireStatus_t** type is another name for **admin_status_t**, which is described in "Errors returned by administrative RPC functions".

Errors returned by administrative RPC functions

All of the administrative functions discussed in this document return a status code of type **admin_status_t**. It is important to retrieve and translate the underlying, component-specific status codes in case of an error. This is done through the last parameter to each of the administrative interfaces described in this document, which is always an *out* parameter and can therefore be used to relay component-specific error codes back to the administrative application.

Table 9. Possible return values from administrative RPC functions

Return status code	Interpretation
ADMIN_AUTH_FAILURE	The caller is not authorized to execute the function.
ADMIN_SUCCESS	The requested administrative function completed successfully.
ADMIN_SERVER_NOT_IN_ADMIN_MODE	The server has to be in administrative mode for certain commands such as **restore lvols**.
ADMIN_CANNOT_OPEN_TRACE_FILE	The function cannot open the specified file as trace output.
ADMIN_INVALID_COMPONENT	An invalid component for the query, trace, untrace, or dump component was specified.
ADMIN_ALREADY_INITIALIZED	The administrative initialization function has been called.
ADMIN_TRACE_DESTINATION_NOT_INITIALIZED	The trace destination has not been initialized.

It is important to realize that only the combination of a successful status code from the administrative function and a successful component-specific status

code indicates that the administrative function completed successfully. Both status codes must be examined because underlying services can generate exceptions that the administrative interfaces have no way of catching.

Administrative functions return errors that fall into the following four classes:

1. An exception can be raised by the DCE runtime during the execution of the administrative function. For example, `rpc_communication_failure` is raised if the remote call failed due to a communication error.
2. The function can be rejected by the server because the caller is not authorized to execute administrative functions. In this case, the function returns a status code of `ADMIN_AUTH_FAILURE`, and the values of all the *out* parameters of the function are undefined.
3. The function is accepted by the server but the server finds the arguments invalid; for example, an invalid filename was specified for the redirect trace component command.
4. The function returns `ADMIN_SUCCESS`, but there is an error returned by the component-specific error code in the function. For example, the function**admin_tran_TidTopAncestor** returns `ADMIN_SUCCESS` and `TRAN_TID_NOT_VALID` as the values for its status parameter if it is called with an invalid transaction identifier.

Again, an administrative function can be understood to execute successfully only if it generates no exceptions, returns `ADMIN_SUCCESS`, and the component-specific code indicates success.

Using the RPC interfaces in applications

This section describes how to use administrative RPC interfaces in server and client programs. All of the administrative interfaces exported by the Encina Toolkit are nontransactional. There are no guarantees as to whether a specified function completed successfully if a system failure occurs.

Using administrative RPC interfaces in server programs

To export any of the interfaces described in this document in a server program, you must do the following:

1. Include the **admin.h** file, which is located in the **admin** directory in the Encina include directory.
2. Link to the library **admin**. Note that this is not necessary if you already link to the **EncClient** library; the former is subsumed by the latter.
3. If the DCE Security Service is used, register a function provided by your server with the **admin_RegisterAuthorizationCallback** function. An example is

```
admin_RegisterAuthorizationCallback(MyCheckAuthorization)
```

where **MyCheckAuthorization** is a function provided by the server with the following prototype:

```
MyCheckAuthorization(handle_t handle,
                     char * functionName)
```

The variable *handle* is the RPC handle and *functionName* is the RPC manager's function name. This server-provided function verifies the client's authorization on the given function and returns 1 if the authorization test passes and 0 (zero) if the test fails.

Using administrative RPC interfaces in administrative client programs

To use any of the interfaces described in this document in an administrative client, you must do the following:

1. Process the IDL file containing the interface definitions you want to use. The DCE IDL compiler generates the appropriate client and server stubs. The IDL files can be found in the **admin** directory in the Encina **include** directory.
2. For each RPC interface that you want to use, include the IDL-generated header files, compile, and link the IDL-generated client stub files.
3. In your application, before attempting to call any of these administrative interfaces, acquire a *fully bound* RPC handle to the server.

RPC interfaces for Toolkit administration

This section lists the administrative RPC interface for each individual Toolkit component. Using the administrative support provided by Enconsole and by command-line administrative tools for Encina, however, is preferable to coding with the administrative functions.

Administrative RPC functions for the TRAN module

TRAN exports the administrative functions in the following list, which are defined in the files **admin_tran.idl** and **admin_tran_types.idl**.

- **admin_tran_Abort**
- **admin_tran_ApplIdLocal**
- **admin_tran_ApplIsRecoverable**
- **admin_tran_ForceOutcome**
- **admin_tran_GetCoordinator**
- **admin_tran_GetGlobalState**
- **admin_tran_GetLocalState**
- **admin_tran_GetRelCommitState**
- **admin_tran_ListTransactions**
- **admin_tran_PropertyRetrieve**

- **admin_tran_ProvideOutcome**
- **admin_tran_TidKnownDescendents**
- **admin_tran_TidParent**
- **admin_tran_TidTopAncestor**

Administrative RPC functions for the LOG module

LOG exports the administrative functions in the following list, which are defined in the files**admin_log.idl** and **admin_log_types.idl**.

- **admin_log_CreateLogFile**
- **admin_log_DeleteLogFile**
- **admin_log_EnableArchFile**
- **admin_log_InitVol**
- **admin_log_QueryVol**
- **admin_log_RestoreVol**

Administrative RPC functions for the REC module

REC exports the administrative functions in the following list, which are defined in the files **admin_rec.idl** and **admin_rec_types.idl**.

