

Borland VisiBroker™ 8.0 VisiTransact Guide

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file `deploy.html` for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1992–2006 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

Microsoft, the .NET logo, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

VB 80 VisiTransact Guide
April 2007

Borland®

Contents

Chapter 1		
VisiTransact basics	1	
What is VisiTransact?	1	
VisiTransact architecture	1	
VisiTransact Transaction Service	2	
Database integration (Solaris only)	2	
VisiBroker Console	3	
VisiBroker ORB	3	
VisiTransact features	3	
VisiTransact CORBA compliance	3	
Monitoring tools	3	
Minimum overhead with a light footprint	3	
Flexible deployments	4	
Support for open transaction processing standards	4	
Full support for multithreading	4	
Extensions to the OMG specification	4	
VisiTransact and the CORBAservices specification	4	
Chapter 2		
Overview of transaction processing	7	
What are transactions in a distributed environment?	7	
What is CORBA?	8	
What is the CORBA Transaction Service?	8	
Model for a basic transaction	8	
Beginning the transaction	9	
Issuing requests to transactional objects	9	
Completing a transaction	10	
Chapter 3		
A quick start with VisiTransact	13	
Overview of the example	13	
Files for the quick start example	14	
Prerequisites for running the example	14	
What you will learn in this example	15	
Writing the quick start IDL	15	
Writing the transaction originator (transfer client program).	16	
Initializing the ORB	16	
Binding to the Bank object	17	
Beginning the transaction	17	
Obtaining references to transactional objects (source and destination accounts)	18	
Invoking methods (debit() and credit()) on the transactional (account) objects.	18	
Committing or rolling back the transaction.	19	
Handling exceptions	19	
Writing the bank_server program	20	
Writing the Bank object	21	
Understanding the BankImpl class hierarchy	21	
Implementing the Bank object and its get_account() method	21	
Writing the transactional object (Account)	22	
Understanding the AccountImpl class hierarchy.	22	
Making the Account object a transactional object	23	
Implementing the Account object and its methods	23	
Building the example	24	
Selecting a Makefile	24	
Compiling the example with make	25	
Running the example	25	
Starting the Smart Agent (osagent)	25	
Starting the VisiTransact Transaction Service	25	
Starting the storage_server program	25	
Starting the bank_server program	25	
Running the Transaction Originator (transfer Client Program)	26	
Results	26	
Viewing the complete example	27	
IDL for the quick start example	27	
Transfer client program	28	
bank_server program	29	
Bank and account (transactional) objects	31	
Chapter 4		
Creating a transactional object	37	
Inheriting transactional object interfaces.	37	
Implementing transactional object interfaces	37	
Transactional POA policy interfaces	37	
OTSPolicy	37	
InvocationPolicy	38	
NonTxTargetPolicy	38	
Affected Server Behaviors	38	
Affected Client Behaviors	38	
Dealing with UNSHARED transactions	39	
Chapter 5		
Determining your approach to transactions	41	
Transaction management approaches.	41	
Direct vs. indirect context management	41	
Implicit vs. explicit propagation	42	
Context management and propagation.	42	
Indirect context management with implicit propagation	42	
Indirect context management with explicit propagation	43	
Direct context management with implicit propagation	43	
Direct context management with explicit propagation	43	
In-process vs. out-of-process VisiTransact transaction service	43	
Multithreading	44	
Integrating existing applications and transactional systems	44	
Using a combination of approaches	44	
Implementing transactions for the web	44	
Building VisiTransact applications	44	
Using stand-alone VisiTransact Transaction Service instances	45	
Embedding a VisiTransact Transaction Service instance in your application	45	
Binding to the embedded instance of the VisiTransact Transaction Service	46	

Using header files supplied with VisiTransact . . .	46	Terminator decides whether to commit or roll back	76
Chapter 6		Resource objects vote to commit the transaction	.77
Creating and propagating VisiTransact-		Summary of steps for two-phase commit77
managed transactions	49	Summary of steps for single-phase commit78
Introducing Current as used in VisiTransact-managed		Summary of steps for a rollback78
transactions	49	Participating in transaction recovery after failure.	.78
How does Current work?	50		
Obtaining a Current object reference	50	Chapter 10	
Working with the Current interface and its methods.	51	Managing heuristic decisions	81
Multiple threads participating in the same transaction	53	What is a heuristic decision?81
Using multiple transactions within a context or thread	53	What is the heuristic.log file?81
Discovering an instance of the VisiTransact		Interpreting the heuristic log.82
Transaction service	54	What to do once the problem has been isolated . .	.83
Propagating VisiTransact-managed transactions. . .	55		
Ensuring a transaction is in progress.	55	Chapter 11	
Marking a transaction for rollback	56	Implementing Synchronization objects	85
Obtaining transaction information	56	What are Synchronization objects?85
Extensions to the Current interface.	56	Using Synchronization objects before the commit	
		protocol86
Chapter 7		Using Synchronization objects after rollback or	
Other methods of creating and propagating		commit86
transactions	59	Registering Synchronization objects.86
Introduction	59	How failures affect Synchronization objects87
Creating transactions with the TransactionFactory . .	60	The role of Synchronization objects in transaction	
Gaining control of a transaction with the control object.	61	objects87
Explicitly propagating transactions from the originator	62		
Changing from explicit propagation to implicit	63	Chapter 12	
Getting the explicit context from Current	63	Backward compatibility and migration	89
Committing or rolling back transactions with Terminator	63	Backward compatibility89
Marking a transaction for rollback	65	OTS1.1 Clients vs OTS1.4 Servers89
Obtaining transaction information	65	OTS1.1 Servers vs OTS1.4 Clients89
		Migration.89
Chapter 8			
Transaction completion	67	Chapter 13	
Transaction completion	67	Session Manager overview	91
How does the VisiTransact Transaction Service		How are databases integrated into a VisiTransact	
ensure completion?	67	application?91
How does the VisiTransact Transaction Service		What is the Session Manager?92
ensure checked behavior?.	68	Opening a connection to a database.92
Heuristic completion	69	Connection profiles.93
Enabling heuristic reporting to your application . .	70	Configuring connections93
OTS exceptions	71	Associating a connection with a transaction93
		Registering Resources.93
Chapter 9		Releasing Connections94
Coordinating transaction completion with		Pooling connections94
Resource objects	73	Managing thread requirements94
Understanding transaction completion	73	Global transactions using XA protocol95
Participating in transaction completion	74	What is the XA Resource Director?95
Resource object is registered for the transaction . .	75	Distributed transaction recovery95
Transaction originator initiates transaction		DirectConnect Session Managers.96
completion	75	Registering Resources.97
Terminator tells Resource objects to prepare . . .	75	Deployment issues.97
Resource objects return a vote to the terminator . .	76	Restrictions on DirectConnect access transactions .	.97
		Coexistence: DirectConnect and XA access	
		transactions98

Chapter 14 Integrating VisiTransact with databases using the Session Manager 99

Evaluating the impact of integrating VisiTransact with databases using XA	100
Using XA adds overhead	100
Requiring high availability	100
Locked or unavailable data	100
Yielding some control	100
Evaluating the impact of integrating VisiTransact with databases using DirectConnect	101
Preparing databases	101
Connection profile sets	101
Modifying connection profiles used by Session Manager clients	102
Modifying connection profiles used by XA Resource Directors	102
Using the XA Resource Director	102
Deploying an XA Resource Director	102
Starting an XA Resource Director	102
How the XA Resource Director uses connection profiles	103
Deploying client-side libraries	103
Shutting down an XA Resource Director remotely	103
Registering the XA Resource Director with the OAD	103
Starting Session Manager-based application processes	104
Checking for the default path to persistent store files	104
Forcing heuristics	105
Performance tuning	105
For XA	105
Session Manager Configuration Server	105
Directory structure for persistent store files	105
Deploying persistent store files	107
Option 1: Persistent store files on a shared file system	107
Option 2: Persistent store files on each node	107
Option 3: Set of persistent store files copied to each node	107
Starting the Session Manager Configuration Server manually	108
Shutting down the Configuration Server	108
Security	108

Chapter 15 Data access using the Session Manager 111

Preparing for integration	111
Using the Session Manager: Summary of steps	112
Obtaining a ConnectionPool object reference	112
Using ConnectionPool object references	113
Obtaining a Connection object from the Connection Pool	113

Using explicit transaction contexts	113
Optimizing connection pooling	114
Getting a native connection handle	114
Using the native connection handle	114
Threading requirements	114
Releasing a connection	115
De-allocating the instance of Connection	115
Viewing exceptions	116
Viewing attributes	117
Obtaining Session Manager information	117
Using hold() and resume()	118
Using hold()	118
Using resume()	119
Example of a simple integration	119
XA implementation issues	120
Completing or recovering a transaction	120
DirectConnect implementation issues	120
Completing or recovering a transaction	121
Changing from DirectConnect to XA	121

Chapter 16 Pluggable Database Resource Module for VisiTransact 123

Concepts	123
What is pluggable database resource module?	123
Structural descriptions	124
Connection Management	124
Writing a Pluggable Module	127
The Connection Profiles	127
The Interface Definition	127
The Single Function	128
The ITSDataConnection class	128
Native handle acquisition interface	128
Local transaction connection and completion interface	128
Global transaction connection and completion interface	129
Building and Running	129
Running Applications using the pluggable modules	130
Programming restrictions	130
Known limitations	130

Chapter 17 Using the VisiBroker Console 133

Overview of the VisiBroker Console	133
Transaction Services section	133
Session Manager Profile Sets section	134
Starting the VisiBroker Console	134
Starting a VisiTransact Transaction Service	134
Starting the Session Manager Configuration Server	134
Launching the VisiBroker Console	134
Using the Transaction Services section	134
Locating an instance of the Transaction Service	135
Monitoring transactions	135
Refreshing the transaction list	135

Displaying details for specific transactions . . .	135	Session Manager connection profile attributes . . .	162
Controlling specific transactions.	136	Using the Session Manager with the OCI 9i API . .	162
Resolving hung or in-doubt transactions. . .	136	Programming restrictions	163
Filtering the transaction list	137	Troubleshooting	163
Viewing heuristic transactions.	137	VisiTransact message log	163
Viewing heuristic details.	137	Using the xa_trc files.	163
Viewing the message log	138	Distributed update problems	164
Filtering the message logs	138	Data access failures	164
Trimming the message log	138	Lock from in-doubt transaction	164
Using the Session Manager Profile Sets section. .	138	Transaction timeout	164
What are connection profiles?.	139	Oracle error messages.	165
Gaining access to the Session Manager		Forcing heuristic completion.	165
Configuration Server.	139		
Creating and configuring a new connection profile			
139			
Editing an existing connection profile	140		
Filtering the connection profiles	140		
Deleting a connection profile	140		
Refreshing the list of connection profiles	140		
Chapter 18		Chapter 20	
Server Application Model	143	DirectConnect Session Manager for Oracle	
Server Application transaction and database		OCI, version 9i Client	167
management	143	Overview	167
Requirements before reading this section. . . .	143	Who should read this chapter	168
Concepts and terminology.	144	Oracle9i software requirements	168
Scenarios of global transaction and PMT	145	Client requirements	168
Client-initiated global 2PC and 1PC transactions	145	Server requirements	168
Transparent server-initiated transactions with PMT		Oracle9i installation and configuration issues. . . .	168
146		Installation requirements	168
PMT overview	147	Database configuration	169
PMT transaction attribute values	148	Required environment variables.	169
A simple example	150	Session Manager connection profile attributes	169
PMT::Current and connection name	151	Using the Session Manager with the OCI 9i API	169
XA resources configuration	152	Programming restrictions	170
xa-resource-descriptor	152	Troubleshooting	170
xa-resource	152	VisiBroker VisiTransact message log	170
xa-connection	153	Oracle error messages.	171
xa-resource-alias	154		
An example of XA resource descriptor	154		
VisiTransact properties	155		
vbroker.its.its6xmode=<false true>	155		
vbroker.its.verbose=<false true>	156		
vbroker.its.xadesc=<xa-resource xml file name>	156		
vbroker.orb.enableTransactions=<false true> . .	156		
RM recovery utility	156		
Chapter 19		Chapter 21	
XA Session Manager for Oracle OCI,		Commands, utilities, arguments, and	
version 9i Client	159	environment variables	173
Overview	159	Overview of VisiTransact commands	173
Who should read this chapter	160	vbconsole	174
Oracle9i software requirements	160	Syntax.	174
Client requirements	160	Example.	174
Server requirements.	160	Arguments	174
Oracle9i installation and configuration issues	160	ots	174
Installation requirements	160	Syntax.	174
Database configuration	161	Example.	174
DBA_PENDING_TRANSACTIONS view	161	Arguments	174
Required environment variables	161	smconfig_server	175
		Syntax.	175
		Example.	175
		Arguments	175
		vshutdown	176
		Syntax.	176
		Example.	176
		Arguments	176
		xa_resdir	177
		Syntax.	177
		Example.	177
		Arguments	177
		VisiTransact utilities	177
		smconfigsetup	177
		Creating a profile for use with the Session	
		Manager.	177

Command-line arguments for applications	179
Passing command-line arguments to ORB_init() using argc and argv	179
Arguments for applications that originate transactions	179
Arguments for applications with an embedded VisiTransact Transaction Service instance . . .	180
Arguments for applications that use the Session Manager	181
Environment variables	181

Chapter 22

Error codes **183**

VisiTransact common error codes	183
VisiTransact Transaction Service error codes. . . .	184
Session Manager error codes	186
VisiTransact transaction log error codes	189

Chapter 23

Problem determination **191**

General approaches	191
Dealing with problems in transactions	191

Index **193**

1

VisiTransact basics

This section introduces VisiBroker VisiTransact, a complete transaction management solution for transactions with CORBA applications over the Internet or intranets. This chapter describes the VisiTransact features and architectural components.

What is VisiTransact?

VisiTransact provides a complete solution for distributed transactional CORBA applications. Implemented on top of the VisiBroker ORB, VisiTransact simplifies the complexity of distributed transactions by providing an essential set of services: a transaction service, recovery and logging, integration with databases and legacy systems (Solaris platform only), and administration facilities.

It provides OMG OTS 1.4 compliant transaction service functionality, the VisiTransact Transaction Service. On the Solaris platform, VisiTransact supplies an Integrated Transaction Service (ITS) which includes the XA Resource Director, Session Manager Configuration Server, Session Manager for Oracle9i, and a Pluggable Resource Interface for enabling Session Manager to work with the database of your choice.

Note

VisiTransact does not support any of our proprietary extensions. It is only compliant with OMG specification.

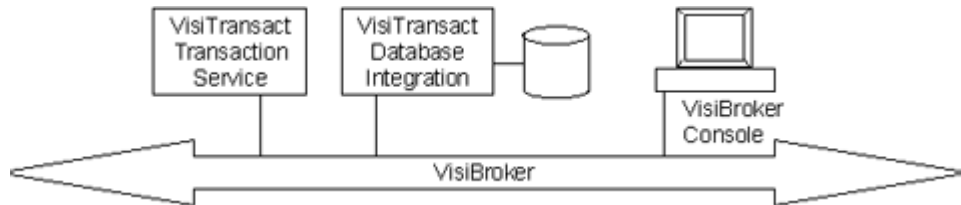
VisiTransact architecture

VisiTransact supplies the following components to provide a complete solution for the management of distributed transactions:

- VisiTransact Transaction Service
- Database integration (Solaris only)
- VisiBroker Console

As shown in the following figure, VisiTransact provides components to be used for distributed transactional CORBA applications and is implemented on top of the VisiBroker ORB.

Figure 1.1 VisiTransact architecture



VisiTransact Transaction Service

The VisiTransact Transaction Service—conforming to the final OMG Transaction Service specification version 1.4 document—manages transaction completion.

VisiTransact is provided as a shared library and an executable. This flexible architecture allows you to deploy the VisiTransact Transaction Service instance as a standalone process or embed the instance in your application. You can load balance transactions by using multiple instances of the VisiTransact Transaction Service on your network.

VisiTransact relies on the osagent (SmartAgent) to start up and to ensure there is only one instance of the Transaction Service. The `vshutdown` utility also relies on the osagent to find the Transaction Service and shut it down.

Database integration (Solaris only)

The database integration components help you integrate transactional applications on Solaris platforms with databases and other Resource Managers. The following components are included for database integration:

- **Session Manager, XA Implementation (Oracle9i only).** The Session Manager XA implementation allows an application to obtain a VisiTransact-enabled connection to an Oracle9i database. The Session Manager handles all XA calls, and enables the VisiTransact Transaction Service to coordinate transactions across Resources. The Session Manager also provides database connection pooling. Additionally, the Session Manager Configuration Server enables you to create connection profiles using the VisiBroker Console.
- **Session Manager, DirectConnect Implementation (Oracle9i only).** The DirectConnect implementation of the Session Manager provides non-XA access to Resources. It consists of a single application server process that contains a Session Manager with embedded single-phase resources. This architecture provides improved performance because it performs a single-phase commit.
- **Session Manager Configuration Server.** The Session Manager Configuration Server enables you to create Session Manager connection profiles using the VisiBroker Console. For more information about the Session Manager Configuration Server, see [“Session Manager Configuration Server”](#).
- **Pluggable Resource Interface.** The Pluggable Resource Interface gives you the capability to enable Session Manager to work with the database of your choice. It is a component that implements a set of predefined interfaces to allow transactional applications to use databases other than Oracle9i as their persistent storage in transactions managed by VisiTransact. For more information about the Pluggable Resource Interface, see [“Pluggable Database Resource Module for VisiTransact.”](#)
- **XA Resource Director.** The XA Resource Director manages all interactions with a particular Resource Manager that is participating in one or more transactions. It handles all transactions with a specific Resource Manager (for example an Oracle9i database) on the network. The XA Resource Director *bridges* the VisiTransact and X/

Open transaction environments, which allows for interoperability between the VisiTransact Resource model and the X/Open Distributed Transaction Protocol (DTP) Resource Manager model. For more information about the VisiTransact Resource model, see [“What is the XA Resource Director?”](#).

VisiBroker Console

The VisiBroker Console is a graphical tool used in managing distributed transactions over the network and configuring connection profiles for use with specific databases. It can be used to monitor and control the status and completion of transactions. Using the Console you can create Session Manager connection profiles with the Session Manager Configuration Server. For more information about the VisiTransact section of the VisiBroker Console, see [“Using the VisiBroker Console.”](#)

VisiBroker ORB

The VisiBroker ORB provides the functionality and benefits of its implementation to applications using VisiTransact for distributed transaction management. The ORB provides many features to VisiTransact applications, including: thread-pooling, multiplexed and recycled connections, load balancing, and fault tolerance. Many of these features are typically part of a transaction processing monitor.

VisiTransact uses OMG portable interceptors, a powerful feature of the VisiBroker ORB, to implement its functionality. VisiTransact users can also take advantage of interceptors to customize transactional applications.

VisiTransact features

VisiTransact manages the completion of flat transactions using a one-phase or two-phase commit protocol, as appropriate. If there is only one resource involved with a transaction, the one-phase commit protocol will be used.

In addition to the distributed transaction management features described in CORBA's Transaction Service specification, VisiTransact also provides extensions to the specification. These extensions and other features are described in the following sections.

VisiTransact CORBA compliance

VisiTransact is fully compliant with the CORBA 3.0 specification from the Object Management Group (OMG). For more details, refer to the CORBA specification located at <http://www.omg.org>.

VisiTransact is also compliant with the CORBAservices specification for the transaction service version 1.4 from the OMG. [“VisiTransact and the CORBAservices specification”](#) lists decisions made by VisiTransact for several options accorded by the specification.

Monitoring tools

Using the Console you can monitor and control the status and completion of transactions, as well as manage the size and location of log files.

Minimum overhead with a light footprint

Depending on your system requirements, you can have as many VisiTransact Transaction Service instances as you need on your network. The VisiTransact Transaction Service does not have to reside on every host machine in your environment.

The Session Manager (available on Solaris only) also saves system resources by pooling database connections, and reusing connections across requests.

Flexible deployments

VisiTransact optimizes deployment by providing you with three choices:

- Linking your application/business objects in directly with the VisiTransact Transaction Service.
- Deploying your application/business objects on the same machine with the VisiTransact Transaction Service.
- Deploying your application/business objects on any machine regardless of the location of the VisiTransact Transaction Service.

For scalability and fault tolerance, you can deploy multiple instances of business objects and multiple instances of the VisiTransact Transaction Service on multiple machines.

On Solaris platforms, if you have a single Oracle9i database, you can achieve an even greater performance gain by linking the following into a single process:

- Application code
- VisiTransact Transaction Service
- Session Manager (for Oracle9i databases on Solaris platforms only)


Support for open transaction processing standards

Currently, VisiTransact supports the OMG's CORBAservices Transaction Service and the XA protocol open transaction processing standards.

Full support for multithreading

Because VisiTransact supports multithreading, your business object can be multithreaded and, therefore, it can handle multiple requests simultaneously.

Extensions to the OMG specification

VisiTransact provides several extensions to the OMG CORBA services specification to simplify development. For example, VisiTransact extends the current interface to provide the `begin_with_name()` method that allows you to assign a user-defined name to a transaction. These supplemental methods are designated in Transaction Service interfaces and classes with the  icon.

VisiTransact and the CORBAservices specification

See the table below for information about how VisiTransact implements certain options of the CORBAservices specification.

Option	VisiTransact Decision
An implementation of the Transaction Service is not required to support synchronization.	VisiTransact fully supports Synchronization objects (the Synchronization interface).
The Unavailable exception is raised if the Transaction Service implementation chooses to restrict the availability of the transaction context.	VisiTransact will not raise the Unavailable exception because it does not restrict the availability of the transaction context.

Option	VisiTransact Decision
An implementation of the Transaction Service is not required to initialize the transaction context of every request handler.	VisiTransact default behavior is only to initialize the transaction context if the interface supported by the target object is derived from the <code>TransactionalObject</code> interface. VisiTransact can be configured to initialize the transaction context of all requests. See “Creating and propagating VisiTransact-managed transactions.”
An implementation of the Transaction Service may restrict the ability for some or all of these (Coordinator, Terminator and Control) objects to be transmitted to or used in other execution environments, to enable it to guarantee transaction integrity.	VisiTransact does not impose any restrictions on the ability of the Coordinator, Terminator or Control objects to be transmitted to or used in other execution environments. See “How does the VisiTransact Transaction Service ensure completion?” for discussion of how to obtain checked behavior.
It is implementation-specific whether the Transaction Service itself monitors the participants in a transaction for failures or inactivity. Some implementations of the Transaction Service impose constraints on the use of the Transaction Service interfaces in order to guarantee integrity equivalent to that provided by the interfaces which support the X.Open DTP transaction model. This is called checked behavior.	VisiTransact does not impose constraints, but supports checked behavior in VisiTransact-managed transactions as described in “How does the VisiTransact Transaction Service ensure checked behavior?”
Some implementations of the Transaction Service may allow transactions to be terminated by Transaction Service clients other than the one which created the transaction.	VisiTransact allows termination of a transaction by any object that uses the <code>Terminator</code> interface for the transaction (for example, non VisiTransact-managed transactions). However, VisiTransact restricts termination of a transaction when using the <code>Current</code> interface to the client thread that created the transaction.
A TransactionFactory is located using the FactoryFinder interface of the life cycle service and not by the <code>resolve_initial_references()</code> operation on the ORB.	Locate a VisiTransact TransactionFactory using the VisiBroker ORB discovery facilities, such as the <code>bind()</code> method.
A Transaction Service implementation may optionally use the Event Service to report heuristic decisions.	VisiTransact does not use the Event Service to report heuristic decisions.
An implementation of the Transaction Service is not required to support nested transactions.	No major databases support nested transactions at this time. Therefore, VisiTransact does not support nested transactions.

2

Overview of transaction processing

This section provides an overview of transactions and how they are processed. It explains transactions, CORBA, the components of the CORBA Transaction Service, and the process for a basic transaction.

What are transactions in a distributed environment?

In a distributed objects world, a transaction is a unit of work composed of a set of operations on objects. A familiar example is transferring money from one bank account to another. The transfer is two separate actions—a debit from one account, and a credit to another—that comprise a single transaction.

Figure 2.1 Example of a transaction



In this scenario, which implements a flat transaction model, both actions must be completed for the desired result to be achieved. If Action 1 is completed, but Action 2 is not, the customer loses money. If Action 1 is not completed, but Action 2 is, the bank loses money. Therefore, flat transactions are an all-or-nothing proposition—either all steps of a transaction must complete, or none of the steps must complete.

Note

There is another type of transaction - a nested transaction - that does not require that all steps of a transaction are completed. However, VisiTransact Transaction Manager does not support nested transactions.

Many things can happen to prevent all steps of a transaction from completing such as application logic, server failure, hardware failure, and network interruptions. Because of these unpredictable environment factors, transactions must adhere to the following properties, called ACID properties, to ensure the consistency, reliability, and integrity of applications:

- **Atomicity.** If a transaction is completed successfully (it *commits*) all of the actions associated with the transaction are performed. Otherwise, if the transaction is not completed successfully, none of the actions are performed and the transaction is *rolled back*.
- **Consistency.** All actions that comprise a transaction must be performed accurately so that the system moves from one consistent state to another. In the bank example, this means that the total money in both accounts before the transaction begins is the same as the total money in both accounts after the transaction completes.
- **Isolation.** This means that intermediate results performed by a transaction are not visible outside the transaction until the entire transaction completes.
- **Durability.** The results of a transaction are persistent.

Transactions do not always involve the transfer of funds as in the banking example we have just covered. Transactions are necessary for all sorts of business activities. For example, an online bookstore needs transactions to perform many activities including: ordering books from suppliers, transferring inventory from suppliers, updating available quantities of books accurately, charging customers appropriately for purchases, and fulfilling customer orders. All of these actions, and a multitude of others, may need to be executed within a transaction.

What is CORBA?