- **admin_rec_BeginBackupSegment** The administrative version of this function takes an explicit filename and file size rather than the **backupFile_t** structure used in the standard Recovery Service function.
- **admin_rec_DisableMediaArchiving**
- **admin_rec_DisableVolume**
- **admin_rec_EnableLogFile**
- **admin_rec_EnableMediaArchiving**
- **admin_rec_EnableVolume**
- **admin_rec_EndBackupSegment**
- **admin_rec_FlushVol**
- **admin_rec_ForceCheckpoint**
- **admin_rec_GetBackupInfo**
- **admin_rec_GetRestoreInfo**
- **admin_rec_QueryBackup**
- **admin_rec_QueryConfig**
- **admin_rec_QueryRestore**
- **admin_rec_QueryVolume**
- **admin_rec_RecoverVolumes**
- **admin_rec_Restore** The administrative RPC interface to this function does not take the *openFileCallback* parameter required by the standard Recovery Service function.

- **admin_rec_RestoreLvols** The *resumeAllowed* parameter determines whether the Recovery Service is allowed to resume a restore that has been interrupted.
- **admin_rec_RetainLastBackup** The *fileList* and *oldestSegment* parameters to the RPC version of this function are OUT rather than INOUT.
- **admin_rec_SetCheckpointInterval**
- **admin_rec_TruncateBackup** The *fileList* and *oldestSegment* parameters are typed as `OUT` rather than `INOUT` (as they are in the standard Recovery Service function) for the convenience of the DCE IDL compiler.

Administrative RPC interfaces for the VOL module

VOL exports the administrative functions in the following list, which are defined in the files **admin_vol.idl** and **admin_vol_types.idl**.

- **admin_vol_AddMirror**
- **admin_vol_CertifyLogicalVol**
- **admin_vol_CertifyPhysicalVol**
- **admin_vol_CreateLogicalVol**
- **admin_vol_CreatePhysicalVol**
- **admin_vol_DestroyDisk**
- **admin_vol_DestroyLogicalVol**
- **admin_vol_DestroyPhysicalVol**
- **admin_vol_DismountLogicalVol**
- **admin_vol_ExpandLogicalVol**
- **admin_vol_ExpandPhysicalVol**
- **admin_vol_GetDiskInfo**
- **admin_vol_GetDiskList**
- **admin_vol_GetLogicalByName**
- **admin_vol_GetLogicalVolInfo**
- **admin_vol_GetLogicalVolList**
- **admin_vol_GetLogicalVolLockInfo**
- **admin_vol_GetLogicalVolLockList**
- **admin_vol_GetPhysicalByName**
- **admin_vol_GetPhysicalVolInfo**
- **admin_vol_GetPhysicalVolList**
- **admin_vol_InitializeDisk**
- **admin_vol_LockLogicalVol**
- **admin_vol_MapLogicalVol**
- **admin_vol_MountLogicalVol**
- **admin_vol_RecreateLogicalVol**

- **admin_vol_RelabelDisk**
- **admin_vol_RelocatePhysicalVol**
- **admin_vol_ReMapVol**
- **admin_vol_RemoveMirror**
- **admin_vol_RenameLogicalVol**
- **admin_vol_RenamePhysicalVol**
- **admin_vol_SyncLogicalVol**
- **admin_vol_UnlockLogicalVol**
- **admin_vol_UnmapLogicalVol**
- **admin_vol_VerifyDisk**

Administrative RPC interfaces for general service

The RPC interface for general Toolkit server administration includes the following functions:

- **admin_gen_CancelAlarm**
- **admin_gen_CheckAuthorization**
- **admin_gen_DumpMemoryPlumbing**
- **admin_gen_ExecGenericCallback**
- **admin_gen_GetLocalIdentity**
- **admin_gen_GetProcessIdentifier**
- **admin_gen_GetWorkingDirectory**
- **admin_gen_QuiesceServer**
- **admin_gen_ShutdownServer**

The interface specification for these functions is listed in the **admin_gen.idl** file.

```
interface admin_gen
{
    import "admin/admin_types.idl";
    typedef unsigned long admin_genStatus_t;
    typedef [string, ptr] char *admin_genString_t;
    admin_wireStatus_t
        admin_gen_CancelAlarm(
                [in] unsigned long alarmTag,
                [out] admin_genStatus_t *statusOut
                );
    admin_wireStatus_t
        admin_gen_CheckAuthorization(
                [in, string] char *functionName,
                [out] unsigned long *authorizedOut
                );
    admin_wireStatus_t
        admin_gen_DumpMemoryPlumbing(
                [in] unsigned long flags,
                [in, string] char *fileName
```

```
            );

    admin_wireStatus_t
        admin_gen_ExecGenericCallback(
                [in, string] char *callbackName,
                [in, size_is(len)] byte blob[],
                [in] unsigned long len,
                [out] admin_genStatus_t *statusOut,
                [out] unsigned long *callbackStatusOut
                );

     admin_wireStatus_t
        admin_gen_GetLocalIdentity(
                [out, ref] admin_genIdentityList_t *identities
                );


     admin_wireStatus_t
        admin_gen_GetProcessIdentifier(
                [out] unsigned long *processIdOut
                );

     admin_wireStatus_t
        admin_gen_GetWorkingDirectory(
                [out] admin_genString_t *workingDirectory
                ):
     admin_wireStatus_t
        admin_gen_QuiesceServer(
                [in] unsigned long exitCode,
                [in] unsigned long seconds,
                [out] unsigned long *alarmTagOut,
                [out] admin_genStatus_t *statusOut
                );

     admin_wireStatus_t
        admin_gen_ShutdownServer(
                [in] unsigned long exitCode,
                [in] unsigned long seconds,
                [out] unsigned long *alarmTagOut,
                [out] admin_genStatus_t *statusOut
                );


     typedef struct {
        unsigned long identifierCount;
        [ptr, size_is(identifierCount)] unsigned long *identifiers;
        [ptr, size_is(identifierCount)] admin_genString_t *identifierNames;
        [ptr, size_is(identifierCount)] admin_genString_t *identifierTypes;
      } admin_genIdentityList_t;


}
```

Administrative RPC interfaces for the TM-XA module

The Toolkit TM-XA module exports administrative functions that use the following two data types:

```
typedef struct {
     char                  name[RMNAMESZ];
     int                     rmid;
     unsigned long   tmxaFlags:
} admin_tmxa_RMID_t;
typedef struct {
     [string] char           *openInfo;
     [string] char           *closeInfo;
     [ptr] xa_switch_t   *switchP;
     int                              rmid;
     int                              threadSupport;
     unsigned long           tmxaFlags
} admin_tmxa_RMInfo_t;
```

The functions in the RPC interface have the following signatures:

- `admin_wireStatus_t` **`admin_tmxa_DisableRMI`** `(`
  `[in] int rmid,`
  `[in] int force,`
  `[out] admin_tmxa_status_t *status )`
- `admin_wireStatus_t` **`admin_tmxa_EnableRMI`** `(`
  `[in] int rmid,`
  `[out] admin_tmxa_status_t *status )`
- `admin_wireStatus_t` **`admin_tmxa_ListRMIs`** `(`
  `[out, ptr] int *numRms,`
  `[out, ptr, size_is(numRms)] admin_tmxa_RMID_t *rmlist )`
- `admin_wireStatus_t` **`admin_tmxa_QueryRMI`** `(`
  `[in] int rmid,`
  `[out, ptr] admin_tmxa_RMInfo_t **rmInfo)`

See the programming reference pages for information on the corresponding library functions.

Administrative RPC interfaces for the Trace module

The support for tracing provided in the Encina Toolkit includes administrative RPC interfaces that enable applications to extract tracing information programmatically. These functions are provided for use in administrative applications. The tracing support provided through Enconsole and the command-line administrative tools for Encina should be used whenever possible, rather than coding applications to use the administrative functions directly.

The Toolkit Tracing Module exports the following administrative functions:

- **admin_trace_AddTraceSpec**
- **admin_trace_DumpRingBuffer**
- **admin_trace_GetComponentList**

- **admin_trace_GetComponentMask**
- **admin_trace_GetModuleVersion**
- **admin_trace_GetProductList**
- **admin_trace_QueryRedirect**
- **admin_trace_RedirectTrace**
- **admin_trace_SetComponentMask**
- **admin_trace_SetProductMask**
- **admin_trace_StateDump**

The functions in this section are shown in the context of the TIDL (Transactional Interface Definition Language) file that defines them, **admin_trace.idl**, shown in Figure 56 on page 262. The data types used in these functions are defined in the IDL file **admin_types.idl**, shown in Figure 57 on page 263. See "Appendix C. Administrative RPC interfaces for the Encina Toolkit" on page 251 for information about incorporating calls to these functions in an Encina application.

```
interface admin_trace
{
    import "admin/admin_types.idl";
    admin_wireStatus_t admin_trace_SetComponentMask(
        [in, ptr, string] char *componentName,
        [in] unsigned long traceMask);
    admin_wireStatus_t admin_trace_GetComponentMask(
        [in, ptr, string] char *componentName,
        [out] unsigned long *traceMaskP );
    admin_wireStatus_t admin_trace_GetComponentList(
        [in, ptr, string] char *productName,
        [out] admin_strings_t *componentNames);
    admin_wireStatus_t admin_trace_SetProductMask(
        [in, ptr, string] char *productName,
        [in] unsigned long traceMask);
    admin_wireStatus_t admin_trace_GetProductList(
        [out] admin_strings_t *productNames );
    admin_wireStatus_t admin_trace_RedirectTrace(
        [in] unsigned long klass,
        [in, ptr, string] char *traceFile);
    admin_wireStatus_tadmin_trace_StateDump(
        [in, ptr, string] char *componentName );

    admin_wireStatus_t admin_trace_DumpRingBuffer(
        [in, ptr, string] char *fileName);

    admin_wireStatus_t admin_trace_GetModuleVersion(
        [in, string] char *componentName,
        [out] admin_string_t *versionString);
    admin_wireStatus_t admin_trace_AddTraceSpec(
        [in, string] char *specification );

    admin_wireStatus_t admin_trace_QueryRedirect(
        [in] unsigned long traceClass,
        [out] admin_string_t *traceFile);
}
```

Figure 56. admin_trace.idl

```
interface admin_types
{
    typedef long unsigned admin_wireStatus_t;

    typedef struct {
        [ptr, string] char *strP;
    } admin_string_t;

    typedef struct {
        unsigned long listLength;
        [ptr, size_is(listLength)] admin_string_t *strings;
    } admin_strings_t;
}
```

Figure 57. admin_types.idl

Appendix D. Using the Toolkit to write a recoverable server

This appendix discusses how to develop recoverable servers using components of the Encina Toolkit and Tran-C.

General server information

When you write a server application, you can organize the program into three main parts: initialization code, callback and upcall definitions, and the manager functions that constitute the interface for whatever service it provides.

Initialization typically involves setting up the Transactional RPC (TRPC) runtime environment, beginning to initialize Tran-C, initializing or recovering any recoverable data used by the server, completing Tran-C initialization, and finally making the server available to clients on the network. Setting up TRPC differs slightly depending on whether the server uses the Distributed Computing Environment (DCE) Directory Service to make itself available to clients or whether it establishes itself at a well-known endpoint. Handling recoverable data and serializing accesses to it entails interacting with the Log Service (LOG), Recovery Service (REC), Volume Service (VOL), and the Lock Service (LOCK).

Callbacks and upcalls are routines the server implements and registers with the services it incorporates (for example, Recovery, Lock, and Volume). These services invoke the registered routines in the user's application when special events occur. For example, when REC undoes modifications to recoverable data due to a transaction's aborting, it invokes an undo upcall the application registered when it initialized REC.

The manager functions are the functions that a client invokes by using remote procedure calls (RPCs). These functions implement the application's business logic.

Initializing server applications

Toolkit servers using Tran-C typically use a two-stage initialization process. The two-stage method uses the **preInitTC** and **postInitTC** functions, allowing an application to initialize services that must be initialized after Tran-C begins to initialize and before Tran-C completes its initialization.