The Common Object Request Broker Architecture (CORBA) specification was adopted by the Object Management Group (OMG) to ensure a common approach to implementing and managing distributed objects. CORBA uses an object-oriented approach for creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well defined interface, which reduces application complexity. The cost of developing applications is also reduced, because once an object is implemented and tested, it can be used over and over again.

What is the CORBA Transaction Service?

The CORBA Transaction Service, defined by the OMG, enables mission-critical applications in distributed environments by providing transactional integrity. It defines IDL interfaces that allow multiple distributed objects to participate in a transaction, and enable a distributed application to handle transaction completion over the Internet and intranets.

Borland's implementation of the CORBA Transaction Service is embodied in the VisiTransact Transaction Service, a component of the Transaction Management architecture.

Model for a basic transaction

The VisiTransact Transaction Service can be used to *manage* the completion of a transaction. It works with objects at the ORB level to coordinate and manage a transaction's commit or rollback. The ORB enables the VisiTransact Transaction Service to propagate the *transaction context* to each object participating in the transaction. To accomplish this, the VisiTransact Transaction Service interfaces with participants of the transaction at specific points in the transaction management process.

In a distributed application, a transaction can involve multiple objects performing multiple requests. The objects that are involved can play a number of different roles.

For example, an object that begins a transaction is called the *transaction originator*. The following table provides descriptions of these roles.

Role of Participant	Description
Transactional Client	A transactional client is the user's interface to a transactional application. Sometimes the transactional client is also the transaction originator.
Transaction Originator	A transaction originator is the object that begins a transaction. A transaction originator is not necessarily a transactional client—it might be a transactional server that originates a transaction.
Transactional Object	A transactional object is an object whose behavior is affected by the transaction, but has no recoverable state of its own. Although a transactional object does not participate in the completion of a transaction, it can force the transaction to rollback. See “Coordinating transaction completion with Resource objects” for information about recoverable objects, or objects whose recoverable state is affected by the transaction.
Transactional Server	A transactional server is a collection of one or more transactional objects.

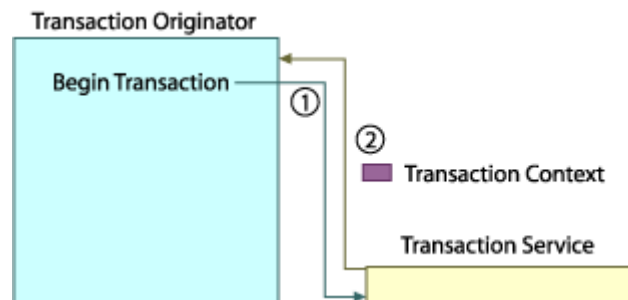
The VisiTransact Transaction Service interacts with an application when the transaction originator begins the transaction, as transactional information is propagated to transactional objects, and finally, coordinates the transaction's completion (commit or rollback) across multiple objects.

Although this chapter does not address it, most transactions involve persistent data (such as databases). For these types of transactions, there are two additional participant roles—Resource and Recoverable Server. These roles are discussed in [“Coordinating transaction completion with Resource objects.”](#)

Beginning the transaction

When an object initiates a transaction, an instance of the VisiTransact Transaction Service begins a transaction for a transaction originator and establishes a *transaction context*. The transaction context is then associated with the originator's thread of control that was issued by the VisiBroker ORB. The transaction context contains transaction information, including an object transaction identifier (OTRID) that uniquely identifies the transaction.

Figure 2.2 Beginning the transaction



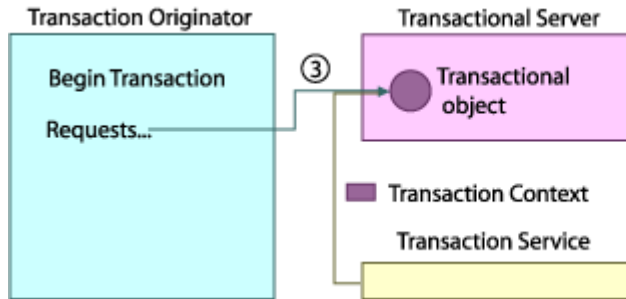
In step 1 of the figure above, the transaction originator registers with the VisiTransact Transaction Service its desire to begin a transaction. The VisiTransact Transaction Service answers this request with step 2 by returning a transaction context to the transaction originator.

Issuing requests to transactional objects

As the transaction originator issues requests to transactional objects in Step 3, each of these requests is also associated with the transaction context. Using the ORB, the

VisiTransact Transaction Service propagates the transaction context to all objects participating in the transaction.

Figure 2.3 Issuing requests to transactional objects and passing the transaction context



Note

The transaction context is passed as a Service Context in the GIOP request and response headers. This makes its propagation completely transparent, and is compliant with the CORBA services specification for interoperability between Transaction Service implementations.

Completing a transaction

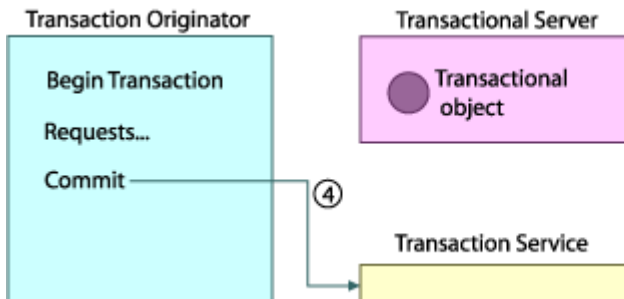
A transaction can be completed in the following ways:

- The transaction originator commits the transaction. This is the typical scenario.
- Any component in the application, as long as Current is not used, can complete the transaction.
- The transaction times out.

If a commit is requested and all participating Resources agree to commit, then the changes are committed. If any participant votes for rollback, then the transaction is rolled back.

If completion is not requested by the application, the VisiTransact Transaction Service will rollback the transaction when the timeout period expires.

Figure 2.4 Completing the transaction



3

A quick start with VisiTransact

This section describes the development of distributed, object-based transactional applications with VisiTransact using a sample C++ application.

For **Java example**, refer to `\\VisiBroker\examples\vbroker\its` folder.

Overview of the example

The scenario in this C++ example involves a bank that has several accounts. During a transaction, money is transferred between at least two of these accounts—depending upon the parameters passed to the client program.

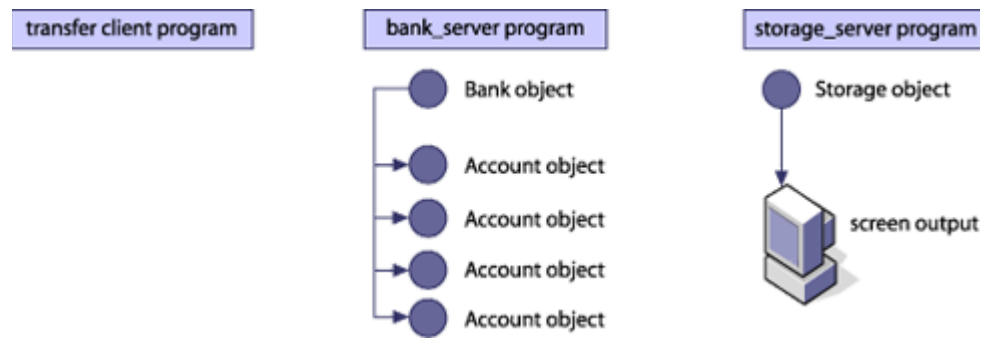
The programs for the quick start are:

- **transfer**. This program takes input from the command line about how much money should be transferred between which accounts. It then begins the transaction and performs the requested transfer. After all requested transfers have completed, it requests to complete the transaction (either commit or rollback).
- **bank_server**. This program binds to a Storage object and creates a Bank object with the name entered at the command line.
- **storage_server**. This program implements a Storage object for the non-database quick start, ensures that changes made during the transaction to balances are stored persistently (if committed), or that the account balances are returned to their state before the transaction (if rolled back).

The objects for the quick start are:

- **Bank**. This object provides access to existing Account objects. It creates instances of the Account object for accounts that exist in the Storage object.
- **Account**. This object lets you view the balance for an account, and credit or debit an account's balance. It uses the Storage object to interact with persistent data.
- **Storage**. The purpose of this object is to abstract data access into one object that makes changes to data on behalf of the accounts.
- **StorageServerImpl**. This implementation of the Storage object contains a lightweight Resource (FakeResourceImpl) that simply updates balances in memory. It is only provided to help you get up and running quickly with VisiTransact.

Figure 3.1 Components of the quick start example



Files for the quick start example

If you do not know the location of the VisiTransact package, see your system administrator. The files listed in the following table are included for the example.

File	Description
quickstart.idl	The IDL for the quickstart example that defines the required interfaces for objects.
transfer.C	The client program that gathers input from the user, and is the originator of a VisiTransact-managed transaction that invokes transactional server objects as part of the transaction.
storage_server.C	The server program that creates Storage objects. In this example, the Storage object is a simple implementation that updates balances in memory and outputs them. The storage_server is only provided to get you up and running quickly.
storage_server.h	Contains the specification of the Storage object.
bank_server.C	The server program that creates the Bank object with information from the storage_server or storage_ora program, and makes it available to client programs.
bank.h	The specification of the Bank and Account objects.
bank.C	Contains the implementations of the Bank and Account interfaces. The Bank object creates transactional objects (Account objects). The Account object is the transactional object that calls on the Storage object to credit or debit account balances.
Makefile	Used to build all the test targets.
Makefile.cpp	Used to build all the test targets.
../itsmk	Specific make definitions for platforms supported

Note

To aid in portability, the example files use the .C extension on both Windows and UNIX so that there can be a common Makefile.

Prerequisites for running the example

You must install the VisiTransact product, and the VisiBroker C++ Developer (ORB). You must also start an instance of the VisiTransact Transaction Service, as described in ["Running the example"](#).

What you will learn in this example

Here are the steps to implement the quick start:

- 1 Implement a simple interface in IDL that defines the three objects (Bank, Account, and Storage) required for the transactional application. See [“Writing the quick start IDL”](#).
- 2 Implement the client program and transaction originator (transfer). Gather input from the user about which accounts to use, and how much money to transfer; initialize the ORB, begin a transaction, bind to a Bank object, obtain a reference to a transactional object (Account), perform actions with the transactional object (Account), commit or rollback the transaction, and handle exceptions. See [“Writing the transaction originator \(transfer client program\)”](#).
- 3 Implement the **bank_server** program. Initialize the ORB, create a Bank and access a Storage object, register the Bank object with the POA, and prepare to receive requests. See [“Writing the bank_server program”](#).
- 4 Implement the Bank. Instantiate and return a transactional object (Account) upon request. See [“Writing the Bank object”](#).
- 5 Implement a transactional object (Account). Handle requests to view account balances, and credit or debit account balances. See [“Writing the transactional object \(Account\)”](#).
- 6 Implement a Storage object. Access and update data as requested by business (Account) objects.
- 7 Build the example. To create the client program, we must compile and link the client program code with the client stub. To create the server programs, we must compile and link the server code with the client and server skeletons. See [“Building the example”](#).
- 8 Run the example. Start the Smart Agent, the VisiTransact Transaction Service, server programs, and the client program. See [“Running the example”](#).

Writing the quick start IDL

The first step to creating a transactional application with VisiTransact is to specify all of your interfaces using the CORBA Interface Definition language (IDL). IDL is language-independent and has a syntax similar to C++, but can be mapped to a variety of programming languages.

The example below shows the contents of the **quickstart.idl** file which defines the three objects required for the quick start example—Bank, Account, and Storage.

```
// quickstart.idl
#include "CosTransactions.idl"
#pragma prefix "visigenic.com"

module quickstart
{
    //requires
    interface Account
    {
        float balance();
        void credit(in float amount);
        void debit(in float amount);
    };

    exception NoSuchAccount
    {
        string account_name;
    };

    interface Bank
```

Writing the transaction originator (transfer client program)

```
{
    Account get_account(in string account_name)
        raises(NoSuchAccount);
};

typedef sequence<string> AccountNames;

//adapts
interface Storage
{
    float balance(in string account)
        raises(NoSuchAccount);
    void credit(in string account, in float amount)
        raises(NoSuchAccount);
    void debit(in string account, in float amount)
        raises(NoSuchAccount);
    AccountNames account_names();
};
};
```

The interface specification you create in IDL is used by the VisiBroker ORB's `idl2cpp` compiler to generate C++ stub routines for the client application, and skeleton code for the objects. The stub routines are used by the client program for all method invocations. You use the skeleton code, along with code you write, to create the server programs that implement the objects.

The code for the client and servers, once completed, is used as input to your C++ compiler and linker to produce your client and server programs.

Writing the transaction originator (transfer client program)

The file named **transfer.C** contains the implementation of the transaction originator, which also happens to be the client program. As discussed in [“Overview of transaction processing,”](#) the transaction originator is not always the client program. The **transfer** client program performs a single VisiTransact-managed transaction (see [“Creating and propagating VisiTransact-managed transactions”](#) for details). For information on how to manage transactions in other ways, see [“Other methods of creating and propagating transactions.”](#)

The client program performs these steps:

- 1 Initializes the ORB.
- 2 Binds to the Bank object named at the command line.
- 3 Begins a transaction.
- 4 Obtains a reference to the transactional objects (the source and destination Account objects) named at the command line.
- 5 Invokes the `debit()` and `credit()` methods on the Account objects for each set of source/destination/amount entries to the **transfer** client program. It prints out the current balances for each Account before and after the transfer.
- 6 Commits or rolls back the transaction.
- 7 Handles exceptions.

Initializing the ORB

The first task that your transaction originator needs to do is initialize the ORB, as shown in the example below. As a component of VisiBroker, command-line arguments for VisiTransact are supplied to VisiTransact through the VisiBroker ORB initialization call `ORB_init()`. Therefore, in order for arguments specified on the command line to have effect on the VisiTransact operation in a given application process, applications must pass the original `argc` and `argv` arguments to `ORB_init()` from the main program.

```
...
int main(int argc, char* const* argv)
```



```

{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        ...

```

The `ORB_init()` function will parse both ORB arguments and VisiTransact arguments, removing them from the `argv` vector before returning.

In Java

The initialization of the ORB is done using the property `vbroker.orb.dynamicLibs=com.borland.vbroker.CosTransactions.implicit.Init` or by setting `vbroker.orb.enableTransactions=true`.

Binding to the Bank object

Before the **transfer** client program can invoke methods on the transactional (Account) objects, it must first use the `_bind()` method to establish a connection to the Bank object. The implementation of the `_bind()` method is generated automatically by the `idl2cpp` compiler. The `_bind()` method requests the ORB to locate and establish a connection to the Bank object.

The following example shows how to bind to the Bank object as specified in the `bank_name` parameter passed at the command line when the **transfer** client program is started. Notice how a `_var` is used to facilitate memory management.

```

const char *bank_name = argv[1];
//Locate the bank.
Quickstart::Bank_var bank;
//Get the Bank ID
PortableServer::ObjectId_var bankId =
    PortableServer::string_to_ObjectId(bank_name);
try
{
    bank = quickstart::Bank::_bind("/bank_agent_poa", bankId);
    //bank = quickstart::Bank::_bind(bank_name);
}
catch (CORBA::Exception &ex)
{
    const char *name;
    (bank_name == 0) ? name="NULL" : name=bank_name;
    cerr << "Unable to bind to Bank \"" << name << "\": " << ex << endl;
    return 1;
}

```

Beginning the transaction

Before beginning a transaction, you must obtain a transaction context. VisiTransact-managed transactions are handled transparently to your application with `Current`—an object which maintains a unique transaction for each active thread. To use a VisiTransact-managed transaction, you must obtain a reference to this `Current` object. The `Current` object is valid for the entire process under which you create it, and can be used in any thread.

The next example shows how to obtain a VisiTransact-managed transaction. First an object reference is obtained for the `TransactionCurrent` object using the `CORBA::ORB::resolve_initial_references()` method. The `Current` object returned from this method is then narrowed to the specific `CosTransactions::Current` object using the `narrow()` method. See the `VisiBroker` documentation for a full description of the `resolve_initial_references()` and `narrow()` methods.

```

// Start a transaction.
CosTransactions::Current_var current;
{
    CORBA::Object_var initRef =
        orb->resolve_initial_references("TransactionCurrent");
    current = CosTransactions::Current::_narrow(initRef);
}

```

```
}  
...
```

To perform work that is managed by VisiTransact, you must first begin a transaction using the `Current` interface's `begin()` method. Only one transaction can be active within a thread at a time. The following example shows how to begin a VisiTransact-managed transaction.

```
...  
CosTransactions::Current_var current;  
...  
current->begin();  
...
```

Obtaining references to transactional objects (source and destination accounts)

Once you bind to the `Bank` object, you can obtain a reference to the transactional (`Account`) objects specified when the **transfer** program is started. Within the **transfer** program, these references are obtained using the `get_account()` method in the `Bank` interface. The example below shows the relevant code from the **transfer** program.

```
...  
try  
{  
  for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)  
  {  
    const char* srcName = argv[i];  
    const char* dstName = argv[i + 1];  
    float amount = (float)atof(argv[i + 2]);  
  
    quickstart::Account_var src = bank->get_account(srcName);  
    quickstart::Account_var dst = bank->get_account(dstName);  
    ...  
  }  
}  
catch(const quickstart::NoSuchAccount& e)  
{  
  cout << "Exception: " << e << endl;  
  commit = 0;  
}  
catch(const CORBA::SystemException& e)  
{  
  cout << "Exception: " << e << endl;  
  commit = 0;  
}  
...
```

In the above example, the **transfer** client program loops through its input arguments (received at the command line when the program started), and calls `get_account()` for each source and destination account name entered. If the account name entered is valid, the `Bank` object returns a corresponding `Account` object. See [“Implementing the Bank object and its `get_account\(\)` method”](#) for details on the `Bank` object's `get_account()` method.

Notice that if an invalid account name was entered, an error message is printed and the value of the `commit` variable is set to false. Likewise, if a system exception is raised when performing the invocation of `get_account()`, an error message is printed and the value of the `commit` variable is set to false. See [“Committing or rolling back the transaction”](#) to find out how the `commit` variable is used for transaction completion.

Invoking methods (`debit()` and `credit()`) on the transactional (account) objects

Once the **transfer** client program has established a connection with the source and destination `Account` objects, the `debit()` and `credit()` methods of the `Account` interface can be invoked for each source/destination/amount triplet that was entered when the **transfer** program was started.

The `debit()` and `credit()` methods are invoked from within the **transfer** program's main `try()` clause using the information returned to the `src` and `dst` variables by the invocation of the `get_account()` method shown in the previous example. The next example shows the parts of the `try()` clause that invoke `credit()` and `debit()`.

```
try
{
    for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)
    {
        ...
        src->debit(amount);
        dst->credit(amount);
        ...
    }
}
...
```

Committing or rolling back the transaction

Once a transaction has begun, it must be committed or rolled back to complete the transaction. If an originator of a VisiTransact-managed transaction does not complete the transaction, the VisiTransact Transaction Service will rollback the transaction after a timeout period. However, it is important to commit or rollback transactions so that hung transactions do not consume system resources.

The example below shows how the **transfer** program uses the `commit` variable to decide whether to commit or rollback the transaction. If the `commit` variable is 1 (true), the transaction is committed. If the `commit` variable is 0 (false), the transaction is rolled back. In the next example, the 0 sent to `commit()` means that heuristics will not be reported. See [“Transaction completion”](#) for information about heuristics.

```
...
CORBA::Boolean commit = 1;
...
if(commit)
{
    cout << "**** Committing transaction ****" << endl;
    current->commit(0);
}
else
{
    cout << "**** Rolling back transaction ****" << endl;
    current->rollback();
}
...
```

Handling exceptions

The following example shows the outer `try` and `catch` statements for the **transfer** client program. Notice how these statements are used to detect any failures (CORBA or application exceptions), print a message, and return.

```
try
{
    ...
}
catch(const CORBA::Exception& e)
{
    cerr << "Exception: " << e << endl;
    return 1;
}
catch(...)
{
    cerr << "Unknown Exception caught" << endl;
    return 1;
}
return 0;
...
```

Writing the bank_server program

The **bank_server** program performs these steps in the `main` routine:

- 1 Initializes the ORB.
- 2 Obtains a Storage object and instantiates a Bank object with it.
- 3 Registers the Bank object with the ORB and POA.
- 4 Enters a loop waiting for client requests.

The `argc` and `argv` parameters passed to the `ORB_init()` methods are the same parameters that are passed to the `main` routine. These parameters can be used to specify options for the ORB.

```
int main(int argc, char* const* argv)
{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        ...
    }
}
```

Next, the **myPOA** that is to be used to activate the Storage object is created. The **bank_server** program then obtains a Storage object, and retrieves account information from it. Using the account information, the **bank_server** program instantiates the Bank object. Lastly, the **bank_server** program calls the `orb->run()` method to start the event loop that receive client requests.

```
const char* bank_name = argv[1];
// get a reference to the root POA
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
    PortableServer::PERSISTENT);

// get the POA Manager
PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

// Create myPOA with the right policies
PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
    poa_manager,
    policies);

// Get the Bank Id
PortableServer::ObjectId_var bankId =
    PortableServer::string_to_ObjectId(bank_name);

// Get a storage object for the bank.
quickstart::Storage_var storage = quickstart::Storage::_bind("/
    bank_storage_poa", bankId);

// Create the bank servant
PortableServer::ServantBase_var bankServant = new BankImpl(bank_name, storage, orb);

// Decide on the ID for the servant
PortableServer::ObjectId_var managerId =
    PortableServer::string_to_ObjectId(bank_name);

// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(managerId, bankServant);

// Activate the POA Manager
poa_manager->activate();

CORBA::Object_var reference = myPOA->servant_to_reference(bankServant);
cout << reference << " is ready" << endl;

// Wait for incoming requests
orb->run();
```

Writing the Bank object

There are a few tasks you must do to implement the Bank object:

- Derive the `BankImpl` class from the `POA_quickstart::Bank` skeleton class.
- Implement the Bank object that produces the transactional (`Account`) objects.

Understanding the `BankImpl` class hierarchy

The `BankImpl` class that you implement is derived from the `POA_quickstart::Bank` class that was generated by the `idl2cpp` compiler. The following example shows the `BankImpl` class.

```
class BankImpl : public POA_quickstart::Bank
{
private:
    quickstart::AccountNames_var _account_names;
    quickstart::Storage_var _storage;
    AccountRegistry _accounts;
    PortableServer::POA_var _account_poa;
public:
    BankImpl(const char* bank_name,
             quickstart::Storage* storage, CORBA::ORB* orb);
    virtual ~BankImpl();
    virtual quickstart::Account* get_account(const char* account_name);
};
```

Implementing the Bank object and its `get_account()` method

The `BankImpl` interface defines its constructor and destructor. The constructor creates a **Bank object with the name provided when the `bank_server` program is started (`bank_name`). It also creates an instance of `AccountRegistry` which is used to keep trace of all instantiated `Account` objects. The account names are obtained from the `Storage` object.**

```
BankImpl::BankImpl(const char* bank_name,
                  quickstart::Storage* storage, CORBA::ORB* orb)
{
    _account_names = storage->account_names();
    _storage = quickstart::Storage::_duplicate(storage);

    PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow(orb->resolve_initial_references("RootPOA"));
    CORBA::PolicyList policies;
    policies.length(2);
    CORBA::Any policy_value;
    policy_value <<= CosTransactions::REQUIRES;
    policies[0] = orb->create_policy(CosTransactions::OTS_POLICY_TYPE,
                                   policy_value);
    policies[1] =
        root_poa->create_implicit_activation_policy(PortableServer::IMPLICIT_ACTIVATION);
    _account_poa = root_poa->create_POA("account_poa",
                                       PortableServer::POAManager::_nil(),
                                       policies);
    _account_poa->the_POAManager()->activate();
    return;
}

BankImpl::~BankImpl()
{
}
```

The next example shows the **Bank object's `get_account()` method. +Note that the `get_account()` method performs a check to see if the account exists, else it will create a new account. If it does not, a `NoSuchAccount` exception is thrown.**

```
quickstart::Account_ptr
BankImpl::get_account(const char* account_name)
{
```

Writing the transactional object (Account)

```
// Lookup the account in the account dictionary.
PortableServer::ServantBase_var servant = _accounts.get(account_name);
CORBA::Boolean foundAccount = 0;

if (servant == PortableServer::ServantBase::_nil()) {
    for(CORBA::ULong i = 0; !foundAccount && i < _account_names->length(); i++) {
        if (!strcmp(_account_names[i], account_name)) {
            servant = new AccountImpl(account_name, _storage);

            // Print out the new account
            cout << "Created " << account_name << "'s account." << endl;

            // Save the account in the account dictionary.
            _accounts.put(account_name, servant);
            foundAccount = 1;
        }
    }
    if (!foundAccount) {
        throw quickstart::NoSuchAccount(account_name);
        return 0;
    }
}

try {
    CORBA::Object_var ref = _account_poa->servant_to_reference(servant);
    quickstart::Account_var account = quickstart::Account::_narrow(ref);
    cout << "account generated." << endl;
    return quickstart::Account::_duplicate(account);
}
catch(const CORBA::Exception& e) {
    cerr << "_narrow caught exception: " << e << endl;
    return quickstart::Account::_nil();
}
throw quickstart::NoSuchAccount(account_name);
return 0;
}
```

Writing the transactional object (Account)

There are a few tasks you must complete to implement the transactional (Account) object:

- Derive the `AccountImpl` class from the `POA_quickstart::Account` class.
- Implement the `Account` object with implementations for the `balance()`, `credit()`, and `debit()` methods that invoke the `Storage` object.