The example (called the merchandise server) shows typical server initialization in its main program body (see Figure 58 on page 267) and in its **Initialize** function (see Figure 59 on page 268). These examples are discussed in the sections that follow.

General overview of initialization

The merchandise server first calls its **ParseArguments** function to get information from command-line arguments (for example, the server name and volume names). The server then sets up principal and key files for Transarc/Encina DCE (TRDCE), creates a login context, and registers authentication information with DCE. This step is required to send or receive secure RPCs. Even though a server does not require secure RPCs, it cannot receive them from secure clients without registering authentication information with the runtime environment. See "Security issues" on page 268 for more information.

Initialization of a Tran-C application typically consists of calling the **preInitTC** function, initializing the communications and recovery components, and then calling the **postInitTC** function. The **Initialize** function for the merchandise server encapsulates the Tran-C initialization steps (see Figure 59 on page 268).

The **Initialize** function initializes communication by calling the **trdce_ServerRegister** function to register the server with the Directory Service or set up a well-known endpoint. (See "Making servers available to clients" on page 280.) It then calls the **trpc_InitWithTrdce** function, which informs TRPC that TRDCE functions have been used for server registration and requests that TRPC use any protocol sequences and well-known endpoints described by those TRDCE calls. The **tc_InitTRPC** function declares that the TRPC component will be used for communications.

Recovery initialization is performed as part of the **recArray_Init** function, which sets up the recoverable array used by the server. This function is shown in Figure 60 on page 270 . The **recArray_Init** function performs most of the merchandise server's initialization, because it initializes the recoverable array, Recovery Service, Lock Service, and Volume Service.

After completing the initialization of the other services used by a server, the server finishes initializing Tran-C by calling **postInitTC**. At this point, servers can call the **deadlockDetect** function, which periodically looks for local deadlocks among Lock Service requests and aborts transactions to resolve conflicts.

The merchandise server calls the DCE **rpc_server_register_if** function to register its name with the RPC runtime. Then it calls the **trdce_ServerListen** function to spawn threads that listen for incoming RPCs and invoke the

appropriate manager functions. Supplying zero as the first argument indicates that this function spawns the default number of threads for the thread pool.

```
/* main -- perform necessary initializations then start server */
int main(int argc, char **argv)
{
   merchandiseData_configParams_t configParams;
   unsigned32 status;
   ParseArguments(argc, argv, &configParams);
   /* Set the principal and key file for trdce */
   trdce_SetPrincipal((unsigned_char_t *)configParams.principalName,
                      &status);
   CHECK_STATUS(status);
   trdce_SetKeyFile((unsigned_char_t *)configParams.keyFile,
                    &status);
   CHECK_STATUS(status);
   /* Create the login context, register the auth. info with DCE,
    * and manage the login context and the keyfile. */
   trdce_SecManagement(&status);
   CHECK_STATUS(status);

   /* Initialize Encina Components */
   Initialize(configParams.serverName,
              configParams.volRestartString,
              configParams.logVolName,
              configParams.logArchDevName,
              configParams.dataVolName);

   StockUpIfFirstRun();

   deadlockDetect(MERCHANDISE_DEADLOCK_INTERVAL);

   /* Register with RPC runtime */
   rpc_server_register_if(merchandise_v1_0_s_ifspec, NULL,
                          merchandise_v1_0_mgr_epv, &status);
   CHECK_STATUS(status);

   /* Allow anyone admin privileges for this example's purposes */
   admin_RegisterAuthorizationCallback(AnyoneCanAdminister);

   /* Start server listening (if it isn't already listening) */
   printf("[Merchandise server is running.]\n");
   trdce_ServerListen(0, &status);
   CHECK_STATUS(status);
   exitTC(0);
}
```

Figure 58. The main routine from the merchandise server

```
/* Initialize:  Initialize the Encina components used: Tran-C,
   TRPC, and the Encina Server Core, through recArray. */
static void Initialize(char *serverName, char *volRestartString, char *logVolName,
                       char *logArchDevName, char *dataVolName)
{
    unsigned long status;
    /* Begin initialization of TRPC, Tran-C and recoverable array */
    preInitTC();
    /* Register with CDS and initialize TRPC */
    trdce_ServerRegister((unsigned char *)serverName, &status);
    CHECK_STATUS(status);
    status = trpc_InitWithTrdce();
    CHECK_STATUS(status);
    tc_InitTRPC();

    /* Initialize underlying data storage package */
    recArray_Init(volRestartString, logVolName, logArchDevName,
                  dataVolName);

    /* Finish Tran-C and recoverable array initialization */
    postInitTC();
}
```

Figure 59. The Initialize routine from the merchandise server

Security issues

Toolkit server applications must manage some security information regardless of whether they need to run in a secure environment. This management enables secure clients to send secure RPCs to a server. A server must register a principal ID and provide a DCE key file to make it possible for a secure client to communicate with the server. The sample merchandise server calls the **ParseArguments** function to get the principal ID and the DCE key file name from the command line arguments. It then registers the principal ID and key file names with the DCE runtime and calls **trdce_SecManagement**, which performs basic security functions for the current principal and key file.

Listening for RPCs

After a server makes itself available to clients as part of its initialization, the server needs to start listening for incoming RPCs by calling the Encina **trdce_ServerListen** function instead of the DCE **rpc_server_listen** function, which ensures the server is listening for RPCs and to cause the calling thread to sleep every time it is called in a process, as long as no thread has called **rpc_server_listen**. Any Encina component that must start the server listening for RPCs as part of its initialization, calls the **trdce_ServerListen** function.

Servers with recoverable data

Services used to maintain recoverable data include:

- The Log Service records the modifications to the data and the locks that were held when the data was changed.
- The Recovery Service coordinates the logging, updates to memory during recovery, and undoing a transaction's updates when an abort occurs.
- The Lock Service provides the mechanism for isolating transactions.
- The Volume Service manages the physical storage containing the data.

These services must be initialized. Part of this initialization involves the restart data (see "Chapter 11. Restart Service" on page 167) and any data recovery necessary due to the server crashing during a previous execution.

Figure 60 on page 270 shows the **recArray_Init** function from the merchandise server, and it contains a typical initialization sequence that handles all the services mentioned above:

1. This function first initializes the Recovery Service to which it passes information for initializing LOG and several pointers to routines that are upcalls required by REC (see "Recovery Service upcalls" on page 276, "Volume Service upcalls" on page 277, and "Lock Service upcalls" on page 278). After initializing LOG, the Recovery Service opens the log file identified as the first argument to the **rec_Init** function. The second argument is the minimum protection level required by the Log Service for REC to use when communicating with LOG. After a server calls **rec_Init**, it must check its restart data, either creating the restart data if the server is running for the first time, or retrieving the restart data if the server previously was running. REC and LOG interact to provide the restart data if it is available.
2. Next, a server initializes the LOCK, which requires the restart data that LOCK caused the application to store for it on previous executions. If the server has never run before, there is no restart data for the Lock Service. The call to the **lock_Init** function also takes a reference to a required upcall routine that LOCK calls to request that the application save restart data on behalf of LOCK.
3. At the end of a server's initialization, the **recArray_Init** function initializes VOL and either initializes or recovers the volume that holds its recoverable data. If the server has never run before, this routine makes the volume available to the application. If the server is restarting, this function replays the log to bring the recoverable data stored in the volume up-to-date.

```
/*  recArray_Init -- Initialize the recArray package.  Must be called
 *  between preInitTC and postInitTC.  */
void recArray_Init(char *volRestartString, char *logVolName,
                   char * logArchDevName, char *dataVolName)
 {
    rec_descr_t assocBufferDataDescr;
    rec_configParams_t recConfigParams;
    void *restartDataP;
    int firstRun = (dataVolName != NULL);       /* boolean */
    restart_mode_t volRestartMode;
    char logFileName[256];
    unsigned long status;
    /* Initialize VOL. */
    volRestartMode = server_Restart(volRestartString);
    if (volRestartMode == RES_COLD_START && !firstRun) {
        FATAL(("Must specify data volume name\n"));
    }
    /* Determine the full name of the log file by prefixing volume name. */
    sprintf(logFileName, "%s/%s", logVolName, TELSHOP_LOG_FILE_NAME);

    /* Create a log file if this is the first time. */
    if (firstRun) {
        server_CreateLogFile(logFileName, logVolName, logArchDevName);
    }

    /* Initialize REC */
    assocBufferDataDescr.descrP = NULL;  /* Not using assocBuff facility */
    assocBufferDataDescr.descrLength = 0;
    recConfigParams.bufferPoolSize = BUFFER_POOL_SIZE;
    recConfigParams.maxRestartTime = MAX_RESTART_TIME;
    rec_Init((log_file_t) logFileName, LOG_NO_AUTHENTICATION,
             RedoPageUpdate, UndoPageUpdateSet, UndoOp, DropLocks,
             ReobtainLocks, BuffersExhausted, &assocBufferDataDescr,
             &recConfigParams);

    /* Initialize in-memory copy of restart data (ours and LOCK's) */
    GetInitialRestartData(dataVolName, &restartDataP);
```

Figure 60. Initializing the recArray package

```
    /* Initialize LOCK */
    status = lock_Init(RD_LOCK_DATA_P(restartDataP),
                       RD_HEADER(restartDataP).lockDataLen,
                       LockSaveRestartData);
    CHECK_STATUS(status);

    /*  Initialize or recover our data volume  */
    if (firstRun) {             /*  Initialize volume */
        int numPages;

        volId = server_CreateVolume(dataVolName, CHUNK_SIZE);

        status = rec_EnableVolume(volId);
        CHECK_STATUS(status);
        SetVolumeGlobals();
        rec_RecoverVolumes();

        /* Zero out the array */
        numPages =
           (REC_ARRAY_SIZE+(recArrayElsPerPage-1))/recArrayElsPerPage;
        status = rec_InitVolume(volId, 0, numPages-1, rec_writeOnly, TRUE);
        CHECK_STATUS(status);

        RecordInitialization();
    } else {                    /*  Recover volume */
        /* Get the logical-volume identifier. */
        status = vol_GetLogicalByName(RD_HEADER(restartDataP).volName,
                                      &volId);
        CHECK_STATUS(status);

        SetVolumeGlobals();
        rec_RecoverVolumes();
    }

    free(restartDataP);
}
```

Figure 61. The recArray_Init function (continued)

Using RPC interfaces

Applications use remote procedure calls (RPCs) to communicate with other applications. Tran-C uses transactional remote procedure calls (TRPCs) and out-of-band data as communications mechanisms. Application developers can use either the Directory Service or well-known endpoints to establish communication between two entities (see "Making servers available to clients" on page 280).

Transactional RPC

The data structures and limited modes of indicating success or failure provided by standard RPCs are not robust enough for a transactional environment. Remote procedure calls typically identify only the host system and port from which the RPC was issued. This is not precise enough for a transaction processing environment, where many transactions and RPCs can take place simultaneously and where the remote process identification information must also include the ID of the transaction that issued the RPC. This extra information is especially important in a threaded environment, where a single system, port, and process can send and receive many different RPCs on behalf of multiple threads.

TRPCs are based on DCE RPCs, and they provide stronger guarantees of success and failure than standard RPCs. For example, a standard RPC, which does not return a code indicating success or failure, can fail for a number of reasons. It also can fail after partially modifying information on the remote node. Transactional RPCs provide exactly-once execution semantics for remote procedure interaction. If the TRPC returns a code indicating success, the actions specified in the TRPC call are guaranteed to have occurred once and only once on the target machine. If the TRPC fails, or the initiating transaction is notified of an error, the transaction is aborted. The remote function can still have been invoked, but the Toolkit's Distributed Transaction Service notifies the remote application that those updates will be undone. The exactly-once execution semantics of a transactional RPC guarantees that the remote procedure to which the call is placed is executed once if the transaction commits and not at all if it aborts.