Understanding the `AccountImpl` class hierarchy

The `AccountImpl` class that you implement is derived from the `POA_quickstart::Account` class that was generated by the `idl2cpp` compiler. Refer to the first code example in the previous section. The `account_poa` has a policy `OTS_POLICY_TYPE` of `REQUIRE` defined, hence all objects that are activated on this poa will need to be transactional objects. Hence, `account`.

```
class AccountImpl : public POA_quickstart::Account
{
private:
    CORBA::String_var _account_name;
    quickstart::Storage_var _storage;
public:
    AccountImpl(const char* account_name,
                quickstart::Storage* storage);
    virtual CORBA::Float balance();
    virtual void credit(CORBA::Float amount);
    virtual void debit(CORBA::Float amount);
private:
    virtual void markForRollback();
};
```

Making the Account object a transactional object

To make an object transactional two things must be done:

- Create a poa with OTS_POLICY_TYPE with values REQUIRE or ADAPT.
- Use the poa to activate the object

The `_account_poa` was created during the construction of the `BankImpl` object. Refer to the first code sample in [“Implementing the Bank object and its `get_account\(\)` method”](#). In the `get_account()` function, whenever a new account is needed it will be activated using the `_account_poa`. This makes the `Account` object a transactional object.

Implementing the Account object and its methods

As shown in the following example, the `AccountImpl` class defines its constructor which creates an `Account` object with the `account_name` and `storage` parameters provided by the `Bank` object.

```
AccountImpl::AccountImpl(const char* account_name,
    quickstart::Storage* storage)
{
    _account_name = CORBA::strdup(account_name);
    _storage = quickstart::Storage::duplicate(storage);
}
```

As shown in the next example, the `Account` class also implements a `markForRollback()` method. When invoked, this method calls `rollback_only()` to force the transaction originator to rollback the transaction.

```
void AccountImpl::markForRollback()
{
    try
    {
        CORBA::ORB_var orb = CORBA::ORB_init();
        CORBA::Object_var initRef =
            orb->resolve_initial_references("TransactionCurrent");
        CosTransactions::Current_var current =
            CosTransactions::Current::_narrow(initRef);
        current->rollback_only();
    }
    catch(const CosTransactions::NoTransaction&)
    {
        throw CORBA::TRANSACTION_REQUIRED();
    }
}
```

Notice how the `markForRollback()` method obtains a handle to the `TransactionCurrent` object, and then narrows to a `Current` object so that it can call `current->rollback_only()`. Since the `Account` object is not the transaction originator, it cannot invoke `rollback()`—with `VisiTransact`-managed transactions, only the transaction originator can complete the transaction.

As shown in the next example, the `Account` object also implements the `balance()`, `credit()`, and `debit()` methods:

- The `balance()` method requests the current balance for the `Account` object from the `Storage` object.
- The `credit()` method requests that the `Storage` object increment the balance by the `amount` parameter.
- The `debit()` method requests that the `Storage` object decrease the balance by the `amount` parameter.

Note

Although the `Account` object for the quick start example could easily interact with the database itself, the example is architected to mirror real-world scenarios where a back-end data access object is used by multiple business logic objects. This makes it easy to change your database in the future, if it is necessary to do so.

```
CORBA::Float AccountImpl::balance()
{
    try
    {
        return _storage->balance(_account_name);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cerr << "Account::balance: " << e << endl;
        markForRollback();
        return 0;
    }
}

void

AccountImpl::credit(CORBA::Float amount)
{
    if(amount < 0)
    {
        cerr << "Account::credit: Invalid amount: " << amount << endl;
        markForRollback();
    }
    try
    {
        _storage->credit(_account_name, amount);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cerr << "Account::credit: " << e << endl;
        markForRollback();
    }
}

void
AccountImpl::debit(CORBA::Float amount)
{
    if(amount < 0 || balance() - amount < 0)
    {
        cerr << "Account::debit: Invalid amount: " << amount << endl;
        markForRollback();
    }
    try
    {
        _storage->debit(_account_name, amount);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cerr << "Account::debit: " << e << endl;
        markForRollback();
    }
}
```

Building the example

The **transfer.C** file that you created and the generated **quickstart_c.C** file are compiled and linked together to create the client program. The **bank_server.c** file that you created, along with the generated **quickstart_s.C**, **quickstart_c.C**, and **bank.C** files, are compiled and linked to create the **bank_server** program. Because Current is a pseudo object and VisiTransact-managed transactions use the Current object, the client program and server programs must also be linked with the VisiTransact **its_support** library.

Selecting a Makefile

The `<install_dir>/examples/vbe/its/` directory of your VisiTransact release contains a **Makefile** for this example. This directory also contains a **itsmk** file which is included by

the **Makefile** and defines all site-specific settings. You may need to customize the **itsmk** file. The **itsmk** file assumes that VisiTransact has been installed in the default installation directory for VisiBroker.

Compiling the example with make

WinNT: Assuming the VisiBroker ORB and VisiTransact distribution was installed in C:\vbroker, use the following commands:

```
prompt> C:
prompt> cd c:\vbroker\examples\vbe\its
prompt> nmake cpp
```

The Visual C++ **nmake** command, a standard facility, runs the **idl2cpp** compiler and then compiles each file.

Assuming the VisiBroker ORB and VisiTransact distribution was installed in /usr/local/vbroker, issue these commands:

```
prompt> cd /usr/local/vbroker/examples/vbe/its
prompt> make cpp
```

In this example, **make** is the standard UNIX facility.

Running the example

Now that you have compiled the necessary components, you are ready to run your first VisiTransact application.

Starting the Smart Agent (osagent)

Before you attempt to run VisiTransact transactional applications, you must first start the VisiBroker Smart Agent on at least one host in your local network.

If the Smart Agent has not been set up as a Windows NT service, use the following command to start the Smart Agent:

WinNT: `prompt> osagent`

For UNIX, use the following command to start the Smart Agent:

UNIX: `prompt> osagent`

While running the example, you only need to start the Smart Agent once.

Starting the VisiTransact Transaction Service

You must start an instance of the VisiTransact Transaction Service to enable transactions across the network. To do so, use the following command:

```
prompt> ots
```

While running the example, you only need to start the VisiTransact Transaction Service once.

Starting the storage_server program

Start the **storage_server** program at the command line by typing the following:

```
prompt> storage_server MyBank
```

The argument **MyBank** is the name of the Bank.

Starting the bank_server program

Start the **bank_server** program at the command line by typing the following:

```
prompt> bank_server MyBank
```

In the above example, the argument is the name of the Bank.

Note

Make sure the `PATH` environment variable includes the path to the VisiTransact directory (where the binaries are located). On Solaris, make sure the `LD_LIBRARY_PATH` environment variable includes the path to the VisiTransact shared libraries.

Running the Transaction Originator (transfer Client Program)

Start the **transfer** program at the command line with the name of the bank, followed by the source account, destination account, and amount of money you wish to transfer.

```
prompt> transfer MyBank Paul John 20
```

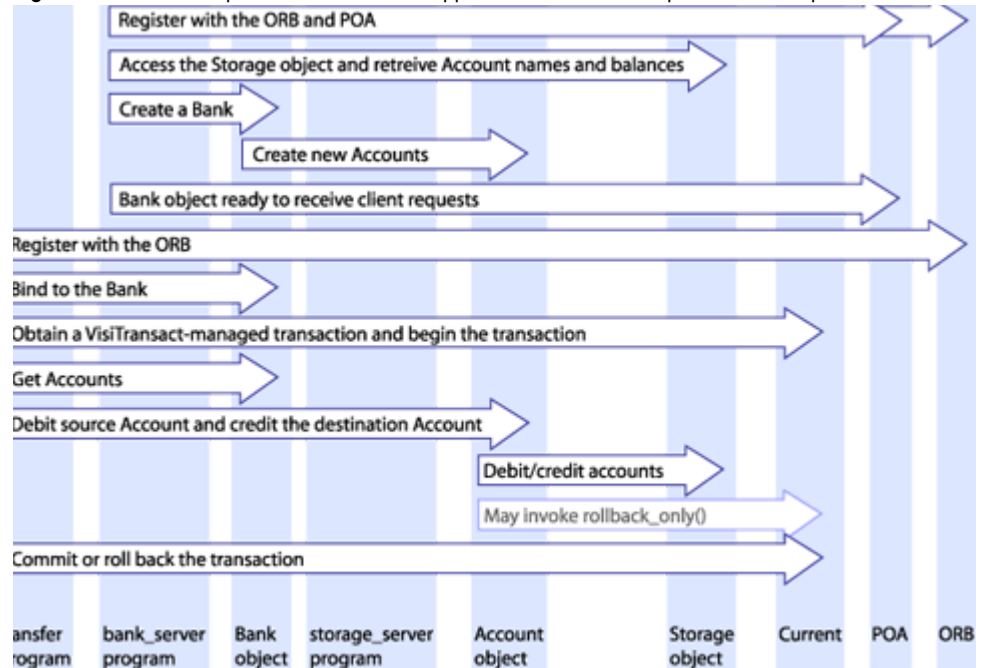
You can include multiple transfers within one execution of the transfer program. To do so, include the source account, destination account, and amount in sequence for each transfer:

```
prompt> transfer MyBank Paul John 20 Ringo George 40
```

Results

Running the **transfer** client program with “**MyBank Paul John 20**” results in the following output from the **transfer** client program:

```
Account Balance
=====
Paul      100.0
John      100.0
*** Transfer $20.0 from Paul's account to John's account ***
Account Balance
=====
Paul       80.0
John      120.0
*** Committing transaction ***
```

Figure 3.2 Visual depiction of the calls the application makes in the quick start example

Viewing the complete example

The following sections show the complete code for the quick start application.

IDL for the quick start example

```
// quickstart.idl
#include "CosTransactions.idl"
#pragma prefix "visigenic.com"

module quickstart
{
    //requires
    interface Account
    {
        float balance();
        void credit(in float amount);
        void debit(in float amount);
    };

    exception NoSuchAccount
    {
        string account_name;
    };

    interface Bank
    {
        Account get_account(in string account_name)
            raises(NoSuchAccount);
    };

    typedef sequence<string> AccountNames;

    //adapts
    interface Storage
    {
        float balance(in string account)
            raises(NoSuchAccount);
    };
};
```

```

        void credit(in string account, in float amount)
            raises(NoSuchAccount);
        void debit(in string account, in float amount)
            raises(NoSuchAccount);
        AccountNames account_names();
    };
};

```

Transfer client program

The following example shows the complete **transfer** client program in the **transfer.C** file.

```

// transfer.C

#include "quickstart_c.hh"

USE_STD_NS

int
main(int argc, char* const* argv)
{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Check the command line arguments
        if (argc % 3 != 2)
        {
            cerr << "Usage: " << argv[0] <<
" <bank-name> [<src> <dst> <amount>] ..." << endl;
            return 1;
        }

        // parse first arg
        const char *bank_name = argv[1];

        // Locate the bank.
        quickstart::Bank_var bank;

        // Get the Bank Id
        PortableServer::ObjectId_var bankId =
            PortableServer::string_to_ObjectId(bank_name);

        try
        {
            bank = quickstart::Bank::_bind("/bank_agent_poa", bankId);
            // bank = quickstart::Bank::_bind(bank_name);
        }
        catch (CORBA::Exception &ex)
        {
            const char *name;
            (bank_name == 0) ? name="NULL" : name=bank_name;
            cerr << "Unable to bind to Bank \"" << name << "\": " << ex << endl;
            return 1;
        }

        // Start a transaction.
        CosTransactions::Current_var current;
        {
            CORBA::Object_var initRef =
                orb->resolve_initial_references("TransactionCurrent");
            current = CosTransactions::Current::_narrow(initRef);
        }

        current->begin();

        CORBA::Boolean commit = 1;
        try
        {
            for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)

```

```

    {
    const char* srcName = argv[i];
    const char* dstName = argv[i + 1];
    float amount = (float)atof(argv[i + 2]);

    quickstart::Account_var src = bank->get_account(srcName);
    quickstart::Account_var dst = bank->get_account(dstName);

    cout << "Account\tBalance" << endl;
    cout << "=====\t=====" << endl;
    cout << srcName << "\t" << src->balance() << endl;
    cout << dstName << "\t" << dst->balance() << endl;
    cout << "\n*** Transfer $" << amount << " from " <<
        srcName << "'s account to " <<
        dstName << "'s account ***\n" << endl;

    src->debit(amount);
    dst->credit(amount);

    cout << "Account\tBalance" << endl;
    cout << "=====\t=====" << endl;
    cout << srcName << "\t" << src->balance() << endl;
    cout << dstName << "\t" << dst->balance() << endl;
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cout << e << endl;
        commit = 0;
    }
    catch(const CORBA::SystemException& e)
    {
        cout << "Exception: " << e << endl;
        commit = 0;
    }

    // Commit or rollback the transaction.
    if(commit)
    {
        cout << "*** Committing transaction ***" << endl;
        current->commit(0);
    }
    else
    {
        cout << "*** Rolling back transaction ***" << endl;
        current->rollback();
    }
}

catch(const CORBA::Exception& e)
{
    cerr << "Exception: " << e << endl;
    return 1;
}
catch(...)
{
    cerr << "Unknown Exception caught" << endl;
    return 1;
}
return 0;
}

```

bank_server program

The following example shows the **bank_server** program in the **bank_server.C** file.

```

// bank_server.C
#include "bank.h"

```

Viewing the complete example

```
USE_STD_NS

int
main(int argc, char* const* argv)
{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Check the command line arguments
        if(argc != 2)
        {
            cerr << "Usage: " << argv[0] << " <bank-name>" << endl;
            return 1;
        }
        const char* bank_name = argv[1];

        // get a reference to the root POA
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
            PortableServer::PERSISTENT);

        // get the POA Manager
        PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
            poa_manager,
            policies);

        // Get the Bank Id
        PortableServer::ObjectId_var bankId =
            PortableServer::string_to_ObjectId(bank_name);

        // Get a storage object for the bank.
        quickstart::Storage_var storage = quickstart::Storage::_bind("/
            bank_storage_poa", bankId);

        // Create the bank servant
        PortableServer::ServantBase_var bankServant = new BankImpl(bank_name,
            storage, orb);

        // Decide on the ID for the servant
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId(bank_name);

        // Activate the servant with the ID on myPOA
        myPOA->activate_object_with_id(managerId, bankServant);

        // Activate the POA Manager
        poa_manager->activate();

        CORBA::Object_var reference = myPOA->servant_to_reference(bankServant);
        cout << reference << " is ready" << endl;

        // Wait for incoming requests
        orb->run();
    }
    catch(const CORBA::Exception& e)
    {
        cerr << "Exception: " << e << endl;
        return 1;
    }
    catch(...)
    {
        cerr << "Unknown Exception caught" << endl;
        return 1;
    }
}
```

```

    }
    return 0;
}

```

Bank and account (transactional) objects

The example below shows the AccountRegistry, Bank, and Account classes in the bank.h file.

```

// bank.h

#include "quickstart_s.hh"
#include <vport.h>

// The AccountRegistry is a holder of Bank account implementations
class AccountRegistry
{
public:
    AccountRegistry() : _count(0), _max(16), _data((Data*)NULL)
    {
        _data = new Data[16];
    }

    ~AccountRegistry() { delete[] _data; }

    void put(const char* name, PortableServer::ServantBase_ptr servant) {

        VISMutex_var lock(_lock);

        if (_count + 1 == _max) {
            Data* oldData = _data;
            _max += 16;
            _data = new Data[_max];
            for (CORBA::ULong i = 0; i < _count; i++)
                _data[i] = oldData[i];
            delete[] oldData;
        }

        _data[_count].name = name;
        servant->_add_ref();
        _data[_count].account = servant;
        _count++;
    }

    PortableServer::ServantBase_ptr get(const char* name) {

        VISMutex_var lock(_lock);

        for (CORBA::ULong i = 0; i < _count; i++) {
            if (strcmp(name, _data[i].name) == 0) {
                _data[i].account->_add_ref();
                return _data[i].account;
            }
        }
        return PortableServer::ServantBase::_nil();
    }

private:
    struct Data {
        CORBA::String_var name;
        PortableServer::ServantBase_var account;
    };

    CORBA::ULong _count;
    CORBA::ULong _max;
    Data* _data;
    VISMutex _lock; // Lock for synchronization
};

class BankImpl : public POA_quickstart::Bank
{
private:

```

Viewing the complete example

```
quickstart::AccountNames_var _account_names;
quickstart::Storage_var _storage;
AccountRegistry _accounts;
PortableServer::POA_var _account_poa;
public:
    BankImpl(const char* bank_name,
             quickstart::Storage* storage, CORBA::ORB* orb);
    virtual ~BankImpl();
    virtual quickstart::Account* get_account(const char* account_name);
};

class AccountImpl : public POA_quickstart::Account
{
private:
    CORBA::String_var _account_name;
    quickstart::Storage_var _storage;
public:
    AccountImpl(const char* account_name,
               quickstart::Storage* storage);
    virtual CORBA::Float balance();
    virtual void credit(CORBA::Float amount);
    virtual void debit(CORBA::Float amount);
private:
    virtual void markForRollback();
};
```

The next example shows the `BankImpl` and `AccountImpl` classes in the `bank.C` file.

```
// bank.C

#include "bank.h"

USE_STD_NS

BankImpl::BankImpl(const char* bank_name,
                  quickstart::Storage* storage, CORBA::ORB* orb)
{
    _account_names = storage->account_names();
    _storage = quickstart::Storage::_duplicate(storage);

    PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow(orb->resolve_initial_references("RootPOA"));
    CORBA::PolicyList policies;
    policies.length(2);
    CORBA::Any policy_value;
    policy_value <<= CosTransactions::REQUIRES;
    policies[0] = orb->create_policy(CosTransactions::OTS_POLICY_TYPE,
                                   policy_value);
    policies[1] = root_poa-
>create_implicit_activation_policy(PortableServer::IMPLICIT_ACTIVATION);
    _account_poa = root_poa->create_POA("account_poa",
                                       PortableServer::POAManager::_nil(),
                                       policies);
    _account_poa->the_POAManager()->activate();
    return;
}

BankImpl::~BankImpl()
{
}

quickstart::Account_ptr
BankImpl::get_account(const char* account_name)
{
    // Lookup the account in the account dictionary.
    PortableServer::ServantBase_var servant = _accounts.get(account_name);
    CORBA::Boolean foundAccount = 0;

    if (servant == PortableServer::ServantBase::_nil()) {
        for(CORBA::ULong i = 0; !foundAccount && i < _account_names->length(); i++) {
            if (!strcmp(_account_names[i], account_name)) {
                servant = new AccountImpl(account_name, _storage);

                // Print out the new account
            }
        }
    }
}
```



```

    cout << "Created " << account_name << "'s account." << endl;

    // Save the account in the account dictionary.
    _accounts.put(account_name, servant);

foundAccount = 1;
}
}
if (!foundAccount) {
    throw quickstart::NoSuchAccount(account_name);
    return 0;
}
}

try {
    CORBA::Object_var ref = _account_poa->servant_to_reference(servant);
    quickstart::Account_var account = quickstart::Account::_narrow(ref);
    cout << "account generated." << endl;
    return quickstart::Account::_duplicate(account);
}
catch(const CORBA::Exception& e) {
    cerr << "_narrow caught exception: " << e << endl;
    return quickstart::Account::_nil();
}
throw quickstart::NoSuchAccount(account_name);
return 0;
}

AccountImpl::AccountImpl(const char* account_name,
    quickstart::Storage* storage)
{
    _account_name = CORBA::strdup(account_name);
    _storage = quickstart::Storage::_duplicate(storage);
}

void
AccountImpl::markForRollback()
{
    try
    {
        CORBA::ORB_var orb = CORBA::ORB_init();
        CORBA::Object_var initRef =
            orb->resolve_initial_references("TransactionCurrent");
        CosTransactions::Current_var current =
            CosTransactions::Current::_narrow(initRef);
        current->rollback_only();
    }
    catch(const CosTransactions::NoTransaction&)
    {
        throw CORBA::TRANSACTION_REQUIRED();
    }
}

CORBA::Float
AccountImpl::balance()
{
    try
    {
        return _storage->balance(_account_name);
    }
}

catch(const quickstart::NoSuchAccount& e)
{
    cerr << "Account::balance: " << e << endl;
    markForRollback();
    return 0;
}
}

void
AccountImpl::credit(CORBA::Float amount)
{

```

Viewing the complete example

```
    if(amount < 0)
    {
        cerr << "Account::credit: Invalid amount: " << amount << endl;
        markForRollback();
    }
    try
    {
        _storage->credit(_account_name, amount);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cerr << "Account::credit: " << e << endl;
        markForRollback();
    }
}

void
AccountImpl::debit(CORBA::Float amount)
{
    if(amount < 0 || balance() - amount < 0)
    {
        cerr << "Account::debit: Invalid amount: " << amount << endl;
        markForRollback();
    }
    try
    {
        _storage->debit(_account_name, amount);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
        cerr << "Account::debit: " << e << endl;
        markForRollback();
    }
}
```


4

Creating a transactional object

The creation of a transactional object for server and clients can be achieved in two ways: by inheriting transactional object interfaces or by direct implementation.

Inheriting transactional object interfaces

By inheriting of `CosTransactions::TransactionalObject` interfaces in the object interfaces will define the object to be a transactional object in the server and client.

Implementing transactional object interfaces

The other way, which is compliant to OMG OTS 1.4 specification and illustrated in the example, will allow server and clients to define their transactional object to enhance its transaction controls. With these new improvements, servers are able to force a transaction requirement upon a target object by setting appropriate policies for it. Meanwhile, clients should have to make corresponding invocations according to the target object's requirement, with some new client side policies to regulate their behaviors. Thus a strong semantic control is guaranteed.

It also provide the support for policy creation and policy check on both client and server sides. It safeguards the transactional object reference creation and transactional invocations in a distributive transaction environment.

Transactional POA policy interfaces

OTSPolicy

This policy is used to describe the shared transaction behavior of a target object. It has three possible values:

- `REQUIRES` – the target object need a transaction to be present with the incoming calls.
- `FORBIDS` – no transaction should be present with the invocations on the target object.
- `ADAPTS` – the target object is sensitive to the presence or absence of a current transaction.

InvocationPolicy

This policy specifies what kind of transactions the target object supports. A target object can choose to supported SHARED transaction model, UNSHARED transaction model, or EITHER of them by setting the invocation policies with the corresponding values.

Note:

When defining both OTSPolicy and InvocationPolicy for a target object, not all combinations are valid. See the OMG OTS specification version 1.4 for details. Any one of invalid combinations in policy creation will result in an InvalidPolicy exception.

NonTxTargetPolicy

This policy is used to PERMIT a client invocation on a non-transactional target object during an active transaction or to PREVENT the client from doing it. Any client invocation that conflicts with the policy will get an INVALID_TRANSACTION exception.

Affected Server Behaviors

A new transactional server should use OTSPolicy and InvocationPolicy (optional) to control the transactional behaviors of the objects it creates. The previously specified TransactionalObject should not be used in the new server.

To create an object with needed transactional behaviors, a server must create POAs with proper policies. A POA uses the policy values with which it is created to control the object reference creation. VisiTransact Transaction Manager will examine the validity of those policies and then do one of the following:

- 1 If all policies are valid, a POA with specified policies is created for object activation and reference creation;
- 2 If policies are invalid, an exception is raised; or
- 3 If OTSPolicy is absent on creating a POA, VisiTransact Transaction Manager provides a default value. (FORBIDS).

In the absence of InvocationPolicy, objects thus created should be treated as if they support the InvocationPolicy of EITHER.

Affected Client Behaviors

Clients shall make invocations under circumstances conforming to the requirements of the target objects. Otherwise they will get exceptions from VisiTransact Transaction Manager.

For an object that REQUIRES transactions, a call on it must happen within the scope of an active transaction. For example, the calling thread must be associated with an active transaction.

For an object that FORBIDS transactions, a call on it must be made outside the scope of any active transaction. For example, the calling thread is associated with no transaction.

For an object that ADAPTS transactions, a call on it is allowed in either case. However, the target object will behave differently depending on whether the incoming call is associated with an active transaction or not.

During an active transaction, the client uses the NonTxTargetPolicy to manipulate calls on non-transactional objects. If a client doesn't set the policy, the default value for this is PERMIT.

Dealing with UNSHARED transactions

The support for UNSHARED transactions is not fully provided with this release because the current Visibroker has a model of Asynchronous Method Invocation called NativeMessaging that is different from OMG AMI model. So VisiTransact Transaction Manager servers and clients shouldn't directly participate in an unshared transaction.

However, the InvocationPolicy of any valid values can be created successfully for a POA at the server side irrespective of this limitation.

5

Determining your approach to transactions

This section provides an overview of the directions you can take when building transactional applications with VisiTransact Transaction Manager.

Transaction management approaches

A program can choose the type of context management it will use, and the method of context propagation used to transmit the transactional context to other objects. Using a type of context management does not restrict your choice of transaction propagation.

Direct vs. indirect context management

The CORBAservices Transaction Service specification from OMG defines the following types of context management:

- **Indirect Context Management.** With indirect context management, an application uses the Current object provided by the Transaction Service to associate the transaction context with the application thread of control and manage it.
- **Direct Context Management.** In direct context management, an application manipulates the Control and other objects associated with the transaction.

Using indirect context management simplifies programming, and enables your application to take advantage of performance enhancements and optimizations that are possible when the VisiTransact Transaction Service controls the transaction context. For example, VisiTransact-managed transactions take advantage of the underlying VisiBroker ORB to minimize remote calls. Further, VisiTransact-managed transactions save system resources by caching the propagation context and transaction context at the application end, thereby eliminating unnecessary remote calls to retrieve this data.

Direct context management might be more convenient if you are using explicit propagation or you are trying to use multiple VisiTransact Transaction Service instances to originate transactions. In addition, if you do not want to link in VisiTransact libraries, you must use direct context management. In rare circumstances you may want to use your own stubs from your own IDL files rather than use VisiTransact libraries. The only way you can use your own stubs is by using direct context

management. If you use indirect context management, you use Current; when you use Current, you use VisiTransact libraries.

If you use direct context management, or a mixture of both context management modes, you must ensure transactional integrity for your application. Once you use direct context management, the VisiTransact Transaction Service has lost the ability to check transaction completeness. See [“How does the VisiTransact Transaction Service ensure checked behavior?”](#) for more information about checked behavior.