Out-of-band data

To coordinate transactions, the Transaction Service (TRAN) requires that the underlying communications mechanism provide a way for TRAN at one application (for example, a client) to exchange information with TRAN at other applications (for example, servers) involved in the same transaction. TRPC passes some Transaction Service information when executing remote procedure calls, but sometimes TRAN needs to send data when TRPC is not sending any RPCs for the application. These auxiliary TRAN messages are referred to as *out-of-band data*.

TRPC provides a default environment that uses the Directory Service to pass out-of-band data. Applications can customize the TRPC environment by calling the **trpc_SetEnvironment** function, but this function is rarely needed, because TRPC uses reasonable defaults for all the settings specified with this call. See the reference page for **trpc_SetEnvironment** for a description of the defaults.

The merchandise server calls the **trpc_InitWithTrdce** function to initialize communications protocols and well-known endpoints for TRPC. This function

instructs TRPC to use the protocol sequences and well-known endpoints specified previously by the TRDCE calls used to register servers.

Using the Recovery Service

Int the writing of applications, recovery typically distinguishes servers from clients. Servers need to be recoverable, so that clients can expect transactionally-consistent data in the presence of aborts, system failures, and media failures. Servers use the Recovery Service (REC), which entails using the Log Service (LOG) and the Volume Service (VOL).

Applications manage persistent data by using the Recovery Service's buffer pool mechanism. This mechanism is backed by the Volume Service, which manages physical pages. Applications request REC to get pages from VOL and pin them in the buffer pool, and the application modifies the storage there. After updating the page, the application issues REC calls, which use the Log Service to maintain the information necessary to redo or undo the update. When the update and logging are complete, the application issues REC calls to signify this and to release the pinned page in the buffer pool. When the server initializes, it makes calls to REC. If the server was previously running and became unavailable, REC can review the log and make calls in the application to redo or undo modifications to bring the server up-to-date.

Updating recoverable data in the sample application

The following code examples show the two routines, **recArray_Write** and **WriteArrayElement**, that perform the recoverable updates to the array.

The **recArray_Write** function sets the value in the array at `arrayIndex` to `value`. Note that there is no locking. This function is meant to be general and able to be compiled into any server. It does not know the locking style of its server; the server does its own locking.

```
/
void recArray_Write(unsigned long arrayIndex, unsigned long value)
{
    tran_tid_t    tid = tc_getTid();
    rec_descr_t   lockDescr;
    rec_opId_t    rootOpId;
    unsigned long status;
    /* Gather the lock descriptions for REC.  REC logs the locks
     * with the update (to know what locks to get again if
     * recovering). */
    status = lock_Save(tid, NULL, 0, &lockDescr.descrP,
                       &lockDescr.descrLength);
    CHECK_STATUS(status);
    /* Get the REC Operation-ID corresponding to the transaction. */
    status = rec_GetRootOp(tid, &rootOpId);
    CHECK_STATUS(status);
    /* Recoverably update the array element. */
    WriteArrayElement(rootOpId, arrayIndex, value, &lockDescr);
}
```

Figure 62. The recArray_Write function

The **recArray_Write** function's parameters are an array index and a value to store in that element of the array. After getting lock and recovery information, **recArray_Write** calls **WriteArrayElement** to do most of the work. The merchandise server's interface routine, **merchandise_OrderItem**, gets a write lock on the index; the **recArray_Write** function obtains from the Lock Service whatever information it needs to reacquire the lock during recovery.

```
static void WriteArrayElement(rec_opId_t opId, unsigned long arrayIndex,
   unsigned long value, rec_descr_t *lockDescrP)
{
    rec_pageId_t pageId;
    updateDescr_t redoInfo , undoInfo;
    rec_descr_t redoDescr, undoDescr;
    rec_descrList_t redoDescrList, undoDescrList;
    void *pageP, *assocBufferDataP;
    unsigned long status;
    /* Pin the page corresponding to array element (for read/write). */
    pageId.pageNum = REC_ARRAY_PAGE_NUM(arrayIndex);
    pageId.volId = volId;
    status = rec_PinPage(&pageId, rec_normal, rec_writeOnly, &pageP,
                         &assocBufferDataP, NULL);
    CHECK_STATUS(status);
    /* Notify REC of intention to update page. */
    status = rec_BeginPageUpdate(pageP);
    CHECK_STATUS(status);
    /* Update the array element after saving its previous value. */
    undoInfo.value = ELT(pageP, arrayIndex);
    ELT(pageP, arrayIndex) = redoInfo.value = value;

    /* Gather the rest of the redo and undo descriptions. */
    redoInfo.arrayIndex = arrayIndex;
    redoDescr.descrP = (void *)&redoInfo;
    redoDescr.descrLength = sizeof(updateDescr_t);
    redoDescrList.nDescrs = 1;
    redoDescrList.descrList = &redoDescr;

    undoInfo.arrayIndex = arrayIndex;
    undoDescr.descrP = (void *)&undoInfo;
    undoDescr.descrLength = sizeof(updateDescr_t);
    undoDescrList.nDescrs = 1;
    undoDescrList.descrList = &undoDescr;

    /* Install the update on the page. */
    status = rec_UpdatePageSet(&pageP, 1, opId, lockDescrP,
                               &redoDescrList,&undoDescrList);
    CHECK_STATUS(status);

    /* End the page update and unpin the page. */
    status = rec_EndPageUpdate(pageP);
    CHECK_STATUS(status);
    status = rec_UnPinPage(pageP);
    CHECK_STATUS(status);
}
```

Figure 63. The WriteArrayElement function

The **WriteArrayElement** function's parameters are the REC and LOCK information obtained in **recArray_Write**, the array index, and the value to store in the array element. This function calls the **rec_PinPage** function to cause REC to get the page from VOL and place it in REC's buffer pool.