Implicit vs. explicit propagation

The CORBAservices Transaction Service specification from OMG defines the following propagation types:

- **Implicit propagation.** With implicit propagation, requests are implicitly associated with the application's transaction—meaning they share the application's transaction context. The transaction context is transmitted implicitly to the participating objects by the VisiTransact Transaction Service, without direct intervention by the transaction originator. An object that supports implicit propagation would not typically expect to receive any Transaction Service object as an explicit parameter.
- **Explicit propagation.** With explicit propagation, the transaction originator (and potentially participating transactional objects) propagates a transaction context by passing objects defined by the Transaction Service as explicit parameters.

The major advantage to implicit propagation is that the VisiTransact Transaction Service handles transaction propagation for you. Another advantage is that implicit propagation does not require you to change the signatures of existing methods to support transactions—by making the object transactional, you enable all of the object's methods to be executed as part of a transaction.

Explicit propagation also has its advantages. First, it allows you to mix transactional and non-transactional methods within an object. This is useful if you want to have transactional semantics for one method but not for others in a transaction.

Secondly, you might use explicit propagation if you require interoperability with CORBA 1.x implementations (such as VisiBroker 2.0). Because explicit propagation does not require cooperation between the ORB and the Transaction Service, it can be used for this kind of backward-compatibility.

A third reason for using explicit propagation is that it allows other objects to terminate transactions. In other words, explicit propagation enables you to pass the Terminator to another transaction participant; this enables the participant to terminate the transaction.

Context management and propagation

A client may use either direct or indirect context management with either implicit or explicit propagation. This results in several ways in which client applications may communicate with transactional objects:

- Indirect Context Management with Implicit Propagation
- Indirect Context Management with Explicit Propagation
- Direct Context Management with Implicit Propagation
- Direct Context Management with Explicit Propagation

Indirect context management with implicit propagation

The client application uses methods on the Current object to create and control its transactions. When it issues requests on transactional objects, the transaction context associated with the current thread is implicitly propagated to the object.

VisiTransact-managed transactions fall into this category. With VisiTransact-managed transactions, VisiTransact guarantees checked behavior. For more information about checked behavior, see [“How does the VisiTransact Transaction Service ensure checked behavior?”](#).

Note

Indirect context management with implicit propagation is not exactly the same as VisiTransact-managed transactions. VisiTransact-managed transactions specifically dictate the use of `Current::begin` followed by implicit propagation.

See [“Creating and propagating VisiTransact-managed transactions”](#) for details on using VisiTransact-managed transactions.

Indirect context management with explicit propagation

The client uses a combination of the `Current`, `Control`, and other objects which describe the state of the transaction. A client application that uses the `Current` object (and therefore, is also automatically using implicit propagation) can use explicit propagation by gaining access to the `Control` object with the `Current::getControl()` method. It can use a VisiTransact Transaction Service object as an explicit parameter to a transactional object. This is explicit propagation.

Direct context management with implicit propagation

The client uses a combination of the `Current`, `Control`, and other objects which describe the state of the transaction. A client that accesses the VisiTransact Transaction Service objects directly can use the `Current::resume()` method to set the implicit transaction context associated with its thread. This allows the client to invoke methods of objects that require implicit propagation of the transaction context.

Direct context management with explicit propagation

The client application directly accesses the `Control` object and the other objects which describe the state of the transaction. To propagate the transaction to an object, the client must include the appropriate VisiTransact Transaction Service object as an explicit parameter of a method.

See [“Other methods of creating and propagating transactions”](#) for details on managing transactions from your application.

In-process vs. out-of-process VisiTransact transaction service

If most of your transactions are isolated to, and used within, a single process, you may decide to use an in-process instance of the VisiTransact Transaction Service. However, this means that the requirements for transactions (i.e., high availability)—usually handled by a stand-alone instance of the VisiTransact Transaction Service—can only be met if the application process remains running when transactions are in progress. This requirement is especially important if other applications (outside of the process) are using the instance of the VisiTransact Transaction Service that you have embedded within your application process. See [“Embedding a VisiTransact Transaction Service instance in your application”](#).

You can use multiple instances of the VisiTransact Transaction Service on your network. To make the behavior of your transactions more predictable, you can specify which instance of the VisiTransact Transaction Service your transaction originator will use.

- You can control the instance of the VisiTransact Transaction Service used by arguments passed to `ORB_init()` or by how you set the `Current` interface attributes. The `Current` attributes will override any arguments passed to `ORB_init()`. This will only take effect for subsequent transactions using `Current::begin()`.
- For direct context management, bind by name to the appropriate instance of the `TransactionFactory`.

See [“Discovering an instance of the VisiTransact Transaction service”](#) for an explanation of how to set the `-Dvbroker.ots.currentName` argument.

Multithreading

VisiTransact is multithreaded. Multithreaded applications benefit from the features of the underlying VisiBroker ORB including its thread pooling and connection management capabilities.

Although the thread and connection management of the VisiBroker ORB can conserve system resources, the thread pooling strategy could be a disadvantage if you need control over which thread is assigned to a particular transaction. With the thread pooling model, a worker thread is assigned for each client request, but only for the duration of that particular request. Consider other threading models offered by VisiBroker if you need more control. Also note that thread safety issues may arise if other libraries are not thread safe.

Integrating existing applications and transactional systems

You can integrate several external transactional systems using other CORBA Transaction Services. Since VisiTransact is fully CORBA 3.0-compliant, it is interoperable with other CORBA 3.0-compliant implementations of the OMG CORBA Transaction Service specification. VisiTransact provides valuable extensions to the CORBAservices specification (useful methods such as `begin_with_name()`, and other features) that cannot be handled by other transaction service implementations.

In addition, you can use any CORBAservices-compliant resource provided by yourself, a third-party, or a database vendor.

Another option is to implement your own Resource using the `Resource` interface. This option requires complex programming because logging and recovery, heuristics, and other necessary coding is not handled for you.

Using a combination of approaches

You can mix and match any of the approaches described in this chapter to suit the purposes of your distributed, transactional application.

- **Mixing various types of transaction approaches.** For example, you might have a transaction using explicit propagation and then switch to implicit. See [“Changing from explicit propagation to implicit”](#) for more information.
- **Integrating multiple systems with your VisiTransact application.** For example, you can use databases, transaction processing monitors, and messaging software in your transactional application—and integrate them all with VisiTransact.

Implementing transactions for the web

If you are developing a web-based transactional application, you may decide to use web browsers as a front-end to the application, and leave the transaction origination and other logic to a server-based object.

Keeping the VisiTransact transaction within the boundaries of the Web server's local network means that you gain performance advantages because of the locality of the VisiTransact Transaction Service and the transaction participants. Also, you provide local autonomy of transactions within one company's control. With this application architecture, communication problems across external networks will not affect transaction completion or integrity.

Building VisiTransact applications

When designing C++ applications that use VisiTransact you can use standalone instances of the VisiTransact Transaction Service, or embed instances of the VisiTransact Transaction Service in your C++ application components.

The following sections describe these alternatives in greater detail.

Using stand-alone VisiTransact Transaction Service instances

Most VisiTransact applications will use an instance of the VisiTransact Transaction Service that is running on the network—rather than embedding an instance in their process. When the application is executed it can use any available VisiTransact Transaction Service instance, or control the instance of the VisiTransact Transaction Service that is used.

Any program that uses the VisiTransact Transaction Service interfaces must be linked with `its_support.lib` (`its_support.so` on Solaris).

Note

If the program only uses direct context management with explicit propagation, it can use the stubs and header files generated from the `CosTransactions.idl` or `VISTransactions.idl` files.

Embedding a VisiTransact Transaction Service instance in your application

Embedding a VisiTransact Transaction Service instance in a C++ executable entails linking in `ots_r.lib` (`ots_r.so` on Solaris) and `its_support.lib` (`its_support.so` on Solaris) libraries with your application. Adding these libraries to the link line embeds an instance of the VisiTransact Transaction Service in the application's process.

If you link with VisiTransact libraries, you must include the `_c.hh` and `_s.hh` files provided by VisiTransact. You cannot generate your own stub files. This is to ensure you are using versions of the headers that are compatible with the objects embedded in VisiTransact libraries. You must perform this step if you link with the VisiTransact libraries.

Additionally, you must explicitly initialize and terminate the instance of the VisiTransact Transaction Service from your application as described in the following steps:

- 1 Include `visits.h` in your C++ application.
- 2 Initialize the VisiTransact server components with `ORB_init()`. Invoke `VISTransactionService::init()` to initialize the VisiTransact Transaction Service instance. This must happen after the `ORB_init()` invocation. For example:

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
VISIts::init(argc, argv);
```

- 3 Invoke `VISTransactionService::terminate()` to shutdown the VisiTransact Transaction Service instance.
- 4 You must have the following in the link line:

```
UNIX:ots_r.so
WinNT:ots_r.lib
```

Note

`ots64_r.so` and `its_support64.so` on 64 bit platform. `ots_r.a`(`ots64_r.a`) and `its_support.a`(`its_support64.a`) on AIX and `ots_r.sl`(`ots64_r.sl`) and `its_support.sl`(`its_support64.sl`) on HP-UX.

- 5 Confirm that the VisiTransact Transaction Service is up and running by using `osfind`.

The example below shows an application that embeds the VisiTransact Transaction Service.

```
// Application main
#include <visits.h> // for VISIts
#include <corba.h>
```

```
int main(int argc, char** argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    VISTransactionService::init(argc, argv);
    // the main work of the application is now performed
    ...
    VISTransactionService::terminate();
}
```

Binding to the embedded instance of the VisiTransact Transaction Service

When you have the VisiTransact Transaction Service embedded in the application server, you must make sure that the client binds to the correct instance of the VisiTransact Transaction Service. To do so, you must specify the name of the VisiTransact Transaction Service when starting the client application using certain command-line arguments. This name must match the one that is embedded in the application server.

Note

If you are creating transactions directly from the `TransactionFactory` rather than using `Current`, then the client will have to bind to the correct `TransactionFactory`. Refer to the semantics for binding to any CORBA object to make sure the client binds to the correct object.

Using header files supplied with VisiTransact

To compile a C++ source file that will link with `its_support.lib` or `ots_r.lib`, you must include the version of `CosTransactions_c.hh` or `VISTransactions_c.hh` supplied by VisiTransact, *not* an IDL client stub header file that you generate from `CosTransactions.idl` or `VISTransactions.idl`. (The objects you will link against in the VisiTransact-supplied libraries are valid only against the header files used to build them.) Any application using the `Current` interface will be linking against these libraries.

6

Creating and propagating VisiTransact-managed transactions

This section focuses on using the `Current` interface in VisiTransact-managed transactions. It includes information about how to gain access to a VisiTransact-managed transaction with `Current`, and `begin`, `rollback`, and `commit` the transaction using the methods in the `Current` interface. It also explains how transactional objects can share in a VisiTransact-managed transaction.

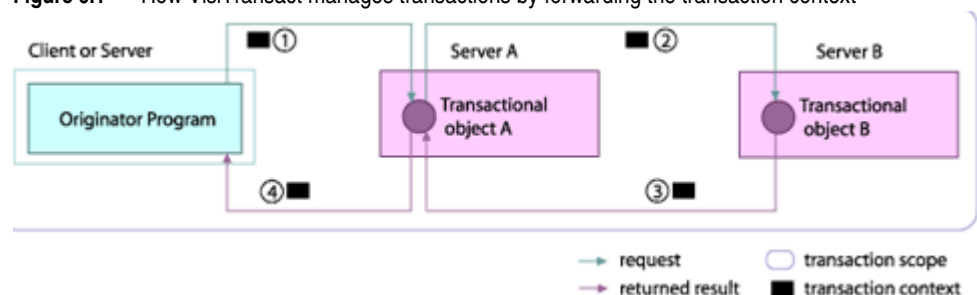
Introducing `Current` as used in VisiTransact-managed transactions

With VisiTransact-managed transactions you are using the `Current` interface for all transaction management. You are beginning transactions using `Current` and you are using `Current` for the implicit transaction propagation. This means that you will always originate your transactions using `Current::begin()`.

`Current` is an object that is valid for the entire process and manages the association of each thread's transaction context. Each thread has its own independent, isolated association with a transaction context.

In VisiTransact-managed transactions, transaction participants share the same transaction context because VisiTransact transparently forwards the transaction context to each participant. This means that the state of a transaction is maintained as the originator calls on other objects to perform actions, which may in turn call other objects.

Figure 6.1 How VisiTransact manages transactions by forwarding the transaction context



- 1 The transaction originator requests that Object A performs the `doWork()` method.

- 2 Object A requests that Object B performs the `doMoreWork()` method.
- 3 Object B returns its results to Object A.
- 4 Object A returns its results to the transaction originator.

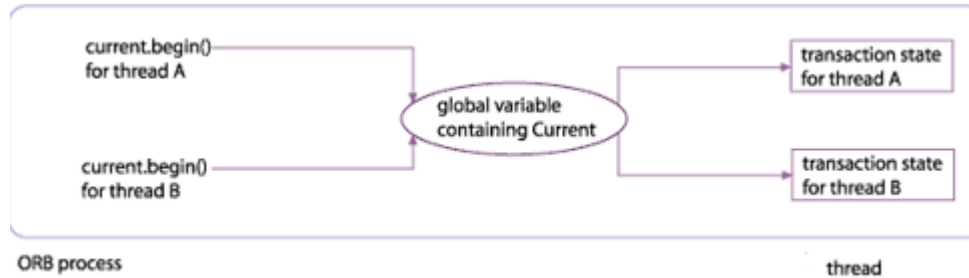
In all four steps, the VisiTransact Transaction Service automatically and transparently propagates the transaction context between the transactional objects. If the first transactional object makes a subsequent request of another object, the transaction context flows on to this second object, as is shown in steps 1 to 4 in the figure above. If an object is not a transactional object, it does not receive the context, and, therefore, it cannot forward the context to any other object.

How does Current work?

VisiTransact-managed transactions are made possible with the Current object. The `Current` interface defines methods that simplify transaction management for most applications.

The `Current` interface is supported by a pseudo object whose behavior depends upon, and may alter, the transaction context associated with the invoking thread. Because Current is not a CORBA object, it cannot be accessed remotely.

Figure 6.2 Using a global Current object to start transactions across threads in a process



A new transaction created with the `begin()` method is associated with the specific thread that called the method. A thread can be associated with only one transaction at a time. If a thread exits, or if the transaction originator's thread returns without completing the transaction, then any active transaction left associated with the thread will timeout and be rolled back.

Note

The application does not need to implement critical sections to ensure synchronization between threads when using the Current object.

Obtaining a Current object reference

For C++:

To gain access to a VisiTransact-managed transaction, you must obtain an object reference to the Current object. The users need to resolve to Current only once in the lifetime of a process. The Current object reference is valid throughout the process.

The following steps describe the general process for obtaining a reference to a Current object, and are include code examples.

- 1 Call the ORB `resolve_initial_references()` method. This method obtains a reference to the Current object.
- 2 Narrow the returned object to a `CosTransactions::Current` or `VISTransactions::Current` object.

When you narrow to `CosTransactions::Current`, you specify your use of the original set of methods provided by the `CosTransactions` module. When you narrow to

`VISTransactions::Current`, you specify the original set and the extensions to the `Current` interface provided by VisiTransact.

See [“Extensions to the Current interface”](#) for descriptions of VisiTransact extensions to the `Current` interface.

The following example shows examples of these alternatives in C++.

```
// To use OMG-compliant methods and behavior
CORBA::Object_var
obj = orb->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var
current = CosTransactions::Current::_narrow(obj);
// To use OMG behavior on CosTransactions methods and also use the
// additional VisiTransact methods
CORBA::Object_var
obj = orb->resolve_initial_references("TransactionCurrent");
VISTransactions::Current_var
current = VISTransactions::Current::_narrow(obj);
```

For Java:

However for Visibroker for Java, the users need to resolve to `Current` everytime they access it in another thread, to gain access to a VisiTransact-managed transaction.

The following steps describe the behavioral difference between Java and C++ OTS:

- When a user call `send_deferred()` during a transaction, the resultant behavior is indeterminate.
- Commitment of a transaction in another thread within the same process that did not start the transaction is not restricted. This is also the case for VBC implementation.
- Users might not be able to make use of OTS in conjunction with naming service in certain programmatic ways (example: do rir to naming service and then resolve to a name).
This is because in VBJ implementation looks for a transaction service instance only upon the first transaction related call. (e.g. `Current::begin()`)

Working with the Current interface and its methods

The `Current` interface offers several methods for managing the current thread or context's transaction. The table below describes these methods.

See [Extensions to the Current interface](#) for descriptions of VisiTransact extensions to the `Current` interface.

Method	Description
<code>begin()</code>	Creates a new transaction. The <code>SubtransactionsUnavailable</code> exception will be raised if a transaction is already in progress. The transaction created will have a timeout from the last call to <code>set_timeout()</code> . If <code>set_timeout()</code> is not issued, the default timeout value of the VisiTransact Transaction Service is used.
<code>commit(in boolean report_heuristics)</code>	Completes the transaction. Only the originator can call this method. The transaction is rolled back if it cannot be committed.
<code>rollback()</code>	Rolls back the transaction. Only the originator can call this method.
<code>rollback_only()</code>	Modifies the transaction so that it will be rolled back. This method is used by participants other than the originator to ensure that the transaction will be rolled back.
<code>get_status()</code>	Returns the status of the transaction. If there is no transaction in progress, the <code>StatusNoTransaction</code> value is returned.
<code>get_transaction_name()</code>	Returns the transaction name—this is a descriptive string assigned to the transaction either by the VisiTransact Transaction Service or the user. If there is no transaction in progress, an empty string is returned.

Method	Description
set_timeout()	Sets a timeout period by which any new transactions in the process must complete. If the timeout is set to 0, it sets any subsequent transaction that is begun to the default transaction timeout for the VisiTransact Transaction Service instance that it uses. If the timeout is greater than 0, it sets the new timeout to the specified number of seconds. If the seconds parameter exceeds the maximum timeout valid for VisiTransact Transaction Service instance being used, then the new timeout is set to that maximum, to bring it in range. When a transaction created by a subsequent call to <code>begin()</code> in any thread in the process takes longer to start transaction completion than the established timeout, it will be rolled back. Otherwise, the timeout is ignored. The timeout does not affect transactions that are already in progress.
get_control()	Returns a Control object that represents the transaction context currently associated with the process or thread. This Control object can be used to resume this transaction context if the transaction context is suspended, or to perform explicit propagation.
suspend()	Suspends the current transaction. This method returns a Control object that represents the transaction context currently associated with the process or thread. This object can be used to resume this transaction context.
resume()	Resumes a suspended transaction or associates a transaction context with the process or thread.

Note

If you use `get_control()`, `suspend()` or `resume()`, it might affect checked behavior. For more information, see [“How does the VisiTransact Transaction Service ensure checked behavior?”](#).

As shown the following example, you can use the methods shown in the table above to perform actions with VisiTransact-managed transactions. This example shows the **MyBank** interface for the transactional object which defines the `withdraw()` method.

```
#include <CosTransactions.idl>

interface MyBank
{
    float    balance(in long accountNo);
    boolean  withdraw(in long accountNo, in float amount);
};
```

The next example shows an example of an originator beginning a transaction and calling the `withdraw()` method on the **MyBank** transactional object. Then the originator either commits or rolls back the transaction.

```
...
// get object reference to my object implementation
MyBank_var bank = MyBank::_bind();

// start a transaction
current->begin();

if(bank->withdraw(10, 444))
{
    // invoke a CORBA request
    current->commit(0);
}
else
{
    current->rollback();
}
...

```

If an originator begins a transaction, it must commit or rollback the transaction. The VisiTransact Transaction Service will rollback the transaction if it times out. For example, a situation when a transaction may time out is if the originator's thread dies before the transaction is committed or rolled back.

Multiple threads participating in the same transaction

If you have a process and want to use multiple threads in the same transaction, you must pass the transaction context to each of the threads. In the typical scenario, you will start with a thread that has the transaction context either because it is the originator and invoked `Current::begin()`, or because an operation passed the transaction context to it (implicitly or explicitly) and it needs to propagate that context to the other threads. This can be achieved by making the transaction's Control object available to the other threads and they can invoke `Current::resume()` specifying that Control object. Note that VisiTransact cannot provide checked behavior in this case.

Using multiple transactions within a context or thread

Note

This release of the VisiTransact Transaction Service does not support nested transactions. However, the procedure described in this section can be used to enable multiple transactions per thread or context.

You can manage multiple transactions within a thread; however, a thread can have only one active transaction at a time. The `suspend()` method is used to disassociate the current context, and `resume()` is used to associate another context. The table in [“Working with the Current interface and its methods”](#) describes the methods used to implement multiple transactions within a thread.

The following example shows an example of an object that originates multiple transactions from within a thread. This example illustrates that the `MyBank_impl::withdraw()` method can suspend the transaction in which the method was called, start a new transaction, and then resume the earlier transaction.

```

CORBA::Boolean MyBank_impl::withdraw( CORBA::Long accountNo,
                                       CORBA::Float amount)
{
    try
    {
        // check to see if a transaction has been started
        CORBA::Object_var
            obj = orb->resolve_initial_references("TransactionCurrent");
        CosTransactions::Current_var
            current = CosTransactions::Current::_narrow(obj);
        // Suspend the current transaction. If there is no current transaction,
        // the control will be null.
        CosTransactions::Control_var control = current->suspend();
        // start a new transaction
        try
        {
            current->begin();
            // do your logic
            current->commit(0);
        }
        catch(...)
        {
            // resume earlier transaction
            current->resume(control);
            throw;
        }
    }
    catch(..) { }
}

```

Discovering an instance of the VisiTransact Transaction service

By default, the first time you start a transaction with `begin()` an instance of the VisiTransact Transaction Service is found using the Smart Agent. For details on the Smart Agent, see the *VisiBroker Developer's Guide*.

You can control the instance of the VisiTransact Transaction Service used with arguments passed to `ORB_init()`, or by how you set the `VISTransactions::Current` interface arguments. The Current arguments will override any arguments passed to `ORB_init()`. The arguments will only take effect for subsequent transactions that use `Current::begin()`.

The arguments that you can set are:

- **Host Name.** The Smart Agent will find any available VisiTransact Transaction Service instance that is located on the specified host.
- **VisiTransact Transaction Service Name.** The Smart Agent will find the named VisiTransact Transaction Service instance anywhere on the network.
- **IOR.** VisiTransact uses the specified IOR for the requested Transaction Service (`CosTransactions::TransactionFactory`) to locate the desired instance of a Transaction Service implementation on the network. This argument enables VisiTransact to operate without the use of a Smart Agent (**osagent**).

If you specify a combination of Host Name and VisiTransact Transaction Service Name, the Smart Agent will find the named VisiTransact Transaction Service instance on the named host. If you specify the IOR with either the Host Name or VisiTransact Transaction Service Name, the Smart Agent will find the VisiTransact Transaction Service instance by IOR only. It ignores the Host Name and VisiTransact Transaction Service Name.

The following table lists the arguments you can use to specify an instance of the VisiTransact Transaction Service.

Characteristic	Argument to ORB_init()	VISTransactions::Current Interface
Host Name	-Dvbroker.ots.currentHost	ots_host
VisiTransact Transaction Service Name	-Dvbroker.ots.currentName	ots_name
IOR	-Dvbroker.ots.currentFactory	ots_factory
	vbroker.ots.currentTimeout	
	vbroker.orb.enableTransactions	Automatically will load up <code>com.borland.vbroker.CosTransactions.implicit.Init</code>

The following example shows how to specify an instance of the VisiTransact Transaction Service by name using the `ots_name` argument of the `VISTransactions::Current` interface.

```
...
CORBA::Object_var obj =
    orb->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var current =
    VISTransactions::Current::_narrow(obj);

// to set the VisiTransact Transaction Service instance
current->ots_name("MyTxnSvc");
...
```

Propagating VisiTransact-managed transactions

To enable implicit propagation, a participant must be a transactional object—it must inherit from `CosTransactions::TransactionalObject` or define the `OTSPolicy` object with either the `REQUIRE` or `ADAPT` value. To enlist another participant in a transaction, the object enlisting the other participant must have a transaction associated with the current thread.

There are three ways a transaction is associated with the current thread:

- If a participant in a transaction implicitly receives the transaction context from another object.
- If a new transaction is started using `Current::begin()`.
- If a transactional object context has been associated with the thread using `Current::resume()`.

Ensuring a transaction is in progress

If a participant requires a transaction, it should verify that a transaction is not currently in progress before beginning a new transaction. If a participant attempts to begin a new transaction when a transaction is already running, the VisiTransact Transaction Service throws a `CosTransactions::SubtransactionsUnavailable` exception. A participant that begins a new transaction must also rollback or commit the transaction before returning.

The following example illustrates how a server object ensures that its work is done as a transaction and avoids starting a new transaction when a transaction is already in progress.

```

CORBA::Boolean MyBank_impl::withdraw( CORBA::Long accountNo,
                                      CORBA::Float amount)
{
    // get ORB instance
    CORBA::ORB_ptr orb = CORBA::ORB_init();

    // get Current reference
    CORBA::Object_var
        obj = orb->resolve_initial_references("TransactionCurrent");
    CosTransactions::Current_var
        current = CosTransactions::Current::_narrow(obj);

    CORBA::Boolean startFlag = 0; //use to signal creation of the transaction
    CORBA::Boolean status = 0;
    try
    {
        // check to see if a transaction has been started
        if(current->get_status() == CosTransactions::StatusNoTransaction)
        {
            current->begin();
            startFlag = 1; //we started and now own the current transaction
        }
        if(balance(accountNo) > amount)
        {
            // withdraw logic
            ...
            status = 1;
        }
    }
    catch(...) { }
    if(startFlag && status)
    {

```

```

        current->commit();
    }
    else if(startFlag)
    {
        current->rollback();
    }
    return status;
}

```

Marking a transaction for rollback

When using `Current`, only an originator can terminate the transaction with `commit()` or `rollback()`. In this case, if a participant does not want the transaction to commit, it can use the `rollback_only()` method from the `Current` interface. When the `rollback_only()` method is called by a participant, the transaction associated with the target object is modified so that the only possible outcome is to rollback the transaction.

When invoking `rollback_only()`, the `CosTransactions::NoTransaction` exception is raised if there is no transaction in progress. The following example shows how a participant would use the `rollback_only()` method.

```

...
CosTransactions::Current_var current;
current->rollback_only();
...

```

Obtaining transaction information

A participant can obtain information about the current transaction such as its transaction name or transaction status using methods in the `Current` interface. The following table discusses these methods.

Method	Description
<code>get_status()</code>	Returns the status of a transaction associated with the current thread.
<code>get_transaction_name()</code>	Returns a printable string describing the transaction associated with the current thread.

The `get_status()` method can return one of the following values:

- `StatusActive`
- `StatusCommitted`
- `StatusCommitting`
- `StatusMarkedRollback`
- `StatusNoTransaction`
- `StatusPrepared`
- `StatusPreparing`
- `StatusRolledBack`
- `StatusRollingBack`
- `StatusUnknown`

Extensions to the `Current` interface

`VisiTransact` has an extended interface that provides arguments for specifying an instance of the `VisiTransact Transaction Service`, as well as additional methods. See

“[Discovering an instance of the VisiTransact Transaction service](#)” for information on the `VisiTransactions::Current` arguments. The following table shows the methods in the VisiTransact-extended `Current` interface in the `VISTransactions.idl` file. For more information about the `Current` interface, see `Current` interface in the *VisiBroker for C++ API Reference*.

Method	Description
<code>begin_with_name()</code>	Enables its caller to pass a user-defined informational transaction name. For example, this helps with diagnostics because the user-defined transaction name is included in the value returned by the <code>get_transaction_name()</code> method. The name also helps with administration, because the Console will report the name in the detailed information about an outstanding transaction.
<code>get_txcontext()</code>	Returns a <code>PropagationContext</code> which can be used by one VisiTransact Transaction Service domain to export a transaction to a new VisiTransact Transaction Service domain.
<code>register_resource()</code>	Registers a Resource for a recoverable object. This method is a shortcut for using the Control and Coordinator objects to register a Resource for a recoverable object. It returns a Recovery Coordinator object that can be used to help coordinate recovery. Most applications will not normally call this method. See “ Coordinating transaction completion with Resource objects ” for information about Resources.
<code>register_synchronization()</code>	Registers a synchronization object. This method is a short-cut for using the Control and Coordinator object to register a Synchronization object. See “ Implementing Synchronization objects ” for details on Synchronization objects.
<code>get_otid()</code>	Provides the object transaction ID (<code>otid</code>) through the <code>Current</code> interface as a convenience. This avoids going to the Coordinator and looking through a <code>PropagationContext</code> . The <code>otid</code> is used to identify a transaction to a recoverable object. Most applications will not normally call this method.

7

Other methods of creating and propagating transactions

This section focuses on the other facilities available for managing transactions. It includes information on using the VisiTransact Transaction Service interfaces—`TransactionFactory`, `Control`, `Coordinator`, and `Terminator`.

Introduction

Although typically you will use the `Current` interface to manage transactions, there are several other approaches to transaction management you can use:

- **Indirect Context Management with Explicit Propagation.** The client uses a combination of the `Current`, `Control`, and other objects which describe the state of the transaction. A client application that uses the `Current` object (and therefore, is also automatically using implicit propagation) can use explicit propagation by gaining access to the `Control` object with the `Current::getControl()` method. It can use a `VisiTransact Transaction Service` object as an explicit parameter to a transactional object. This is explicit propagation.
- **Direct Context Management with Implicit Propagation.** The client uses a combination of the `Current`, `Control`, and other objects which describe the state of the transaction. A client that accesses the `VisiTransact Transaction Service` objects directly can use the `Current::resume()` method to set the implicit transaction context associated with its thread. This allows the client to invoke methods of objects that require implicit propagation of the transaction context.
- **Direct Context Management with Explicit Propagation.** The client application directly accesses the `Control` object and the other objects which describe the state of the transaction. To propagate the transaction to an object, the client must include the appropriate `VisiTransact Transaction Service` object as an explicit parameter of a method.

Managing transactions with these approaches means using these interfaces:

- `TransactionFactory`. This interface defines methods that allow a transaction originator to begin a transaction. To view the `TransactionFactory` interface, see [“Creating transactions with the TransactionFactory”](#).
- `Control`. This interface allows an application to explicitly manage or propagate a transaction context. To view the `Control` interface, see [“Gaining control of a transaction with the control object”](#).
- `Terminator`. This interface enables an application to commit or rollback transactions. Typically, its methods are used by transaction originators—however, by propagating the `Control` or `Terminator` object, any transaction participant can commit or rollback the transaction. To view the `Terminator` interface, see [“Committing or rolling back transactions with Terminator”](#).
- `Coordinator`. This interface enables a participant to determine the status of a transaction, discover the transaction name, obtain the transaction context, as well as designate that a transaction should be rolled back from a participant other than the transaction originator. See [“Marking a transaction for rollback”](#) and [“Obtaining transaction information”](#) for information on methods in the `Coordinator` interface.

Creating transactions with the TransactionFactory

The `TransactionFactory` interface is provided to allow the transaction originator to begin a transaction. As shown in the following example, this `CosTransactions` interface provides two methods—`create()` and `recreate()`. The `create()` method is used to start a new transaction. The `recreate()` method is used to create a transaction's `Control` object from a propagation context and is not typically used by a normal application.

```
module CosTransactions
{
    interface TransactionFactory
    {
        Control create(in unsigned long time_out);
        Control recreate(in PropagationContext ctx);
    };
};
```

`VisiTransact` also supplies an extension to the `TransactionFactory` interface that allows a transaction to be created using a specific name—`create_with_name()`. Naming a transaction is useful for tracking the progress of a particular transaction, as well as debugging its execution.

```
module VISTransactions
{
    // TransactionFactory
    // This extends the CosTransactions::TransactionFactory by
    // allowing someone to create a transaction with a user-defined
    // name that can be used for debugging, error reporting, etc.

    interface TransactionFactory : CosTransactions::TransactionFactory
    {
        CosTransactions::Control create_with_name(in unsigned long time_out,
            in string userTransactionName);
    };
};
```

The following table defines the methods for creating transactions with `TransactionFactory`.

Method	Description
<code>create(in unsigned long time_out)</code>	Creates a new transaction and returns a <code>Control</code> object which can be used to manage participation in the new transaction. If <code>time_out</code> is set to 0 seconds, the default timeout for the instance of the <code>VisiTransact Transaction Service</code> is used.
<code>create_with_name(in unsigned long time_out, in string userTransactionName)</code>	Creates a new transaction with a user-defined name as supplied in the <code>userTransactionName</code> argument.
<code>recreate(in PropagationContext ctx)</code>	Creates a new representation of an existing transaction as defined by the <code>PropagationContext</code> (the transaction context) and returns a <code>Control</code> object. The <code>Control</code> object can be used to manage or control participation in the existing transaction.

For more information about the `TransactionFactory` interface, see `TransactionFactory` interface in the *VisiBroker for C++ API Reference*.

The following example shows how to begin a new transaction that uses the default timeout period.

```
...
CosTransactions::TransactionFactory_var txnFactory;
CosTransactions::Control_var control;
control = txnFactory->create_with_name(0,"BankTransfer#1");
//use default
//timeout value
...
```

Note

The `PropagationContext` can be obtained from an existing transaction using the `CosTransactions:Coordinator::get_txcontext()` method described in [“Obtaining transaction information”](#).

Gaining control of a transaction with the control object

The `Control` interface allows an application to obtain the `Terminator` and `Coordinator` object references in order to explicitly manage or propagate a transaction context. An object supporting the `Control` interface is associated with one specific transaction.

The following example shows the `Control` interface.

```
module CosTransactions
{
  interface Control
  {
    Terminator get_terminator()
      raises(Unavailable);
    Coordinator get_coordinator()
      raises(Unavailable);
  };
};
```

The table below defines the methods for the `Control` interface.

Method	Description
<code>get_terminator()</code>	Returns a Terminator object which supports operations to end the transaction. The Terminator object can be used to rollback or commit the transaction associated with the Control object. The <code>CosTransactions::Unavailable</code> exception is raised if the Control object cannot provide the Terminator object.
<code>get_coordinator()</code>	Returns a Coordinator object which supports operations needed by Resources to participate in a transaction. The Coordinator object can be used to register Resources for the transaction associated with the Control object. The <code>CosTransactions::Unavailable</code> exception is raised if the Control object cannot provide the Coordinator object.

To obtain references to Terminator and Coordinator objects, you would include statements similar to those shown in the following example in your originator code. These objects are distinct because most methods only require one of them.

```

...
CosTransactions::Control_var control
CosTransactions::Terminator_var newTranTerminator;
CosTransactions::Coordinator_var newTranCoordinator;

newTranTerminator = control->get_terminator();
newTranCoordinator = control->get_coordinator();
...

```

Explicitly propagating transactions from the originator

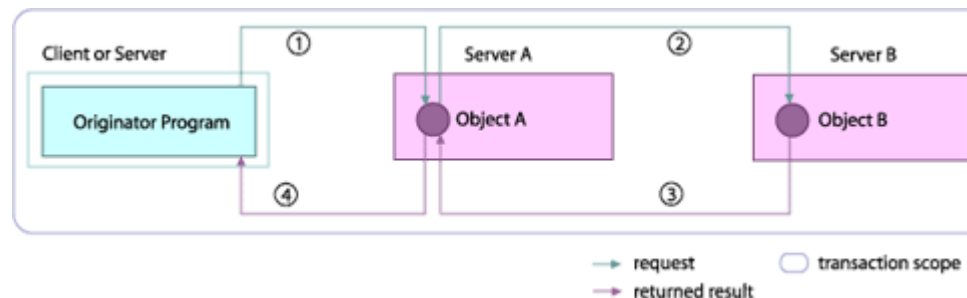
With transactions originated using the `TransactionFactory`, the transaction originator handles transactions using several VisiTransact Transaction Service interfaces. Through these interfaces, more than one transaction may be managed at a time by the transaction originator.

In these types of transactions, participants of a transaction share the same transaction context because the originator forwards the transaction context to each participant through an explicit parameter that is part of the IDL signature for all the operations. This means that the state of a transaction is maintained as the originator calls on other objects to perform actions, which may in turn call other objects using the same parameter. Note that the figure below shows the context being passed between transaction participants from within method calls.

Note

With transactions originated using the `TransactionFactory`, you can use implicit propagation. See [“Changing from explicit propagation to implicit”](#).

Figure 7.1 How the transaction context is explicitly propagated



- 1 The transaction originator requests that Object A performs the `doWork()` method, passing a Control object or Coordinator object.
- 2 Object A requests that Object B performs the `doMoreWork()` method, and also passes it the Control or Coordinator object, allowing Object B to operate as part of the existing transaction.

- 3 Object B returns its results to Object A.
- 4 Object A returns its results to the transaction originator.

To explicitly propagate a transaction to participants of a transaction, the originator must include the Control, Coordinator, or Terminator object as an explicit parameter to remote invocations of transactional objects.

- If you pass a Terminator object, you give the participant the limited ability to terminate the transaction—they cannot do anything else.
- If you pass a Coordinator object, you allow the remote object to be a participant in the transaction, but do not give the ability to terminate the transaction. Passing the Coordinator allows the remote object to ensure that the transaction is rolled back.
- If you pass a Control object, you give the participant the abilities of both the Coordinator and Terminator objects.

The example below shows the Control object, **control**, being passed as an explicit parameter to the `withdraw()` method of the remote transactional object.

```
...
CosTransactions::Control_var control;
CORBA:Boolean didSucceed;
didSucceed=bank->withdraw(10, 444, control) // invoke a CORBA request
...
```

Changing from explicit propagation to implicit

You may want to start a transaction with explicit propagation and then switch to implicit. To set up your implicit transaction context, pass the Control object into `Current::resume()`. See [“Using multiple transactions within a context or thread”](#) for details on using `Current::resume()` and `Current::suspend()`.

Getting the explicit context from Current

If you start a transaction with implicit propagation and later want to get the transaction context explicitly, use `Current::get_control()`.

Committing or rolling back transactions with Terminator

The `Terminator` interface supports operations to commit or rollback a transaction. Typically, these operations are used by the transaction originator. The following example shows the `Terminator` interface.

```
module CosTransactions
{
  interface Terminator
  {
    void commit(in boolean report_heuristics)
      raises (HeuristicMixed, HeuristicHazard);
    void rollback();
  };
};
```

The following table defines the methods provided by the `Terminator` interface.

Method	Description
<code>commit</code> (in boolean <code>report_heuristics</code>)	Commits the transaction if the transaction has not been marked as rollback only, and if all of the participants in the transaction agree to commit. Otherwise, the transaction is rolled back and the <code>CORBA::TRANSACTION_ROLLEDBACK</code> exception is raised. When the transaction is committed, all changes to recoverable objects made in the scope of the transaction are made permanent and visible to other transactions or clients. If the <code>report_heuristics</code> parameter is true, the VisiTransact Transaction Service will report inconsistent outcomes using the <code>CosTransactions::HeuristicMixed</code> and <code>CosTransactions::HeuristicHazard</code> exceptions.
<code>rollback()</code>	Rolls back the transaction. When a transaction is rolled back, all changes to recoverable objects made in the scope of the transaction are rolled back.

The next example shows the **MyBank** interface for the transactional object which the originator is accessing to perform actions.

```
#include <CosTransactions.idl>

interface MyBank {
    float    balance(in long accountNo,
                   in CosTransactions::Coordinator coord);
    boolean  withdraw(in long accountNo,
                    in float amount,
                    in CosTransactions::Control control);
};
```

The following example shows how an originator either commits or rolls back a transaction involving the **MyBank** transactional object. This example is specific for working with transactions in the `withdraw()` method. Note that the `balance()` method would not be allowed to terminate the transaction since it is only passed the `Coordinator`.

```
...
CORBA::Boolean didSucceed;
...
CosTransactions::Terminator_var
    txnTerminator=control->get_terminator();

if(didSucceed)
{ // invoke a CORBA request
    try
    {
        txnTerminator->commit(1);
    }
    catch(CORBA::TRANSACTION_ROLLEDBACK&)
    {
        // Return failure.
    }
}
else
{
    txnTerminator->rollback();
}
...

```

See [“Heuristic completion”](#) for details about heuristic completion when committing a transaction.

Marking a transaction for rollback

If the participant does not want the transaction to commit, it can use the `rollback_only()` method from the `Coordinator` interface. When the `rollback_only()` method is called by a participant, the transaction associated with the current thread is modified so that the only possible outcome is to rollback the transaction. The `CosTransactions::Inactive` exception is raised if the transaction has already been prepared. The example below shows how a participant would use the `rollback_only()` method.

```
...
CosTransactions::Coordinator_var coord;
coord->rollback_only();
...
```

Obtaining transaction information

A participant can obtain information about a transaction such as the transaction name or transaction status, or obtain the transaction context for a transaction using methods in the `Coordinator` interface. The following table describes these methods.

Method	Description
<code>get_status()</code>	Returns the status of a transaction associated with the current thread.
<code>get_transaction_name()</code>	Returns a printable string describing the transaction associated with the current thread.
<code>get_txcontext()</code>	Returns a <code>PropagationContext</code> object.

The `get_status()` method can return one of the following values:

- `StatusActive`
- `StatusCommitted`
- `StatusCommitting`
- `StatusMarkedRollback`
- `StatusNoTransaction`
- `StatusPrepared`
- `StatusPreparing`
- `StatusRolledBack`
- `StatusRollingBack`
- `StatusUnknown`

8

Transaction completion

This section provides information about transaction completion, explains heuristic completion, and provides information necessary for multithreaded applications.

Transaction completion

Transaction completion is a sequence of steps that the VisiTransact Transaction Service goes through when it receives a request to either commit or rollback the work of a transactional application. The request for completion can be initiated in different circumstances:

- The transaction originator initiates completion by invoking either `commit()` or `rollback()`.
- A transaction timeout occurs and triggers completion.
- During recovery of the VisiTransact Transaction Service, incomplete transactions (found in the log records) are reinstated and transaction completion is resumed.

How does the VisiTransact Transaction Service ensure completion?

When a transaction originator requests to commit or roll back a transaction, the VisiTransact Transaction Service initiates the completion procedure for the transaction. Assume there are two Resources that are involved in a single transaction. When a request to commit arrives, the VisiTransact Transaction Service will initiate a two phase commit procedure to coordinate the completion.

If the transaction completes by successfully executing the two phase commit procedure without failure, the originator is notified with the outcome. If the transaction cannot complete due to a specific reason—for example, when one of the Resources is not available in the commit phase of the two phase procedure—the VisiTransact Transaction Service cannot complete the transaction and it will place the transaction in a Retry Queue for later attempts. When a transaction is placed in the Retry Queue, the transaction is not dispatched immediately for completion. There is a programmed delay between each retry attempt to prevent degradation of system performance. The minimum time between retry attempts is 15 seconds and the maximum is 900 seconds. The first retry attempt will start after 15 seconds and for the subsequent attempts, the delay is increased until it reaches 900 seconds. After that, the retry attempts are every 900 seconds. If a retry attempt is made due to a timeout or a recovery, the first retry

attempt is dispatched immediately without waiting for the 15 seconds delay. During retry attempts, the VisiTransact Transaction Service executes only those portions of a transaction that have not yet been completed. The transaction remains in the Retry Queue until it completes or until a “Stop Completion” command is issued by the VisiBroker Console. If you query from the Console for a list of transactions, the transactions with several retry attempts are highlighted.

Retry attempt scenarios are as follows:

- **A transaction timeout occurs.** If a timeout period has been specified for a transaction and the transaction does not complete within this limit, the transaction is placed in the Retry Queue. If the transaction has entered the completion stage when the timeout expires, the timeout will be ignored by the VisiTransact Transaction Service. You can set the default timeout for the VisiTransact Transaction Service at the command line.
- **A Resource is unavailable.** A Resource that is involved in the transaction is temporarily not available due to a communication failure or because the Resource server is down. The transaction is placed in the Retry Queue until it completes.
- **The VisiTransact Transaction Service recovers and decision records show transactions are incomplete.** During recovery, the information is gathered from the transaction log about the transactions that were incomplete when the VisiTransact Transaction Service went down. If the decision records indicate that the transaction has not completed yet, they are placed in the Retry Queue for completion.

The number of times a transaction has to be retried can be configured using the property:

```
vbroker.ots.completionRetryAttempts=n.
```

By configuring the property `vbroker.ots.completionRetryAttempts` and setting it an integer value "n", you can change the number of retries.

If n is set a value less than or equal to 0, transaction is retried for ever, which is the default behaviour. When set to a value greater than 0, transaction is put in retryQ and during each attempt it will try to run the transaction to completion. On failure, the transaction is put in the retry queue no more than n number of times.

How does the VisiTransact Transaction Service ensure checked behavior?

The VisiTransact Transaction Service implements full Distributed Transaction Processing (DTP) checked behavior to provide an extra level of transaction integrity. Checked behavior protects against loss to data integrity by ensuring that all transactional requests made by the application have completed their processing before the transaction is committed. This guarantees that a commit will not succeed unless all transaction participants have completed the processing of their transactional requests. Checked behavior occurs by default when all requests are synchronous.

Checked behavior is enforced for VisiTransact-managed transactions involved with deferred synchronous requests: transactions are rolled back if there are pending replies when a `commit()` is issued. If the request handler of a transactional object makes a deferred synchronous request and replies before the deferred synchronous request returns, the transaction is marked for rollback.

VisiTransact does not enforce checked behavior on one-way requests.

The example below shows the client code for checked behavior when you have a deferred synchronous request and the reply returns *after* `commit()` is invoked. Checked behavior is successful—the transaction is rolled back.

```
...
// get reference to the Current
...

// begin a transaction
current->begin();
```

```

// create a dynamic request
CORBA::Request_var bankRequest = bank->_request("withdraw");
CORBA::NVList_ptr arguments = bankRequest->arguments();

CORBA::Any_var    amt = new CORBA::Any();
*amt<<= ((float)1000.00);

arguments->add_value("amount", amt, CORBA::ARG_IN);

...

//invoke deferred synchronous request
bankRequest->send_deferred();

//forget to get the response
// commit the txn
try
{
    current->commit(0);
}
catch(CORBA::TRANSACTION_ROLLEDBACK& e)
{
    cerr << "SUCCESS, commit check worked()" << endl;
}
...

```

The example below shows the client code for checked behavior when you have a deferred synchronous request and the reply returns before `commit()` is invoked. Checked behavior is successful—the transaction is committed.

```

...
// case where request arrived before commit
current->begin();

cerr << " === Invoking a dii deferred sync request" << endl;
bankRequest->send_deferred();

try {
    //wait for reply
    bankRequest->get_response();
    current->commit(0);
}
catch(CORBA::TRANSACTION_ROLLEDBACK& e)
{
    cerr << "FAILURE, TRANSACTION_ROLLEDBACK not expected" << endl;
}
}

```

Heuristic completion

Heuristic completion is when a transaction attempts to complete and one of its participating Resources makes a heuristic decision during the completion stage. An heuristic decision is a unilateral decision made by one or more Resources to commit or rollback updates without regard to the outcome determined by the transaction manager.

Heuristic decisions typically only occur during unusual circumstances that prevent normal processing, such as a network failure, or if the coordinator does not complete the two-phase commit process in a timely manner. When a heuristic decision is made

there is a risk the decision is different from the outcome determined by the transaction manager, resulting in a loss of data integrity.

The types of heuristic outcome exceptions that are returned by the resources are:

- `HeuristicRollback` - The commit operation on Resource reports that a heuristic decision was made and that all relevant updates have been rolled back.
- `HeuristicCommit` - The rollback operation on Resource reports that a heuristic decision was made and all relevant updates have been committed.
- `HeuristicMixed` - The Resource has committed some relevant updates, and rolled back others.
- `HeuristicHazard` - The Resource does not know the result of at least one relevant update (the disposition of all relevant updates is not known). For the updates that are known, either all have been committed or all have been rolled back.

A Resource can make a heuristic decision at any point during two-phase commit. For example, if the Terminator does not complete the two-phase commit in a timely manner, a Resource can elect to make a heuristic decision. A heuristic decision is a way that a Resource object can break guarantees it made during the two-phase commit process (that is, when it returned `VoteCommit` during `prepare()`).

However, if a Resource has replied `VoteCommit` to the Terminator, and then subsequently makes a heuristic decision, it is still responsible for reporting its action regarding the transaction. The following may occur when the Terminator eventually requests that the Resource rollback or commit:

- **The heuristic decision may be consistent with the outcome.** If this is the case, the transaction can be completed normally, and the Resource may “forget” about the transaction and the heuristic decision. The Terminator does not need to be informed of the heuristic decision since it was consistent with the outcome of the transaction.
- **The heuristic decision may differ from the outcome.** In this case, the Resource consults its record of the heuristic outcome (which it previously placed in the stable storage), and returns one of the heuristic outcome exceptions (`HeuristicCommit`, `HeuristicRollback`, `HeuristicMixed` or `HeuristicHazard`) when completion continues.
- The heuristic outcome details must be retained in stable storage until the Resource is instructed to “forget” the transaction by the Coordinator.

Enabling heuristic reporting to your application

A transaction originator can request to receive heuristic reporting by setting the `report_heuristics` parameter of the `commit()` method to `true`. Notice the following code sample shows the `commit()` method as `commit(1)` for C++.

Handling heuristic reporting

```
...
if (bank->withdraw(10,444))    //invoke the withdraw method.
{
    try
    {
        current->commit(1);    //The parameter 1 requests the server to
                               //return the heuristic outcomes if there are
any.
    }
    catch (const CosTransactions::NoTransaction& e)
    {
        //commit was issued when there is no transaction
        //Handle it.
    }
    catch (const CosTransactions::HeuristicMixed& e)
```