Calling **rec_PinPage** also allows the application to specify the mode in which it will redo the modification, and the merchandise server notifies REC that it can reproduce the change to the VOL page using only memory writes. This is important because it enabless

REC to save significant space in VOL, making more of each VOL page available to the application. The merchandise server can do this because it logs all the information it needs to redo any memory writes it performs.

After pinning the page, **WriteArrayElement** calls the **rec_BeginPageUpdate** function to notify REC it intends to modify a VOL page. Next, it saves the current array element's value in the undo structure before writing the new value to the array element, and **WriteArrayElement** fills the remaining undo and redo information. Then this routine calls the **rec_UpdatePageSet** function to pass the undo and redo information to REC for logging, along with the lock information passed into **WriteArrayElement**. To close the update, the function notifies REC that the page modification has occurred and calls the **rec_UnPinPage** function to release the page from REC's buffer pool.

Notice all of the application's interaction with the Log Service and the Volume Service occurs through the Recovery Service. This sample update shows value logging, manipulating undo and redo information, and pinning pages from VOL into REC's buffer pool. To see how the sample merchandise server uses restart data, see "Servers with recoverable data" on page 268.

Recovery Service upcalls

The Recovery Service requires applications to supply routines that it can call in the application to perform necessary work on its behalf. Consider the Recovery Service as a coordinator and record keeper that ensures the application performs certain tasks at certain times. REC does this through these routines called Recovery Service upcalls.

The merchandise server supplies several REC upcalls, which are the same for any server, but the exact code executed is different for applications that perform different services. These upcalls follow:

RedoPageUpdate
: This upcall is invoked during recovery to redo an assignment to an array location. It does not obtain locks to perform this modification because the Recovery Service calls this function before the server is available to handle RPCs, which prevents transactions from contending for server resources during recovery. REC replays the log, redoing each write to memory as it occurred originally.

ReobtainLocks
: This upcall is invoked with a description of locks that were previously held when recoverable data was modified. An application can reobtain

them during recovery. When operations are logged, some locks can be from an alternative mutual exclusion mechanism rather than from the Lock Service. REC calls this function when replaying the log and redoing updates to pages in the Volume Service. Both of these activities together reestablish the runtime state of the server to the point before the failure.

UndoPageUpdateSet
This upcall is invoked when a transaction aborts and the Recovery Service causes the application to undo an update to an array location. When helping a server restart, REC undoes updates as a second pass over the log, removing the effects of aborted transactions. REC does this while allowing the server to handle RPCs, which allows transactions to begin contending for resources. Receiving RPCs while undoing works well because undoing for server recovery is the same as undoing when a transaction aborts. During recovery, as a first pass over the log, REC reobtains locks that the application logged when originally modifying the data (see the call to **rec_UpdatePageSet** in "Updating recoverable data in the sample application" on page 273). Notice how **UndoPageUpdateSet** modifies the page in contrast to how **RedoPageUpdate** does this. The former uses **WriteArrayElement**, which logs its modification, while the latter avoids logging and simply stores its value. An application must log each update it undoes in case it fails after performing an undo operation. When an application restarts, the Recovery Service needs to know the application has already successfully performed a particular undo operation.

UndoOp
This upcall is invoked to undo operations. Because the merchandise server does not log operations, this is never executed, but REC requires the application to supply some function. The merchandise server implements this upcall to signal a fatal error.

DropLocks
This upcall is invoked to drop all locks acquired for a transaction or an operation. When operation logging is used, some locks can be from an alternative mutual-exclusion mechanism rather than from the Lock Service.

BuffersExhausted
This upcall is invoked when a page-pinning request exhausts the configured buffers for the Recovery Service.

Volume Service upcalls

The Volume Service requires applications to supply routines that it can call to log restart data and handle disk errors. These routines are known as Volume Service upcalls.

The sample merchandise server supplies two VOL upcalls:

VolSaveRestartData
This upcall is invoked when the Volume Service needs to save information necessary for it to restart for this application. The sample server calls **UpdateRestartData** to do this. See "Chapter 11. Restart Service" on page 167 for more information.

VolDiskError
This upcall is invoked if the Volume Service is notified of a disk error.

Lock Service upcalls

The Lock Service requires applications to supply a routine that it can call in the application to log restart data. This routine is known as a Lock Service upcall.

The sample merchandise server supplies one LOCK upcall:

LockSaveRestartData
This upcall is invoked when the Lock Service needs to save information necessary for it to reestablish its state for this application. The sample server calls **UpdateRestartData** to do this. See "Chapter 11. Restart Service" on page 167 for more information.

Interacting with TM-XA

The X/Open Distributed Transaction Processing XA interface is a proposed standard for distributed transactional communications. This standard specifies a bidirectional interface between *resource managers* that provide access to shared resources used within transactions and the transaction service that monitors and consistently resolves transactions.

The Encina Transaction Manager-XA (TM-XA) Service enables the Transaction Service (TRAN) to perform as the transaction manager side of the X/Open XA interface. The TM-XA Service therefore allows transactional applications written with the Encina Toolkit to initiate, coordinate, and resolve distributed transactions with XA-compliant resource managers. The TM-XA Service is built using the Distributed Transaction Service (TRAN) and the Thread-to-Tid Mapping Service (ThreadTid) modules of the Encina Toolkit Executive. The TM-XA service supports the version of XA described in the Draft XA Specification of September 1991 (company review draft).

The TM-XA Service maps transaction events to the appropriate XA calls. All information about the transaction context and the execution scope is translated into appropriate XA calls that the TM-XA Service forwards to the XA resource manager. For example, when a transaction is explicitly associated with a certain thread, the TM-XA Service informs resource managers accessible from that thread of the association. When the association between a transaction and a thread ceases, TM-XA causes the resource managers to

discontinue the association between native resource manager requests and the transaction. Similarly, the TM-XA Service communicates transaction prepares, aborts, and commits to resource managers.

Enabling Tran-C applications to communicate with XA resource managers requires the following calls:

- A Tran-C application registers each resource manager it uses with the TM-XA Service using the **tmxa_RegisterRMI** function.
- The application initializes the TM-XA Service using the **tmxa_Init** function.