```

        {
            //Heuristic decision was made. Some of the relevant updates
            //have been committed and others may have rolled back.
            //Handle it.
        }
    catch (const CosTransaction::HeuristicHazard& e)
    {
        //Heuristic decision was made. The relevant updates that
        //have been made either all have been committed or rolledback.
        //Handle it.
    }
}
else
{
    current->rollback();
}

```

A Resource can handle heuristic reporting programmatically, or can require the intervention of a system administrator.

OTS exceptions

Additional OTS exceptions are:

- **SubtransactionsUnavailable** – This exception is raised if the client thread already has an associated transaction and the transaction service implementation does not support nested transactions.
- **NotSubtransaction** – This exception is raised if the current transaction is not a subtransaction.
- **Inactive** – This exception is raised in a few scenarios whereby no action is taken when the current context is not right for the command issued.
- **NotPrepared** – This exception is raised when a transaction is not prepared (for two-phase commit transactions only).
- **NoTransaction** – This exception is raised when there is no transaction associated with the client thread.
- **Unavailable** – This exception is raised when the application can't get hold of propagation context.
- **SynchronizationUnavailable** – This exception is raised if the system does not support synchronization.

Heuristic completion

9

Coordinating transaction completion with Resource objects

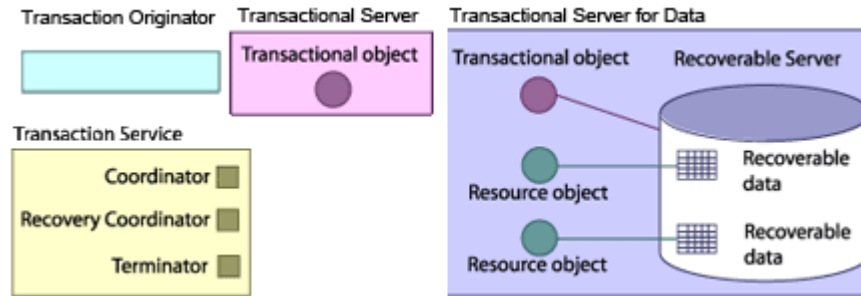
This section provides information about how you can participate in one- and two-phase commits using Resource object(s).

Understanding transaction completion

The transaction process described in [“Model for a basic transaction”](#) was a simple example that did not involve data. The following diagram expands on that earlier example to show the objects that are necessary when transactions involve data—Recoverable Server, Recoverable Object, Recovery Coordinator, and Resource object. In practice, several of these objects are encapsulated by transactional software for data, as shown in the figure below. These objects are shown so that you will understand the process going on underneath, and will recognize these interfaces in the IDL.

If you are using VisiTransact-managed transactions, this diagram also shows you the back-end objects (Coordinator, Terminator, and Recovery Coordinator) that the VisiTransact Transaction Service uses to perform the work of two-phase commits. You only manage these object directly if you are not using VisiTransact-managed transactions.

Figure 9.1 Objects involved in two-phase commit



The table below describes the objects involved in two-phase commit.

Object	Description
Coordinator	Facilitates registration of recoverable objects with a transaction, and manages coordination between transactions.
Terminator	Coordinates the termination of a transaction—it ensures that participants either all commit or all rollback the work of a transaction.
Recoverable Server	A collection of one or more objects. At least one of the objects is recoverable.
Recoverable data	Data, such as a table in a database, whose content is affected by the completion of the transaction. Not all recoverable objects will implement a CORBA interface.
Resource Object	Represents the relationship between the VisiTransact Transaction Service and a recoverable object for the life of a transaction. One Resource object is required for each recoverable object participating in a transaction.
Transactional Software for Data	A collection of objects (Recoverable Server, Recoverable Data, Transactional Object, and Resource Object) used to access data either in a database, file system, or other service such as the VisiBroker Naming Service.
Recovery Coordinator	Used in case of failure by Resource objects to determine the outcome of the transaction, and to coordinate the recovery process with the VisiTransact Transaction Service.

Participating in transaction completion

“[Completing a transaction](#)” is where two-phase commit diverges from the simple example addressed in “[Model for a basic transaction](#)”. When the VisiTransact Transaction Service executes the two-phase commit process, it ensures that the entire transaction is either rolled back or committed atomically. In the first phase of the two-phase commit process, the Terminator asks the participants of the transaction if they can prepare the transaction to commit. If all participants vote that they can, the Terminator then tells all participants to commit the transaction during the second phase. If at least one participant votes that it is not prepared to commit, the Terminator instructs the participants to rollback the transaction.

Note

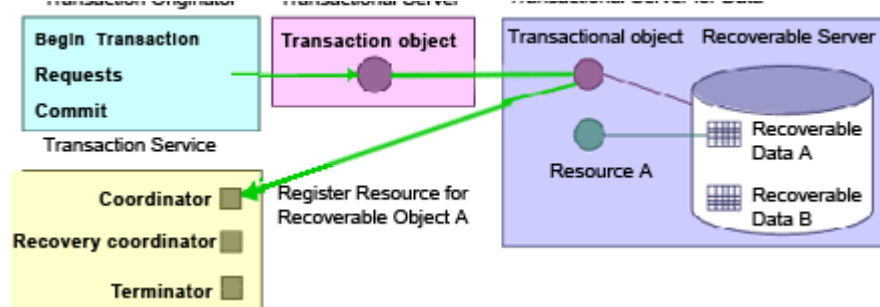
If a transactional application only involves one Resource, the VisiTransact Transaction Service initiates a one-phase commit process rather than a two-phase commit process.

The following sections expand on the concept of completing a transaction to explain the process of two-phase commit.

Resource object is registered for the transaction

Resource objects must be registered for all recoverable data involved in the transaction. The transactional object registers the Resource with the transaction's Coordinator for the recoverable data.

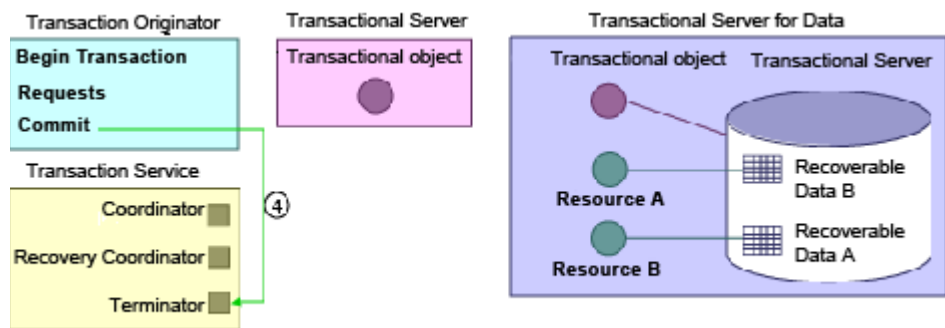
Figure 9.2 Registering a Resource object for recoverable data



Transaction originator initiates transaction completion

The transaction originator notifies the Terminator that it wishes to complete the transaction, which initiates the two-phase commit process with the VisiTransact Transaction Service. This step replaces step 4 in "Completing a transaction".

Figure 9.3 Transaction completion initiated by the transaction originator

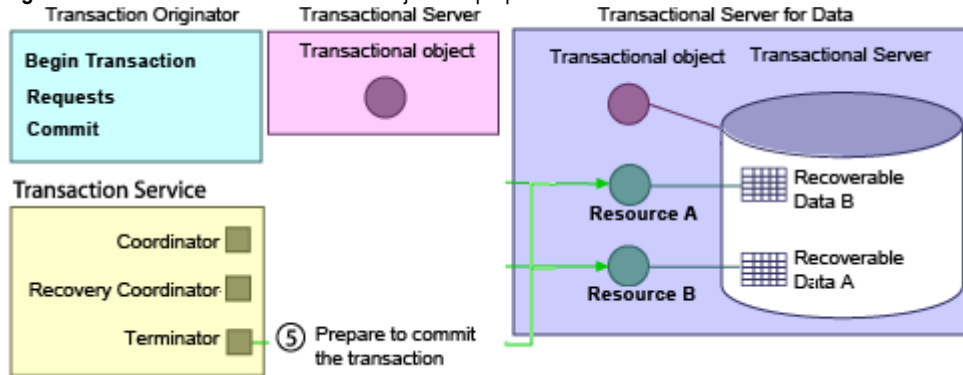


In this step, the same action is taking place, but you see behind the scenes that the invocation of `commit()` is actually handled by the Terminator.

Terminator tells Resource objects to prepare

Once the Terminator receives notice that the transaction originator wishes to commit the transaction, the Terminator contacts all Resource objects participating in the transaction, and notifies them they must prepare to commit the transaction. To do so, the Terminator invokes the `prepare()` method on all Resource objects registered with the transaction.

Figure 9.4 Terminator asks Resource objects to prepare to commit the transaction



Note

If only one Resource is registered with the Coordinator, the Terminator performs a one-phase commit as an optimization. To do so, it invokes `commit_one_phase()` on the Resource rather than invoking `prepare()` and then `commit()`.

Any exception that occurs during the prepare phase causes a rollback of the transaction.

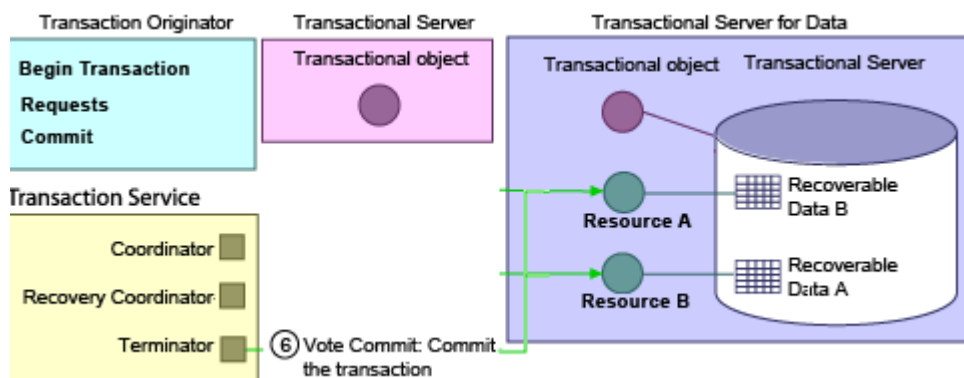
Resource objects return a vote to the terminator

When Resource objects are told to prepare, they respond to the Terminator with a vote:

- **VoteCommit** means the Resource guarantees that it can commit the transaction when asked, even if there is a failure after `prepare()`.
- **VoteRollback** means the Resource requires the transaction to rollback, and is proceeding to rollback its own data.
- **VoteReadOnly** means that the Resource does not have persistent data affected by the transaction. There, it is independent of the two-phase commit—the two-phase commit does not affect its state.

If the Resource returns **VoteRollback** or **VoteReadOnly**, it will not be contacted again by the VisiTransact Transaction Service, and can safely destroy itself. For this example, let's assume that both Resource A and Resource B return **VoteCommit**.

Figure 9.5 Resources return a vote to the Terminator



Terminator decides whether to commit or roll back

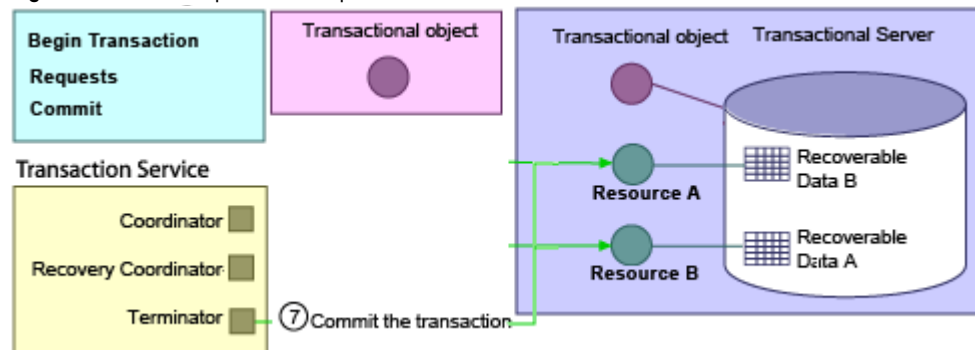
Based on the votes received by the Resource objects, the Terminator determines whether the transaction will be committed or rolled back. At this point, the completion

decision is made and logged. If any of the Resource objects return `VoteRollback`, raise exceptions, or invoke `rollback_only()`, the transaction will be rolled back by the Terminator.

If the transaction decision was to rollback, the Terminator invokes `rollback()` on all Resources—except those that returned `VoteRollback` or `VoteReadOnly`. If the decision is to commit, the Terminator invokes `commit()` on all Resources, and the two-phase commit process is finished.

For this example, both Resource objects involved with the transaction returned `VoteCommit`, so the Terminator object requests that the Resource objects commit the transaction.

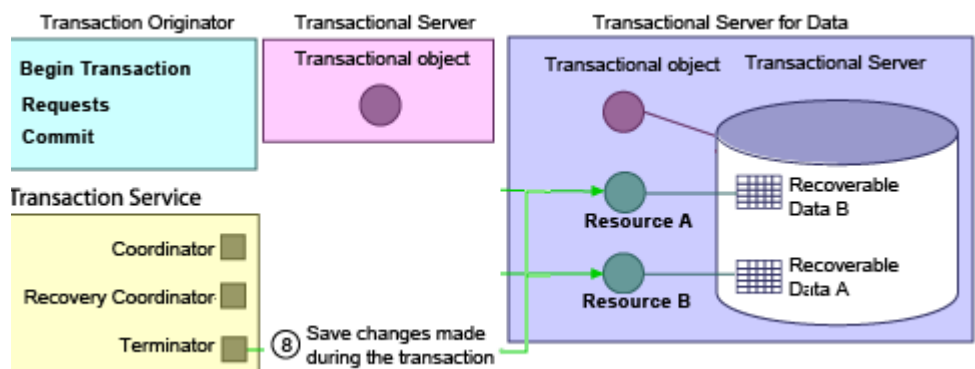
Figure 9.6 Second phase of two-phase commit is initiated



Resource objects vote to commit the transaction

When a Resource object commits a transaction, it makes any data changed by the transaction visible to all readers of the data—the data stored by the recoverable object is changed according to the outcome of the transaction. Also, the Resource object stores other information in case of failure. Lastly, once the transaction has been committed, all objects associated with the transaction are removed (i.e. the Coordinator, Terminator, and Recovery Coordinator).

Figure 9.7 Resource objects commit the changes made during the transaction



Summary of steps for two-phase commit

As shown by the previous sections, the steps for two-phase commit are:

- 1 Resource objects are registered for the transaction.
- 2 Transaction originator initiates transaction completion.
- 3 Terminator tells Resource objects to prepare.
- 4 Resource objects return a vote to the Terminator.
- 5 The Terminator decides whether to commit or rollback.

- 6 The Terminator tells Resource objects to commit or rollback.

Summary of steps for single-phase commit

The steps for a single-phase commit are:

- 1 Resource object is registered for the transaction.
- 2 Transaction originator initiates transaction completion.
- 3 Terminator tells the Resource object to commit one phase.
- 4 Resource object returns a vote to the Terminator.
- 5 The Terminator decides whether to commit or rollback.
- 6 The Terminator tells the Resource object to commit or rollback.

Summary of steps for a rollback

The steps for a rollback are:

- 1 Resource objects are registered for the transaction.
- 2 Transaction originator initiates transaction completion.
- 3 Terminator tells Resource objects to rollback.

Participating in transaction recovery after failure

If the VisiTransact Transaction Service (or its host) experiences a failure once the decision to commit the transaction has been logged, the Terminator proceeds to invoke `commit()` on all Resources once the VisiTransact Transaction Service and participating Resource objects are running again.

If the decision was to rollback, and the VisiTransact Transaction Service (or its host) experiences a failure, the VisiTransact Transaction Service considers the transaction to be rolled back once it is running again. This is because the VisiTransact Transaction Service does not keep track of Resources when a transaction is marked for rollback, and therefore it cannot proactively tell Resources to rollback. Instead, Resources must use the Recovery Coordinator (specifically, the `replay_completion()` method) to find out that the transaction rolled back.

If a VisiTransact Transaction Service fails before a Resource object has committed but after it has been prepared and the VisiTransact Transaction Service has not yet logged the decision, then the Resource is responsible for contacting the Recovery Coordinator and initiating transaction completion.

If a failure occurs and the Terminator cannot reach a registered Resource, the Terminator must keep trying to contact the Resource until it can be reached. In this way, atomic transactions are guaranteed because Resource objects will be restarted, and the VisiTransact Transaction Service will ensure that recoverable objects complete the transaction in conformance with the outcome.

These basic rules apply to transaction recovery following a failure of the VisiTransact Transaction Service:

- If the decision to commit the transaction has been logged, the Terminator invokes `commit()` on all Resources, and the two-phase commit process is finished.
- If the Terminator only contains heuristic information, nothing happens.
- If the transaction is marked for rollback before the failure, it is lost and therefore rolled back.
- If a registered Resource exists but cannot be reached, the Terminator must keep trying to contact the Resource until it can be reached.

10

Managing heuristic decisions

This chapter provides information about the heuristic decisions that you must manage for your transactional applications.

What is a heuristic decision?

A heuristic decision is a unilateral decision made by one or more transaction participants to commit or rollback updates without first obtaining the consensus outcome determined by the VisiTransact Transaction Service. Heuristic decisions are typically made in unusual circumstances, such as communication failures, that prevent normal processing. When a heuristic decision is made, there is a risk that the decision will differ from the consensus outcome, resulting in a loss of data integrity.

The types of heuristic outcome exceptions that can be returned are:

- `HeuristicRollback`. The participant rolled back all relevant updates.
- `HeuristicCommit`. The participant committed all relevant updates.
- `HeuristicMixed`. The participant has committed some relevant updates, and rolled back others.
- `HeuristicHazard`. The participant does not know the result of at least one relevant update.

For more information about heuristic decisions and exceptions, see [“Transaction completion.”](#)

What is the heuristic.log file?

VisiBroker VisiTransact produces one heuristic log per instance of the VisiTransact Transaction Service, located by default in `<VBROKER_ADM>/its/transaction_service/<transaction_service_name>/heuristic.log`. This log is saved in text format and can be viewed, but should not be edited. The heuristic log contains records for any heuristically-completed transaction associated with the VisiTransact Transaction Service instance.

A heuristic log record contains information which is global to the transaction:

- **Exception.** The exception that was reported back to the transaction originator, if requested. This appears before the Transaction Info portion of the log record; for example: `CosTransactions::HeuristicHazard Exception`.
- **Transaction Name.** The name of the transaction (either user-defined, or assigned by the VisiTransact Transaction Service). This appears in the name field of the Transaction Info portion of the log record; for example, `Update_Inventory_Database`.
- **Transaction Identifier.** The ASCII version of the transaction identifier (`otid`). This appears in the id field of the Transaction Info portion of the log record.
- **Host of Transaction Originator.** The IP address of the host machine where the transaction originator is located. This appears in the host field of the Originator Info portion of the log record.

The heuristic log also contains the following information for each Resource in the Participant Info sections of the log record:

- **Resource Name.** The name of the Resource object registered with the VisiTransact Transaction Service instance that made the heuristic decision. This appears in the name field.
- **Resource Host.** The IP address of the host on which the Resource is located. This appears in the host field.
- **Resource IOR.** The interoperable object reference (IOR) of the Resource. This appears in the ior field.
- **Resource Vote.** The vote sent by the Resource when asked to prepare to commit. This appears in the voteForPrepare field.
- **Resource Decision.** The heuristic decision made by the Resource (for example, `OutcomeHeuristicHazard` or `OutcomeHeuristicMixed`). This appears in the outcome field.

You can archive the heuristic log file by moving it to a different location, if you have the appropriate file permissions. If you do this, the next time a heuristic occurs, the VisiTransact Transaction Service instance will recreate the heuristic log file. You can also make backup copies of the log by copying the log file to an alternate location. This is useful if you want to keep a daily backup of the heuristic log for your records.

Caution

Do not edit the `heuristic.log` file.

Interpreting the heuristic log

Assume that a transaction named `Update_Inventory_Database` has begun. Two Resources are registered with this transaction—**inventory** and **customer**. As part of the transaction completion, these Resources are asked to prepare to commit the transaction, and both Resources return a vote of `VoteCommit`. The VisiTransact Transaction Service then requests that the Resources commit the transaction. The **customer** Resource commits successfully and returns, but the **inventory** Resource makes a heuristic decision and returns with an exception of `HeuristicHazard`.

The following heuristic log shows what would appear in the heuristic log for this transaction. Notice that the exception returned to the transaction originator is `CosTransactions::HeuristicHazard`. The boldface type marks the location of the information described in [“What is the heuristic.log file?”](#). Extra white space has been added to the example heuristic log entry for easier viewing.

```
06/02/98, 14:43:43.587, gemini, /net/gemini/vsi2/its/dev/jmitra/vbroker/adm/./
bin/ots,
>None, 0, 0, Error, TransactionService, 4004,
at 0X000001, 0X04110FA4, 896823823, >587
CosTransactions::HeuristicHazard Exception:
```

```
Transaction Info:
name = Update_Inventory_Database
Id =
_56495349_01000000_ce400ff2_0000cac1_67656d69_6e695f6f_74730000_00000000_000
00000_00000000_00000000_00000000_3574720f_0000e845_00000000_00000000
```

```
Originator Info:
host = 206.64.15.75
```

```
Participant Info:
name = inventory
host = 206.64.15.75
ior =
IOR:002020200000002549444c3a73797374656d5f746573742f44756d6d7950617274696369
70656e743a312e3000202020000000100000000000006200010000000000d3230362e3634
2e31352e373500000f730000004600504d4300000000000002549444c3a73797374656d5f74
6573742f44756d6d795061727469636970656e743a312e300000000000000e746573745265
736f757263653100
voteForPrepare = VoteCommit
outcome = OutcomeHeuristicHazard
```