The **tmxa_RegisterRMI** function must be called at initialization time, exactly once for each instance of an XA resource manager used by an application. An application must issue all calls to the **tmxa_RegisterRMI** function in the same thread, and these calls must be executed before the **tmxa_Init** function is called. All registrations must precede the initialization call, because the call to **tmxa_Init** performs resource manager recovery (among other things), and it recovers only the registered resource managers. When an application restarts, it must register the same resource manager instances it registered in the previous execution. The application can also register new resource managers.

A resource manager defines the format of the string arguments to the **tmxa_RegisterRMI** function, and it also defines how to obtain the address of the XA switch structure. However, the XA specification requires the pointers to be non-NULL and the strings to be no more than 256 characters (including the terminating null byte). See the reference page for the **tmxa_RegisterRMI** function for details.

As soon as the **tmxa_RegisterRMI** function has been used to register XA-compliant resource managers, the Tran-C application can call only the **tmxa_Init** function to initialize its interactions with the TM-XA Service. This function initializes the interaction between the TM-XA service and the Distributed Transaction Service and Thread-to-Tid Mapping Service modules of the Encina Toolkit. The **tmxa_Init** function does not initiate communications with previously registered resource managers. The TM-XA Service opens each registered instance of a resource manager when the application first uses that resource manager.

Tran-C applications must arrange for the application-initialization callback to invoke the **tmxa_Init** function. Before calling **tc_InitTRPC**, the application program registers an initialization callback, using a call like the following:

```
registerApplCallback(APPL_INIT_CALLBACK, myAppl_ApplCallBack,
                     (void *) 0);
```

The **myAppl_ApplCallBack** function is defined by the application, and it initializes the TM-XA Service, as shown in Figure 64:

```
int myAppl_ApplCallBack(void * unusedArg)
{
  tmxa_status_t tmxaStatus;

  tmxaStatus = tmxa_Init();
  if (tmxaStatus != TMXA_SUCCESS){
        fprintf(stderr, "Error return from tmxa_init: %d/n",
            tmxaStatus);
        return 0;
  }
  return 1;
}
```

Figure 64. Initializing TM-XA

All calls to the **tmxa_RegisterRMI** function must be made before the call to the **tmxa_Init** function. The call to the **tmxa_Init** function must be made before any resource manager calls.

Tran-C applications that require finer control over their interaction with XA-compliant resource managers can use TM-XA functions selectively to control and interact with those services. A complete description of these functions is in "Chapter 12. The Transaction Manager-XA (TM-XA) Service" on page 169. The following is a quick summary of the TM-XA functions that allow finer control over interactions with XA-compliant resource managers:

- The **tmxa_OpenRMIs** function can be used to open a specific resource manager if some special interaction or initialization is required.
- The **tmxa_CloseRMIs** function must be used in Tran-C applications that call the TM-XA **tmxa_OpenRMIs** function in order to close connections to those services before exiting.
- The **tmxa_SetAutomaticXaAssociation** function controls whether the TM-XA Service automatically reacts to changes in a thread's transaction association. Initially, when TM-XA is notified of a change in the transaction association of that thread, it always attempts to establish or terminate associations between the transaction and resource managers for a thread. The Encina Toolkit ThreadTid module tracks and announces changes in the associations between threads and transactions.

Making servers available to clients

To provide a way for clients to locate and bind to servers in a networked computing environment, the servers can register with a globally available network service, for example, the DCE Directory Service. A directory service enables clients to do logical lookups of names that identify servers. Using a

directory service helps programmers avoid hard-wiring network addresses into their programs, because a client application can query the directory service for server location information. A directory service also provides more flexibility in starting and contacting servers on a network. A server can often be started from any network location and then subsequently register the address and port at which it can be contacted with the directory service.

Alternatively, clients and servers can communicate by using well-known endpoints. A *well-known endpoint* is a stable address and port that a server always uses to listen for RPCs. Well-known endpoints only work with servers that are always located at a unique network address and port ID that does not change.

Servers can use the TRDCE interface to publicize their locations. The **trdce_ServerRegister** function registers the server and creates binding handles through which RPC requests are received. Both dynamic and well-known endpoints are automatically registered with the RPC runtime. Server names are specified as follows:

- If the name supplied to the **trdce_ServerRegister** function is a string binding, it is considered to be a well-known endpoint, and only that binding is used.
- If the name is not a string binding, the RPC system assigns endpoints dynamically, and TRDCE registers those endpoints with the Directory Service.

A client locates a server by specifying the same name that the server used when it registered. Once a client is bound to a server, the client can make remote procedure calls directly to the server. The *OSF DCE Application Development Guide* provides more information on DCE binding handles and string bindings.

The example merchandise server registers itself after initializing the Tran-C runtime, including the RPC mechanism. The merchandise server registers with the Directory Service by calling the **trdce_ServerRegister** function in its **Initialize** function (see Figure 59 on page 268). The server name passed to the **trdce_ServerRegister** function is specified on the command line when the merchandise server is started.

Notices

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

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

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

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

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

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

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

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

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

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

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

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

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

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

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

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

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

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

Trademarks and service marks

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

Advanced Peer-to-Peer Networking
AFS
AIX
APPN
AS/400
CICS
CICS OS/2
CICS/400
CICS/6000
CICS/ESA
CICS/MVS
CICS/VSE
CICSPlex
DB2
DCE Encina Lightweight Client
DFS
Encina
IBM
IBM System Application Architecture
IMS
IMS/ESA
Informix
Language Environment
MQSeries
MVS
MVS/ESA
NetView
Open Class
OS/2
OS/390
OS/400
Parallel Sysplex
PowerPC
RACF
RAMAO
RMF
RISC System/6000
RS/6000
S/390
SAA
SecureWay
TeamConnection
Tivoli
TXSeries
VSE/ESA
VTAM
VisualAge
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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

T

U

V

W

X