```
Participant Info:
name = customer
host = 206.64.15.75
ior =
IOR:002020200000002549444c3a73797374656d5f746573742f44756d6d7950617274696369
70656e743a312e3000202020000000100000000000006200010000000000d3230362e3634
2e31352e373500000f730000004600504d4300000000000002549444c3a73797374656d5f74
6573742f44756d6d795061727469636970656e743a312e300000000000000e746573745265
736f757263653200
voteForPrepare = VoteCommit
outcome = OutcomeNone
```

What to do once the problem has been isolated

Once you've determined the nature of the problem by looking through the heuristic log, you can do several things to correct the problem.

The first thing to do is to match the transaction name and transaction identifier found in the heuristic log with the transaction identifier in the log on the Resource side (that is, the database log). Once you've located the problem, you can correct it manually on the Resource side. For example, as described in ["Interpreting the heuristic log"](#) you would locate Update_Inventory_Database in the Resource log, and take steps to manually commit the changes to the **inventory** Resource.

What to do once the problem has been isolated

11

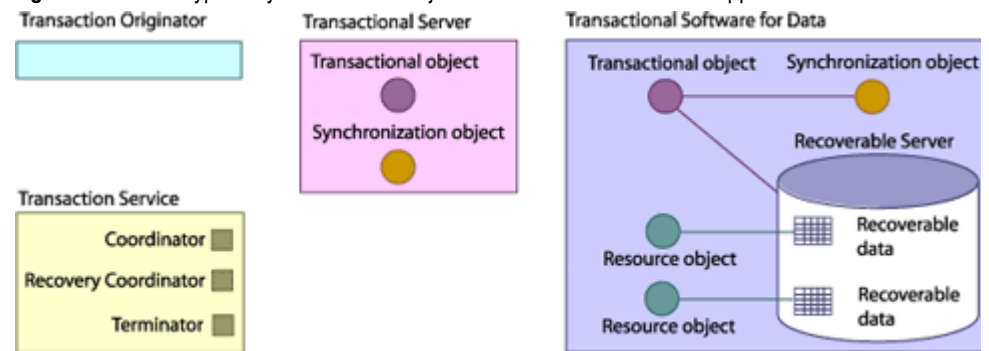
Implementing Synchronization objects

This section provides information about how you can implement Synchronization objects.

What are Synchronization objects?

A Synchronization object enables an object to be notified before the start of, and after the end of, the transaction's completion. The figure below shows how typical Synchronization objects fit into the architecture of a transactional application.

Figure 11.1 How typical Synchronization objects fit within the transactional application



The `before_completion()` method is invoked after the application invokes `commit()`, but before the VisiTransact Transaction Service begins transaction completion. The `before_completion()` method is not invoked for a rollback request. The `after_completion()` method is always invoked during normal processing.

Synchronization objects are not recoverable. If an instance of the VisiTransact Transaction Service fails, Synchronization objects will not be contacted.

Using Synchronization objects before the commit protocol

With the `before_completion()` method, Synchronization objects can perform processing after the work of a transaction has been done, but before the commit protocol starts (i.e. before `prepare()` or `commit_one_phase()`). For example, you can:

- **Improve performance.** You can cache changes during interactions with a transactional object, and then use the Synchronization object to flush the changes to disk, and even register a Resource. The advantage is that you do not have a Resource object or an open database connection until you need one.
- **Trigger additional work.** For example, you can write a record to an audit database and register that database as a Resource from the Synchronization object.
- **Check the transaction's integrity.** You can verify that all of the necessary operations were performed. For example, you might verify that the balance of an account was updated, and that the balance change was recorded in a history table.

Using Synchronization objects after rollback or commit

With the `after_completion()` method, Synchronization objects can do work after the transaction has been completed; that is, after the Terminator tells Resources to `commit()`, `rollback()`, or `commit_one_phase()`. You might use Synchronization objects to perform the following types of actions:

- **Perform cleanup.** For example, you might release memory objects.
- **Notify other processes of the transaction completion.** For example, the Synchronization object might send the results of the transaction as an event to an event channel, or communicate the results of the transaction to another object whose processing depends on the outcome of the transaction. The status condition is either `StatusCommitted` or `StatusRolledBack`.

Registering Synchronization objects

You register a Synchronization object with the `CosTransactions::Coordinator` using one of the following methods:

- `CosTransactions::Coordinator::register_synchronization()`
- `VisiTransactions::Current::register_synchronization()`

Irrespective of whether the transactional application uses VisiTransact-managed or explicitly propagated transactions, the VisiTransact Transaction Service uses implicit propagation to pass transaction information to Synchronization objects.

When a Synchronization object has been registered and a request to commit the transaction is made, the Terminator automatically invokes `before_completion()` on any Synchronization objects before actually performing the completion. You determine what happens during the `before_completion()` invocation from within your Synchronization object. When all registered Synchronization objects have completed, the Terminator proceeds with its transaction completion. A rollback can be ensured by invoking `rollback_only()` (on the `VisiTransactions::Current` or `CosTransactions::Coordinator`) from a `before_completion()` method. Additionally, any exception thrown by a `before_completion()` method (including `CORBA::TRANSACTION_ROLLEDBACK`) will cause the transaction to be rolled back.

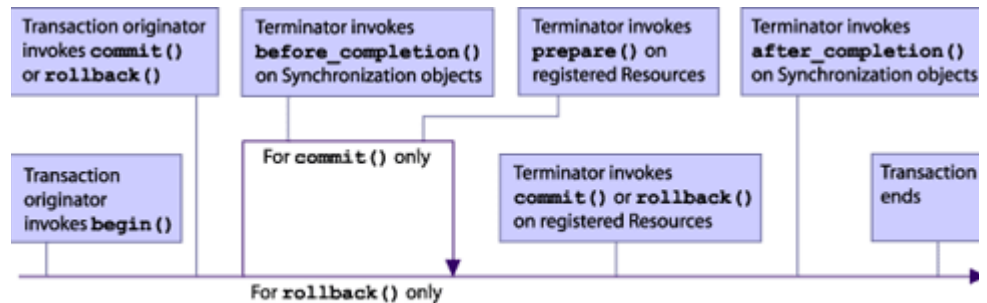
If any of the Synchronization objects mark the transaction for rollback, the Terminator stops invoking `before_completion()` on the remaining Synchronization objects. Because any Synchronization object can invoke `rollback_only()`, invoking `commit()` does not guarantee the transaction will commit.

The next time the Terminator interacts with Synchronization objects is after transaction completion; that is, it has received all `commit()`, `commit_one_phase()`, or `rollback()` responses from Resource objects. At this time, the Terminator automatically invokes `after_completion()` on all registered Synchronization objects and passes them the

transaction outcome as status. You determine what happens during the `after_completion()` invocation from within your Synchronization object.

The following figure shows a time line for the various invocations during the two-phase commit process when Synchronization objects involved.

Figure 11.2 Time line for two-phase commit with Synchronization objects



How failures affect Synchronization objects

If a Synchronization object is unavailable when the Terminator attempts to invoke its `before_completion()` method, the transaction will be rolled back. Any Synchronization objects that have not been contacted will not have `before_completion()` invoked on them. If any Synchronization object is unavailable when the VisiTransact Transaction Service tries to invoke `after_completion()`, it is ignored.

When the VisiTransact Transaction Service instance recovers, it does not remember Synchronization objects, and will only replay completion and not Synchronization objects.

The role of Synchronization objects in transaction objects

If you want your transactional object to be notified of the outcome of a transaction, it must provide a `Synchronization` interface. The VisiTransact Transaction Service notifies Synchronization objects of how a transaction completed when it invokes the `after_completion()` method.

12

Backward compatibility and migration

Backward compatibility

OTS1.1 Clients vs OTS1.4 Servers

OTS1.1 clients can safely call methods on the objects at OTS1.4 servers provided that those server objects have ADAPTS OTS policy values in their IORs.

If the objects obtained from OTS1.4 servers have REQUIRES OTS policy values in their IORs, any invocation on those objects must happen within the scope of an active transaction. Also, if the objects obtained from OTS1.4 servers have FORBIDS OTS policy values in their IORs, any invocation on those objects must happen outside the scope of an active transaction.

OTS1.1 Servers vs OTS1.4 Clients

OTS1.4 clients can work well with OTS1.1 servers, as if they are OTS1.1 clients. However, OTS1.4 clients do not unconditionally propagate the transaction context, in contrast to the OTS1.1 clients.

In cases where call-backs are used, the call-back object that an OTS1.4 client passes to an OTS1.1 server must be of type `TransactionalObject` if the client wants the transaction context to be propagated with the call back from the OTS1.1 server.

Migration

This section describes migration from traditional definition of Transactional object to one using of polices

Transactional Objects that are created using version VBE 5.1 or earlier, will be using the definition of inheriting `CosTransactions::TransactionalObject` interface. In order to take advantage of the ability to control the transactional behaviour, it is needed to migrate use the approach of defining of `VisiTransact` policies.

The steps are as follows:

- 1 Remove the `TransactionalObject` interface from all your idl files. Use proper `OTSPolicy` values to control transactional requirements for all target objects. Unshared transaction model is not supported in this release, so for `InvocationPolicy`, only `SHARED` and `EITHER` are meaningful. Users can choose not to set explicit values for this policy; in that case `VisiTransact` will set the `InvocationPolicy` for each target object with a value of `EITHER`. However, users are free to set `InvocationPolicy` for a target object and `VisiTransact` will check its validity against the `OTSPolicy` value.
- 2 Use proper `NonTxTargetPolicy` at client side to control the invocations on non-transactional objects.
- 3 Use `CORBA::ORB::create_policy()` method to create corresponding policies.
- 4 Compile the code with new `VisiTransact` libraries.

13

Session Manager overview

This chapter explains the general process for integrating databases with an VisiTransact-based application. It describes the Session Manager and XA Resource Director in detail.

Note

Session Manager is supported on the Solaris platform only.

How are databases integrated into a VisiTransact application?

VisiBroker VisiTransact enables easy integration of a DBMS with the VisiTransact Transaction Service, an application, and transactional objects. The Session Manager and its associated Resources provide complete transactional access to the DBMS. Full two-phase commit capability is supported by the XA implementations of the Session Manager and its Resource implementation (the XA Resource Director). Alternatively, the DirectConnect version of the Session Manager provides optimized transactional access to a single database using an integrated Resource, but require a more restrictive programming model. The Session Manager is always deployed by being embedded in your application programs.

The Session Manager is an implementation of a pseudo-IDL interface allowing C++ applications to obtain pre-configured database connections. The Session Manager insulates applications from the database-specific requirements for connection handles, thread management, transaction association, and Resource registration. Once a connection is obtained using the Session Manager, the transaction is coordinated automatically by the VisiTransact Transaction Service. The application developer is free from creating code that incorporates the database's participation in the transaction—the application code only needs to address issues concerning the data access it requires from the database.

Note

VisiBroker VisiTransact's DBMS integration strategy is part of a larger integration strategy. You can also integrate VisiTransact with systems that use popular transaction processing monitors (Tuxedo, CICS, and IMS) and messaging software (MQSeries) on many platforms including mainframes.

Currently, the Session Manager provides connectivity with Oracle9i, and the Pluggable Resource Interface allows you to create session management for the database of your choice. See [“Pluggable Database Resource Module for VisiTransact”](#) for more information about the Pluggable Resource Interface.

What is the Session Manager?

The Session Manager is a multi-function component that provides an interface for managing transactional database connections.

Note

To use the VisiTransact Transaction Service you are not required to use the Session Manager. Other vendors may provide a component that performs the same functionality as the Session Manager. The VisiTransact Transaction Service works with any comparable component that is compliant with the OMG Transaction Service specifications. If you choose to use the XA implementation of the Session Manager, you must use the XA Resource Director too—they are interdependent. Currently, the Session Manager and XA Resource Director work only with the VisiBroker VisiTransact Transaction Service.

The Session Manager provides the following functions:

- Opens a connection to a specific type of database or obtains an open connection from its connection pool.
- Associates the connection with the current transaction context.
- Registers the appropriate Resource with the Coordinator. Registers the XA Resource Director for the XA implementation. Registers the local integrated Resource for the DirectConnect Session Manager implementation.
- Pools connections so they can be re-used.
- Manages connection threading requirements.

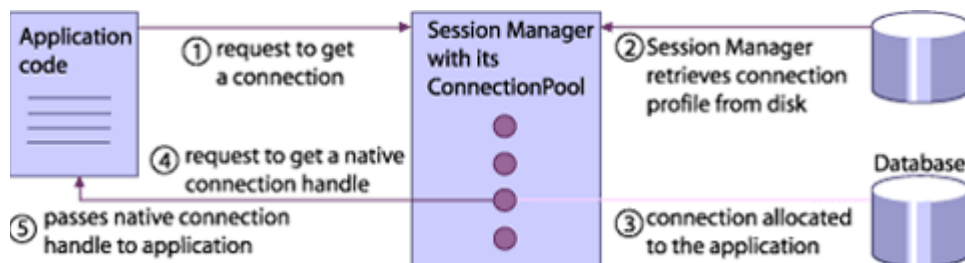
The Session Manager is linked with applications that use it. Using the Session Manager may require certain command-line parameters to be used when starting these application processes. For information about the command-line parameters, see [“Commands, utilities, arguments, and environment variables.”](#)

Opening a connection to a database

The Session Manager allows an application to obtain a VisiTransact-enabled connection to a specific type of database. This connection is created using a connection profile you have previously configured. See the information about configuration in the section that follows. Once the connection is made, the VisiTransact Transaction Service, in conjunction with the Session Manager, handles the transaction coordination.

How do the application and Session Manager work together? The application uses `resolve_initial_references()` to obtain a reference to the Session Manager's `ConnectionPool` object. The application provides the `ConnectionPool` with the appropriate configuration profile name and the `ConnectionPool` obtains a connection to the database using the configuration in that profile. The `ConnectionPool` then returns a `Connection` object which represents this database connection to the application.

Figure 13.1 The Session Manager gets a connection handle and passes the handle to the application code. The application code talks to the database directly through this connection handle.



In order to do database work, the application then requests the native database handle from the `Connection` object. Subsequently, the application code talks to the database

directly through this native connection handle. For example, if you are using Oracle, the application code makes direct OCI calls.

For more specific information about using the Session Manager API to get a connection to the database, see [“Data access using the Session Manager.”](#)

Programming restrictions vary according to the specific Session Manager implementation your application is using. For information on programming requirements and limitations while using the database's native API to do the work of the transaction (SQL statements, etc.), see [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)

Connection profiles

All the information needed for the connection is kept in connection profiles. Each profile has a unique name and consists of attributes such as the database login ID. The set of attributes varies depending on the Session Manager implementation. For more information, see [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)

Configuring connections

Use the VisiBroker Console to create and configure connection profiles. The connection profile has all the required attributes to make a connection to the database. For more information about the VisiBroker Console, see [“Using the VisiBroker Console.”](#)

Associating a connection with a transaction

The Session Manager associates transactions with the database work performed on a database connection—your application does not have to provide this function. This association is maintained until the application releases the Session Manager connection.

Registering Resources

The Session Manager automatically registers the appropriate Resources with the Coordinator—application developers do not need to add anything to their code for Resource registration. While the DirectConnect implementation contains the Resource object invisibly embedded in the Session Manager implementation, the XA implementation uses an external process called the XA Resource Director. The XA Resource Director must be available to use the XA version of the Session Manager. For information about starting an XA Resource Director, see [“Integrating VisiTransact with databases using the Session Manager.”](#)

Releasing Connections

The Session Manager requires that a Connection object be released when the application completes a unit of work against that connection. For implicit transaction contexts, the connection must be released before the transaction is disassociated from that thread. This disassociation occurs:

- In calls to transactional objects, when the call returns to the client.
- When the transaction is completed (`commit()` or `rollback()`).
- When the transaction is suspended.

When the application releases the connection, the Session Manager frees the database connection for use by other transactions.

Note

After releasing the connection, the application may not continue to use that particular Connection object or its associated native connection handle. To perform further work on that transaction or other transactions, the application must obtain a new Connection object.

Pooling connections

The Session Manager pools connections automatically. You do not have to add anything to your application code. When an application releases a connection, the Session Manager does not automatically close it. Rather, it keeps it in the connection pool. When there is another request for a connection, the pool will attempt to reuse connections. It will only open a new connection when there are no available compatible connections.

Within the same transaction, you can obtain and release Connection objects as many times as you need to complete work. Since the Session Manager ConnectionPool is more efficient when Connection objects are released after completing a unit of work, do not wait until the entire transaction is complete before releasing Connection objects.

Note

Because the DirectConnect implementation of the Session Manager uses a single connection to perform work for a transaction, the ConnectionPool can not reuse any connection until the transaction is complete. The connection is returned to the pool for reuse after the transaction is committed or rolled back.

Managing thread requirements

The Session Manager manages any connection threading requirements imposed by the database. Since the details about keeping connections with particular threads are incompletely specified by XA, DBMS companies have interpreted the XA requirements for threading behavior differently. For example, when using Oracle, a connection opened using XA requires that every single call for the rest of that connection's life has to be on that same thread. This makes it difficult to integrate with other software which manages threads according to its own policies.

The Session Manager makes sure that your application will always get a connection handle that works with the current thread.

Not all database connections have threading restrictions. When restrictions do not exist, the Session Manager pools connections more efficiently. For more information, see ["Data access using the Session Manager."](#)

Global transactions using XA protocol

Note

The Session Manager and XA Resource Director are not restricted to DBMSs or RDBMSs. They work with any Resource Managers that support XA protocol. Resource Managers are commonly thought to be databases but they include any XA-compliant Resource that is able to participate in a two-phase commit. Another example of a Resource Manager is a message queue.

XA is an accepted industry standard protocol specified by X/Open to allow Transaction Managers to coordinate global (two-phase commit) transactions. Most RDBMS vendors support XA as a way for external transaction coordinators (like the VisiTransact Transaction Service or TP Monitors) to control transaction completion.

Both the Session Manager and the XA Resource Director “speak” XA. Generally, they speak different pieces of XA. The part of XA that has to do with associating work with a transaction is facilitated by the Session Manager. The part of XA that has to do with transaction completion and recovery is performed by the XA Resource Director.

The Session Manager, in conjunction with the VisiTransact Transaction Service, performs the XA interface calls to include, in the transaction, the application's work on that database.

The XA Resource Director performs the two-phase commit for the database as instructed by the transaction's Terminator, and participates in recovery by acting as a bridge between the VisiTransact Transaction Service and the XA-compliant database. The XA Resource Director is deployed as a standalone process.

There should be one XA Resource Director instantiation (or process) deployed for each database.

What is the XA Resource Director?

Note

The XA Resource Director is used with the XA implementation of the Session Manager. You do not use the XA Resource Director with DirectConnect Session Manager deployments.

For transaction completion and recovery, the XA Resource Director bridges the VisiTransact and X/Open transaction environments, which allows for interoperability between Resource objects and the XA-compliant database. (The Session Manager does the bridging for associating application work with the transaction.) The Resource Director is a persistent object which acts as an intermediary during commit, rollback, or recovery for all transactions using a specific database on a network.

One XA Resource Director is associated with each database server. The Session Manager tells the VisiTransact Transaction Service which Resource Director will coordinate the completion of the transaction. After all the work has been done, the VisiTransact Transaction Service communicates with the Resource Director and tells it to commit or rollback the transaction.

Note

You do not have to implement an XA Resource Director or register it with the VisiTransact Transaction Service. It is done for you automatically; however, the system administrator must ensure that the XA Resource Director is available whenever the database is running.

Distributed transaction recovery

The XA Resource Director ensures that all transactions in its associated database, which were initiated by the VisiTransact Transaction Service, will be completed either by commit or rollback. The transactions will be completed regardless of failure caused by the VisiTransact Transaction Service, the XA Resource Director, or the Resource Manager. Any transaction not completed at the time of failure will be resolved when these three components are back up and running.

For more information about the rules used during transaction completion or a two-phase commit, see [“Transaction completion”](#) and [“Coordinating transaction completion with Resource objects.”](#)

DirectConnect Session Managers

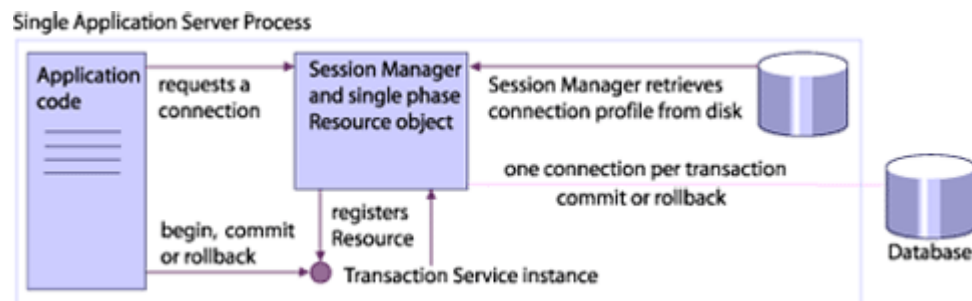
When only one application server talking to one database is involved in a transaction, the DirectConnect Session Manager is an alternative to using global (two-phase commit) transactions as provided by the XA implementation of the Session Manager. This consists of a single process containing a Session Manager with embedded Resources. For optimum performance, an VisiTransact Transaction Service instance is linked with the application code; however, this is not required. A transaction using the DirectConnect Session Manager is considered a *local* transaction because all of the components of the transaction are located locally in one process. With DirectConnect access transactions one process talks to one database. All of the work for a particular transaction is done on one physical database connection. The connection uses the same connection profile whenever the database is accessed for the same transaction. The benefit of this type of architecture is a gain in performance because it performs a single-phase commit only. It also allows transactional semantics for work done on databases which do not support two-phase commit.

Note

When using the DirectConnect Session Manager implementation, you do not need the services of the XA Resource Director. The Session Manager uses an internal, transparent Resource implementation.

The single application server process is a multi-threaded process that includes all the methods that might be used in a particular transaction. This process talks to a single database. Additional Resources are not allowed—databases or other types of Resources like message queues. For example, if you are doing a debit and credit transaction, instead of having your debit/credit application server processes on different machines, they are in a single process on one machine. The debit/credit process can talk to the single database across a network. But, a transaction talks to one database only. All the interactions with that one database happen with that one process.

Figure 13.2 A DirectConnect access transaction with all components of the transaction in one process



The DirectConnect Session Manager can take advantage of several performance optimizations. When you use a single application server process with a database as the only participant in the transaction, the transaction (via VisiTransact) performs a single phase commit. If the VisiTransact Transaction Service is embedded, the Session Manager does all the Resource registrations and all the work of the transaction locally to that process. The Session Manager, the VisiTransact Transaction Service instance, and the debit/credit process are all within the same application server process so they are not talking to each other across a network or across process boundaries. Additionally, because of the single phase commit, the VisiTransact Transaction Service need not log to the disk. Consequently, there is a performance gain.

The other advantage when using DirectConnect access transactions is that it is simpler to get them up and running. You do not have to install the XA client library or the components that enable global transactions. For example, if using XA access transactions with Oracle, you have to install Oracle's distributed option; with DirectConnect access transactions, this is unnecessary.

Note

The XA Resource Director is not used with the DirectConnect Session Manager. The Resource object used is built in to the DirectConnect implementation of the Session Manager.

When the application requests database access through the Session Manager, the ConnectionPool object allocates a database connection for that transaction. Unlike XA connections, the DirectConnect connection remains allocated to that transaction until the whole transaction is done. The application follows the same procedure (as it would with a distributed transaction) to get a connection and release a connection. Consequently, the VisiTransact Transaction Service knows what transaction and database is associated with a particular connection. When the request for a commit comes through, the Resource object that is embedded in the Session Manager gets that same physical connection and does a single phase commit or rollback on the transaction.

In other words, because the Session Manager manages connections, every time the application invokes `getConnection()` for the same transaction, it will get the same connection. The server can have many calls for the same transaction and all the work happens within the same transaction even though the application code did not have to maintain connection state.

Registering Resources

If you try to register a Resource (`register_resource()` call) after a DirectConnect Resource has already been registered, this results in a `CORBA::BAD_PARAM` exception. In other words, the VisiTransact Transaction Service will not accept any `register_resource()` calls after a DirectConnect connection has been used.

If `getConnection()` attempts to obtain a DirectConnect connection, and a Resource has already been registered with the Coordinator, the request will fail throwing a `VISSessionManager::Error` exception.

Deployment issues

You can choose to configure a stand-alone or embedded VisiTransact Transaction Service instance. If you embed an VisiTransact Transaction Service in a single application server, you may realize performance gains when processing a DirectConnect access transaction.

Restrictions on DirectConnect access transactions

In order to obtain the performance gains of DirectConnect access transactions, there are several restrictions that apply:

- Only one application server can be involved in a transaction.
- Only one Resource (Session Manager or other) may be involved in a transaction.
- Only one thread at a time can obtain a connection for a particular transaction. Once `getConnection()` has been invoked, no other thread will be able to obtain a connection for that transaction until the connection has been released.
- Since it is a single connection, anything the application does that alters a property in that connection (which some database calls do) sustains through the life of the connection. If the application uses a different thread later or performs a different unit of work, the properties on the connection will remain as set earlier. Because connections are reused, this could also affect work on subsequent transactions.

Coexistence: DirectConnect and XA access transactions

The design of the Session Manager allows for the coexistence of DirectConnect and XA access transactions in the same process. Certain database implementations will not allow the mixing of DirectConnect and XA access transactions from the same process. For example, the Oracle9i DirectConnect and XA implementations are not compatible in the same process, so VisiTransact will prevent you from mixing these two implementations. See [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client”](#) for details on specific implementations of the Session Manager.

14

Integrating VisiTransact with databases using the Session Manager

This chapter provides information you need to administer VisiTransact transactional applications that integrate with databases.

To integrate VisiTransact with databases, the database administrator is responsible for these tasks:

- 1 Evaluating the impact of integrating VisiTransact with databases.
- 2 Making sure databases are ready for integration with VisiTransact.
- 3 Setting up Session Manager Configuration Servers.
- 4 Configuring the connection profiles.
- 5 Deploying and setting up the XA Resource Directors. This involves starting the XA Resource Directors and registering them with the VisiBroker Object Activation Daemon (OAD) if appropriate. This step is only necessary when using the XA implementation of the Session Manager.
- 6 Starting the application objects that embed the Session Manager.

These tasks are described in detail in the sections that follow. Some of the information is presented in separate sections for XA and DirectConnect. The database administrator has other tasks in addition to the ones listed above. Additional tasks are as follows:

- Handling heuristics.
- Tuning for gains in performance.
- Managing connection profile persistent store files.

These additional tasks are described in detail later on in the chapter.

Evaluating the impact of integrating VisiTransact with databases using XA

One of an administrator's most important tasks is to evaluate the impact of processing distributed transactions in a particular site's environment. Certain circumstances are inherent when processing distributed transactions. Processing a distributed transaction may not always be appropriate for the database your company is using. While making your evaluation, consider the following:

- Using the XA protocol adds overhead.
- The database must have a high degree of availability during two-phase commits.
- Data may be locked or unavailable for longer periods, reducing potential concurrency.
- The database is involved in a more sophisticated transaction. It may have to work with other application components.

These items are discussed in the following sections.

Using XA adds overhead

Generally, there is extra overhead when communicating with the database using the XA protocol and XA interface calls. The overhead incurred for XA is as follows:

- A round trip to the database to perform the association with the transaction.
- A round trip to the VisiTransact Transaction Service to register the database's participation in the transaction.
- One or two round trips from the VisiTransact Transaction Service to the XA Resource Director to perform the prepare and commit processing.

With VisiTransact, the calls to associate and disassociate only happen to those database connections that the application has specifically requested to use at that time. Overhead does not incur for Resource Managers that have not been used.

Requiring high availability

If the VisiTransact Transaction Service invokes two-phase commit on a set of databases, and if any are unavailable during the prepare phase, the transaction is rolled back. Any productive work done during those transactions is lost.

Locked or unavailable data

Performing a two-phase commit may cause concurrency bottlenecks. Between data being locked and committed, the database prevents anyone from reading or modifying data which is locked by that particular transaction. For example, if you lock data in a row because you updated it, it will not be available for someone else to modify until you commit the transaction. This reduces concurrency.

Note

The behavior of databases locking data varies widely—a row or more of data could be locked depending on the database and the application.

Yielding some control

When evaluating the advantages and disadvantages of processing distributed transactions, consider that the scope of administrative tasks has increased. This causes a different set of advantages and disadvantages. You lose some control because you cannot force completion after the prepare phase has started. This is

because the scope is wider and there are other components to consider. If the two-phase commit is interrupted, a heuristic outcome occurs. You can force a heuristic outcome using database utilities.

For more information about how the VisiTransact Transaction Service handles heuristics, see [“Transaction completion”](#).

Evaluating the impact of integrating VisiTransact with databases using DirectConnect

There are fewer administration duties for DirectConnect transactions than for XA transactions.

Restrictions when using DirectConnect are as follows:

- Only one Resource (the DirectConnect Resource) can participate in the transaction.
- The transaction work with one database is restricted to one process.

Advantages to using DirectConnect are:

- Simpler deployment scenarios
- Reduced RPCs to the database for XA coordination

Preparing databases

Before you can use the features in the Session Manager, check with the database administrator that the database has the required software subsets for distributed transaction access. Your database administrator may need to modify your database installation by loading additional libraries, running SQL scripts in the database, modifying configuration parameters for the database server, and installing client-side libraries. For more information, see [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)

In general, for DirectConnect, there are no additional steps in preparing your database because the connections are regular user connections.

Connection profile sets

For the Session Manager to connect to a database, it must be supplied with information about how to make that connection. The information is packaged into a set of attributes called the connection profile. Connection profiles are created using the VisiBroker Console, or using the `smconfigsetup` utility, and saved onto disk so that they can be persistently stored and retrieved by the Session Managers running in the application servers. Because the connection profiles are available on disk, the Configuration Servers do not have to be running continuously. A profile set (logical grouping of connection profiles) is associated with the same Configuration Server. Each Configuration Server is identified by a unique name. This section provides information to help you manage connection profile persistent store files.

Note

You may want to decouple changes made to profile attributes by an application's Session Manager from the profile attributes used by the XA Resource Director. To do so, create separate connection profiles for the Session Manager and the XA Resource Director.

The profile sets are stored in persistent storage files. You can locate the persistent storage files in the default location, or you can use different locations and then point to these locations using the argument `-Dvbroker.sm.pstorePath`.

Modifying connection profiles used by Session Manager clients

To modify a connection profile used by Session Manager clients:

- 1 Change the connection profile using the VisiBroker Console.
- 2 Shut down any application processes which use this connection profile.
- 3 Restart the application processes.

Modifying connection profiles used by XA Resource Directors

To modify a connection profile used by the XA Resource Director:

- 1 Change the connection profile using the VisiBroker Console.
- 2 Shut down any application processes which use the affected XA Resource Director.
- 3 Shut down and restart the XA Resource Director.

Note

While it is possible to leave the application processes running, transactions which attempt completion while the XA Resource Director is shut down may be rolled back.

Using the XA Resource Director

The XA Resource Director is used in conjunction with the XA implementation of the Session Manager for transaction completion and recovery processing. The XA Resource Director is deployed as a standalone program.

If you are using the DirectConnect implementation of the Session Manager, an XA Resource Director is unnecessary.

Deploying an XA Resource Director

Deploy an instance of the XA Resource Director for each database server accessible from VisiBroker VisiTransact. The XA Resource Director should be running whenever the database server is running if the XA implementations are being used. This way the Resource Director is available to take part in the completion and recovery protocols.

We recommend that only one XA Resource Director be deployed for each database for the same `OSAGENT_PORT`. Having multiple XA Resource Directors on the same `OSAGENT_PORT` for the same database is inefficient because, although they are successful at committing and rolling back transactions during normal operation, they also duplicate recovery operations if the VisiTransact Transaction Service goes down and comes back up. This results in overloading the VisiTransact Transaction Service with replay requests when it has finished its internal recovery cycle.

Starting an XA Resource Director

Start the XA Resource Director with the following command:

```
prompt>xa_resdir -Dvbroker.sm.profileName=<profile>  
[-Dvbroker.sm.pstorePath=<path>] [-Dvbroker.sm.configName=<name>]
```

The following table describes the start-up parameters for the XA Resource Director.

Parameter	Description
<code>-Dvbroker.sm.profileName=<profile></code>	The name of the Session Manager connection profile you want to use to establish a connection with the database. This is required.
<code>-Dvbroker.sm.pstorePath=<path></code>	The path to the directory where the persistent store files are located. By default, the persistent store files are located in <code><VBROKER_ADM>/its/session_manager/</code> .
<code>-Dvbroker.sm.configName=<name></code>	The name of the Session Manager Configuration Server you're using. By default, the name assigned to the Session Manager Configuration Server is <code><host>_smcs</code> where host is the name of the server on which you created the Session Manager connection profile. This can also be thought of as the profile set name.

For information about what defaults are used if you do not specify the `-Dvbroker.sm.pstorePath` parameter, see [“Checking for the default path to persistent store files”](#). For information about how to set up a Session Manager connection profile, see [“Using the Session Manager Profile Sets section”](#).

How the XA Resource Director uses connection profiles

Besides creating connection profiles for the Session Manager, you must create them for the XA Resource Directors as well. Depending on what attributes they need for configuring connections, the Resource Director may use the same profile as some of the Session Managers use; it might use a profile that none of the Session Managers use. While there might be multiple profiles that Session Managers use to contact one database, the XA Resource Director for that database uses only one profile.

Deploying client-side libraries

The Session Manager and the XA Resource Director must be able to access the client-side database libraries, including the XA client-side library for the databases that the Session Manager and XA Resource Director objects will be accessing.

Shutting down an XA Resource Director remotely

To shut down an XA Resource Director remotely, use the following command:

```
prompt> vshutdown -type rd [-name <ITS_XA_Resource_Director_name>]
```

The XA Resource Director's type (rd) is a required argument.

You can find the name of the Resource Director by either using the `osfind` command, or by looking at the connection profile. To avoid confusion, it is best to name the Resource Director the same name as its connection profile and database name. See [“vshutdown”](#) for more details on using the `vshutdown` command.

Registering the XA Resource Director with the OAD

To start the XA Resource Director without operator intervention, register it with the VisiBroker OAD (Object Activation Daemon). Implementations of the XA Resource Director can be registered using the `oadutil reg` command-line interface.

The syntax for registering an XA Resource Director with the OAD is as follows:

```
oadutil reg -i visigenic.com/VISSessionManagerSupport/ImplicitResource
-o <resource_director_name> -cpp <installation_dir_path>/bin/xa_resdir -a
-Dvbroker.sm.profileName=<profile> -a -Dvbroker.sm.pstorePath=<path> -a -
Dvbroker.sm.configName=lt;name>
```

The following table describes the parameters for registering the XA Resource Director with the OAD.

Parameter	Description
resource_director_name	This is the name of the XA Resource Director you wish to register with the OAD, and the object name that will be activated. We recommend that the profile used to start the Resource Directors be named the same as the Resource Director name in the profile, and reflect the database name. In installations with multiple databases, this makes it easy to associate Resource Directors with profile names.

Note

For description of the `profile`, `path`, and `name` parameters, see [“Starting an XA Resource Director”](#).

The connection profile used to start the XA Resource Director should be named the same as the XA Resource Director name in the profile. Additionally, it should be the same as the database name. In installations with multiple databases, this makes it easy to associate XA Resource Directors with profile names.

See [“Using the Session Manager Profile Sets section”](#) for details on how to set up a Session Manager connection profile.

Note

The OAD must be running before you can use any of the OAD commands. Refer to Starting the OAD in the *VisiBroker for C++ Developer's Guide* for instructions on starting the OAD. When registering an object implementation, use the same object name that is used when the implementation object is constructed.

Starting Session Manager-based application processes

It is not necessary for an administrator to explicitly start up the Session Manager. The Session Manager is started and initialized automatically in the programs in which it is used. The ORB initialization accesses the command-line arguments that contain the connection profile attributes and any other options relating to the Session Manager.

If you want to use a path other than the default, or a profile set name other than the default, use the following arguments when starting the application so that the persistent store of connection profile attributes is used:

```
-Dvbroker.sm.pstorePath=<path> -Dvbroker.sm.configName=<name>
```

The path argument is not required. If you do not specify a path argument, see [“Checking for the default path to persistent store files”](#) for information about how the Session Manager and the Session Manager Configuration Server check for the default path and profile set name.

Note

See [“Starting an XA Resource Director”](#) for a description of the command-line arguments to application for the Session Manager.

Checking for the default path to persistent store files

When using the Session Manager, the `-Dvbroker.sm.pstorePath` argument is not required. If you do not specify the path argument, the Session Manager and the Session Manager Configuration Server check the following settings in this order:

- 1 What you set in the command-line argument for `-Dvbroker.sm.pstorePath`. If you did not specify the path at the command line, it checks:
- 2 What was set with the `VBROKER_ADM` environment variable. This is the default when you accept all the defaults during installation. VisiTransact puts the persistent store files in subdirectory `its/session_manager` under `VBROKER_ADM`.

Forcing heuristics

You may use database utilities to monitor transactions after they reach the prepare phase. In some cases, you may need to interfere to resolve transactions; for example, in the case of long-lived failures with the VisiTransact Transaction Service or one of its participants, or a failure in network connectivity. When a database administrator intervenes to commit or roll back a prepared transaction without using the VisiTransact Transaction Service, the resulting state is called a heuristic. This means that the database may have completed the transaction in a way different than the VisiTransact Transaction Service has. Most databases which support two-phase commits have interfaces for forcing heuristics.

For more information about how the VisiTransact Transaction Service handles heuristics, see [“Transaction completion.”](#)

Performance tuning

When you have the VisiTransact Transaction Service embedded in the application server, the client should ensure that it binds to the correct instance of the VisiTransact Transaction Service to realize the potential performance gains.

See [“Embedding a VisiTransact Transaction Service instance in your application”](#) for more information on embedding an VisiTransact Transaction Service in your application server.

For XA

Reducing network traffic increases performance when using the VisiTransact Transaction Service with distributed transactions. To reduce network traffic, you can locate some of the components on the same node as the VisiTransact Transaction Service instance, or on the same node as the database. Communication occurs between the Session Manager and the VisiTransact Transaction Service, the Session Manager and the database, the VisiTransact Transaction Service and the XA Resource Director, and the XA Resource Director and the database. Localizing these components on the same node will reduce network traffic. Consider trying to locate the transactional objects which use the Session Manager on the same node with the VisiTransact Transaction Service or the database, and locate the XA Resource Director with the database or with the VisiTransact Transaction Service.

Session Manager Configuration Server

The Session Manager Configuration Server represents one set of connection profiles and servers as the agent for VisiBroker Console. Its purpose is to provide network access for the VisiBroker Console to the connection profiles.

Directory structure for persistent store files

By default, the persistent store files are located on disk in the connection profile set subdirectories. You can use the default directory or specify another one. The default path may be overridden during installation, or when using the command line flag - `Dvbroker.sm.pstorePath`, or with any process that uses the Session Manager (including the XA Resource Director).

For information about how the Session Manager and the Session Manager Configuration Server checks for the default path to the persistent store files, see [“Checking for the default path to persistent store files”](#).

Caution

There is a directory called **install** under the **session_manager** directory. Do not change anything in the **install** directory or add files to it manually. This directory is created automatically when you install VisiTransact.

When you create a connection profile with the VisiBroker Console, a corresponding file is created in a subdirectory in the **session_manager/config** directory. The name of subdirectory file corresponds with the Session Manager Configuration Server's name and can be considered the profile set name. By default, the Configuration Server's name consists of **<host>_smcs** where **host** is the name of the machine where the Configuration Server resides. For example, if the machine's name is **athena**, the Configuration Server is named **athena_smcs**. In the subdirectory for the Configuration Server, the connection profiles are stored, one per file. You can give the connection profiles significant names like **test_oracle_xa**. The name you give a connection profile using the VisiBroker Console is automatically assigned to its associated persistent store file. You do not have to manually create the persistent store files—they are created by the Session Manager Configuration Server when you use the VisiBroker Console. VisiTransact adds the extension **.cfg** to the persistent store file, as shown in the following example.

```
test_oracle_xa.cfg
```

Note

When you create names for the connection profiles using the VisiBroker Console, the case sensitivity rules for these names are the same as the rules used by your file system where these names are stored. For example, on UNIX if you mix the upper and lower case when you assign the connection profile name, that is what you must use when you try to find it later.

While the persistent store files are binary and cannot be edited by hand, it is possible to copy them to alternate locations as backups. Or, you can copy an entire **<configuration_server>** subdirectory to another location and rename it with a different profile set name.

It is possible to partition connection profile sets so that you have multiple connection profile sets on the same node. Unless there is a strong reason for doing this, it usually has few advantages. If you have more than one profile set on a node, a subdirectory is created in the **session_manager/config** (even on different nodes) directory for each profile set. Do not create multiple profile sets with the same name since you will not be able to distinguish them from the ones you create with the VisiBroker Console.

Note

A process using the Session Manager can only access one connection profile set by going to the default location, or to any location you have specified using command-line arguments. It is confined to that namespace for the default location, or location specified by the command-line arguments. It cannot access locations that are not specified. For example, an instance of a Session Manager in a particular application process may only access the Marketing connection profile set. It may not access the Payroll connection profile set.

Deploying persistent store files

Every node which is running an application that uses the Session Manager must be able to read the persistent store files from disk. Consequently, you have several options when deploying the persistent store files:

- **Option 1:** One Configuration Server exists on one of the nodes in a connected group of nodes. The on-disk set of persistent store files is shared through a shared file system.
- **Option 2:** A uniquely-named Configuration Server and its on-disk set of persistent store files exist on each node.
- **Option 3:** One Configuration Server is shared by a connected group of nodes. There is no shared file system. When the on-disk set of persistent store files changes, you must manually copy the sets of persistent store files from the master location to the other nodes.

Option 1: Persistent store files on a shared file system

The preferred method of deployment is to place the set of persistent store files on a shared file system. When VisiTransact is installed, a Session Manager Configuration Server with an associated on-disk set of persistent store files is deployed on only one node. When installing VisiTransact on the node which runs the Session Manager Configuration Server, you can specify the directory structure for the persistent store files so that it will be created in this shared disk.

After installation, the VisiBroker Console can be used to update all Session Manager connection profiles that are used on this network. They will all appear in one set of connection profiles. Application servers which use the Session Manager on other nodes must be configured so that when they start up they look at the shared disk containing the persistent store (using `-Dvbroker.sm.pstorePath=<path>`). When you use the VisiBroker Console to update connection profiles, the modified profile will be seen by application servers when they start up.

Note

Because the connection profiles are shared with this option, it is very important that there is only one instance of the Session Manager Configuration Server used in this group of nodes.

Option 2: Persistent store files on each node

Each node that runs the Session Manager has its own Configuration Server and set of persistent store files. One VisiBroker Console on the network can make modifications to each on-disk persistent store file. If you update a connection profile on one node, you have to update it on the other nodes through the VisiBroker Console so that the other nodes are updated with the change too.

This option has the advantage that no disk sharing is needed, but adds much complexity in synchronizing the connection profiles across many nodes. Neither the VisiBroker Console nor the Session Manager Configuration Server processes can synchronize different on-disk sets of connection profiles.

Option 3: Set of persistent store files copied to each node

To distribute the on-disk cache around the network, you can create a master set of persistent store files and manually copy them to each node. Like the previous option, this option has the disadvantage of trying to keep numerous nodes synchronized. However, it may be easier to copy the profile files using operating system or network copy commands than to update each persistent store file using the VisiBroker Console.

Note

You can copy all persistent store files in a Configuration Server subdirectory or just one persistent store file at a time. You must copy the entire install subdirectory too if it does not already exist in the target location.

Starting the Session Manager Configuration Server manually

Once a Session Manager Configuration Server is registered with the OAD, the OAD can start it automatically; however, if you would like to start the Configuration Server manually, use the following command:

```
prompt>smconfig_server [-Dvbroker.sm.pstorePath=<path>] [-
Dvbroker.sm.configName=<name>]
```

The following table describes the start-up parameters for the Session Manager Configuration Server.

Parameter	Description
-Dvbroker.sm.pstorePath	Provide the path to the directory where the persistent store files are located. By default, the persistent store files are located in <VBROKER_ADM>/its/session_manager.
-Dvbroker.sm.configName	Provide the name of the Session Manager Configuration Service you're using. By default, the name assigned to the Session Manager Configuration Server is <host>_smcs where host is the name of the server on which you created the Session Manager profile.

Shutting down the Configuration Server

You may want to shut down the Session Manager Configuration Server for the following reasons:

- If you need to perform maintenance.
- If an error occurs.
- If you need to reboot the machine the server is on.

Use this command when you want to shut down a Session Manager Configuration Server manually:

```
prompt>vshutdown -type smcs [-name <smcs_name>]
```

The Session Manager Configuration Server name should be the one used in the command-line argument -Dvbroker.ots.configName when you started the Configuration Server, or the default name which is <host>_smcs.

Note

If someone else is using the VisiBroker Console to change or create a connection profile that is associated with the same Session Manager Configuration Server that you are trying to shut down, the Session Manager Configuration Server will finish the work and then will shut down.

See “[vshutdown](#)” for more information on the **vshutdown** command.

Security

Database passwords are secure in the sense that they are not displayed through the VisiBroker Console. Applications which link with the Session Manager are able to query cleartext versions of database passwords. Since read access to the configuration files is required of these applications, you can control access to database passwords by restricting access to the connection profile persistent store files. It is up to the application developer and system administrator to provide proper file access permissions and develop applications which do not disclose password information to unauthorized users.

15

Data access using the Session Manager

This chapter explains how to use the Session Manager to manage connections between transactional objects and databases in a distributed environment—this includes DirectConnect and XA access. It assumes that you are familiar with the CORBA Transaction Service specification, and database concepts.

Note

Session Manager is supported on the Solaris platform only.

The Session Manager includes the following interfaces:

- `Connection`—represents a transactional database connection.
- `ConnectionPool`—allocates a connection from the pool.

For an overview about the Session Manager and the XA Resource Director, see [“Session Manager overview.”](#)

Preparing for integration

Before you can use the features in the Session Manager, you must do the following:

- Install your database. It may require special configurations depending on whether you are processing XA access transactions or depending on the other components in your environment. For more information, see [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)
- Ensure that your VisiTransact system administrator has created connection profile(s) for the Session Manager. If you are processing XA access transactions, your VisiTransact system administrator must create a connection profile for the XA Resource Director as well.
- Verify that your application uses the Session Manager (`ConnectionPool` and `Connection` interfaces) to obtain connection handles. To obtain connections, use connection profiles by name—the name that was given to the connection profile via the VisiBroker Console.

- For XA implementations of the Session Manager: Check with your VisiTransact system administrator that an instance of the XA Resource Director is deployed and running for each database that is accessible from VisiBroker VisiTransact. See [“Integrating VisiTransact with databases using the Session Manager”](#) for more information.
- Check that your application ensures a transaction is in progress. You must have an active transaction (implicit or explicit context) on the current thread. This ensures that Resources are included in a VisiBroker VisiTransact transaction. See [“Creating and propagating VisiTransact-managed transactions”](#) for a description of VisiTransact-managed transactions or see [“Other methods of creating and propagating transactions”](#) for a description of how to manage transactions in other ways.

Using the Session Manager: Summary of steps

The following steps summarize how to work with the Session Manager.

- 1 Obtain a `ConnectionPool` object reference.
- 2 Ensure that there is an active transaction.
- 3 Obtain a `Connection` object for the appropriate connection profile from the `ConnectionPool`.
- 4 Get a native connection handle from the `Connection` object using `getNativeConnectionHandle()`.
- 5 Use the native connection handle to access data.
- 6 Release the Session Manager `Connection` object, and cleanup any copies of the native connection handle.
- 7 De-allocate the `Connection` object.

Note

You can execute lots of pieces of work for a single transaction. Because connections are pooled, you should keep a `Connection` object for a short while and not hold onto it. You can get `Connection` object as often as needed within a single transaction.

The following sections detail each step.

Obtaining a `ConnectionPool` object reference

The following steps describe the general process for obtaining a reference to the `ConnectionPool` object, and are followed by a code example.

- 1 Call the ORB `resolve_initial_references()` method, passing the object type `VISessionManager::ConnectionPool`.
- 2 Narrow the returned object to a `VISessionManager::ConnectionPool`. The following is an example of obtaining a `ConnectionPool` object reference in C++.

```
{
CORBA::ORB_var orb = CORBA::ORB_init();
CORBA::Object_var initRef =
    orb->resolve_initial_references("VISessionManager::ConnectionPool");
VISessionManager::ConnectionPool_var pool =
    VISessionManager::ConnectionPool::_narrow(initRef);
    ...
}
```


Using ConnectionPool object references

The ConnectionPool object reference is valid for the entire process under which you create it; you can use it in any thread. You can either make multiple calls to obtain references to the ConnectionPool object or use just one reference throughout the entire process, saving duplicate `resolve_initial_references()` calls.

Obtaining a Connection object from the Connection Pool

Once the application has obtained a reference to the ConnectionPool object, the `getConnection()` call can be used to obtain a Connection object which represents this database connection to the application. It is at this point that the Session Manager binds a database connection with a Connection object.

The `getConnection()` call requires an active implicit transaction context. The `getConnectionWithCoordinator()` call can be used to explicitly specify a transaction using its Coordinator. For more information about `getConnectionWithCoordinator()`, see [“Using explicit transaction contexts”](#).

The `getConnection()` method does the following:

- 1 Obtains a database connection.

If there is a free connection in the pool with the same connection profile, the pool returns that connection. If there is no free connection with a matching connection profile available, the Session Manager creates a new connection. The connection is created using an appropriate method for the specific Session Manager implementation.

Note

You cannot override connection attributes programmatically.

- 2 Associates the work performed on this connection with the transaction.
- 3 Registers the appropriate Resource object with the VisiTransact Transaction Service.

The following code sample shows how to get a Connection object to represent a connection.

```
...
VISSessionManager::ConnectionPool_var pool;
// Ask the pool for a database connection
VISSessionManager::Connection_var conn = pool->getConnection("quickstart");
...
```

Any errors that happen during any of the steps will be returned as exceptions to `getConnection()` or `getConnectionWithCoordinator()`. If any of these steps fail, the Session Manager will throw an exception rather than returning a Connection object.

Using explicit transaction contexts

You can get a connection for an explicit transaction context by using `getConnectionWithCoordinator()`. The `getConnectionWithCoordinator()` method is used for the following reasons:

- To get a Connection object when there is no active implicit transaction context.
- To get a Connection object and use it with a transaction other than the currently active implicit transaction context.

If you call `getConnectionWithCoordinator()` and pass in a Coordinator reference, the Session Manager will use the Coordinator to perform all the tasks it normally would do with the implicit context. Instead of using an implicit VisiTransact-managed transaction, the Session Manager uses the explicitly-stated transaction Coordinator. The

connection will be set with this transaction Coordinator until you release the connection.

The following code sample shows `getConnectionWithCoordinator()` passing transaction context via the Coordinator.

```
...
VISessionManager::ConnectionPool_var pool;
// Ask the pool for a database connection using the "quickstart" profile
VISessionManager::Connection_var conn =
    pool->getConnectionWithCoordinator("quickstart", coordinator);
...
```

For more information about explicit propagation of transaction context, see [“Other methods of creating and propagating transactions.”](#)

Optimizing connection pooling

The Session Manager automatically keeps a pool of connections and returns connections to applications based on a set of attributes. For efficient connection pooling, the application should use the same connection profile and attributes for all connections to a single data source.

Getting a native connection handle

To use a connection represented by a Connection object, the application must obtain a native connection handle from the Connection object using the `getNativeConnectionHandle()` method. Subsequently, the application code talks to the database directly through this native connection handle. This native connection handle can then be used to do normal data access. In other words, you can do the work in the database API with which you are familiar.

The following code sample shows how to get a connection handle to an Oracle database for use with the OCI interface.

```
...
VISessionManager::Connection_var conn;
// get an Oracle OCI connection handle
handles = (SampleOraHandle *) smconn->getNativeConnectionHandle();
...
```

Using the native connection handle

You can use the connection handle obtained through the Session Manager the way you would use any native connection handle: using the native database API. However, some actions are disallowed when using the Session Manager. For all implementations, all calls to the database which have an effect on the transactional state are prohibited, including any calls which begin, commit, or roll back a transaction. Such calls affect the transactional integrity of the work. Transactional effects may be hidden. For example, DDL statements (like create table) in Oracle force an implicit commit call. For more information about actions that are prohibited, see [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)

Threading requirements

The Session Manager automatically manages the database connection's thread requirements, if there are any. The `isSupported()` method may be used with the `thread_portable` argument to determine whether connections may be used on other threads. In general, the connection returned by the `getConnection()` method is only valid on the thread used to acquire it; the application may not use the connection handle on any other thread. There may be relaxed restrictions for some implementations. For more information about thread requirements for a specific

database, see “[XA Session Manager for Oracle OCI, version 9i Client](#)” and “[DirectConnect Session Manager for Oracle OCI, version 9i Client](#).”

Releasing a connection

Calling `release()` on a `Connection` object ends the association of the transaction with a particular connection. Releasing the `Connection` object does not close the underlying database connection; the connection is returned to the pool for re-use. After releasing a `Connection` object, the application may not use the native connection handle or `Connection` object again. If you decide to perform more work on this transaction, you can obtain another `Connection` object.

There are two methods available for releasing a connection: `release()` and `releaseAndDisconnect()`. For usage, see the following table.

Generally, the native connection handle is a pointer. Therefore, you should set to null any copy you have of that pointer when releasing the connection.

If you are implementing an IDL interface call, you must `release()` or `hold()` the connection before returning from that call. Failure to do so will circumvent the Session Manager's ability to avoid lost connections due to unreliable clients, and to enforce threading restrictions imposed by some database implementations.

The application must not attempt to disconnect using the database API. It should use `release()` or `releaseAndDisconnect()`.

Caution

If the application uses a connection handle after a release method is called, unpredictable results will occur.

The following table explains the behavior of the `release()` and `releaseAndDisconnect()` methods.

Method	Behavior
<code>release(in ReleaseType type)</code>	<p>When the application invokes <code>release()</code>, it must use one of the two enumerated values. If the application uses <code>release()</code> with <code>MarkSuccess</code>, the connection is disassociated from the transaction successfully and the connection returns to the pool.</p> <p>The application uses <code>release()</code> with <code>MarkForRollback</code> to mark the transaction for rollback only. The Session Manager signals to the database and the VisiTransact Transaction Service that the transaction will be rolled back. The application would take such an action when it detects a condition which would invalidate the integrity of the transaction.</p>
<code>releaseAndDisconnect()</code>	<p>This method may be used if the application detects something wrong with the connection to force the connection to close and to notify the VisiTransact Transaction Service that the transaction is marked for rollback only.</p>

The following code sample shows an example of code that releases a transaction successfully.

```
...
VISSessionManager::Connection_var conn;
conn->release(VISSessionManager::Connection::MarkSuccess);
...
```

De-allocating the instance of Connection

The `release()` call does not release the `Connection` object in the sense of the CORBA `_release` method. It indicates to the `ConnectionPool` that the connection will no longer be needed by the application. The application will still need to de-allocate the `Connection` object. Because these are CORBA objects, you cannot call `delete()` on them. To ensure the safest management of the `Connection` object, hold it in a

`Connection_var`. When the `Connection_var` is destructed, everything gets cleaned up. For the `ConnectionPool` object, hold it in a `ConnectionPool_var`.

Note

If `release()` is not called on a `Connection` object, the default destructor releases the connection and marks the transaction `RollbackOnly`. This is an easy way to make sure that all abnormal exits from this method keep the transaction from proceeding. If the application can maintain transactional integrity without rolling back, its own exception handling should release the connection explicitly, marking it for success where appropriate.

Viewing exceptions

Session Manager objects may throw exceptions. The exceptions are defined in IDL. Therefore, the exceptions are handled in the standard CORBA way—the ORB is responsible for transmitting the information back to the caller.

Exception	Behavior
<code>VISSessionManager::Error</code>	This exception is defined in the <code>VISSessionManager</code> module and contains a sequence of <code>ErrorInfo</code> structures. An <code>ErrorInfo</code> is a struct of reason, subsystem, and error code.
<code>VISSessionManager::ConnectionPool::ProfileError</code>	This exception is defined in the <code>ConnectionPool</code> interface and consists of a reason and a code.

If your application receives the `ProfileError` exception, there are two fields in the exception: a reason and an error code. You can look at these fields to see information about the error.

If your application receives the `Error` exception, you need to see how long the sequence of `ErrorInfos` is by invoking the `exception.info.length()` method on the `ErrorInfos` sequence. Once you know the length, you can step through each `ErrorInfo` in the sequence.

The following code sample shows an example of code that catches exceptions in connection calls.

```
...
try
{
    CORBA::ORB_var orb = CORBA::ORB_init();
    CORBA::Object_var object =
        orb->resolve_initial_references("VISSessionManager::ConnectionPool");
    pool = VISSessionManager::ConnectionPool::_narrow(object);

    conn = pool->getConnection("quickstart");

    lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
}
catch (VISSessionManager::ConnectionPool::ProfileError &ex)
{
    cerr << "Profile error: " << ex.code << ex.reason << endl;

    conn->releaseAndDisconnect();

    throw ApplicationException(); //This error is defined by the application
}
catch (VISSessionManager::Connection::Error &ex)
{
    cerr << "Session Manager error: " << endl;

    int len = ex.info.length();
    for (CORBA::ULong i=0; i<len; i++)
```

```

{
    cerr << ex.info[i].subsystem << "-" << ex.info[i].code
        << ": " << ex.info[i].reason << endl;
}
conn->releaseAndDisconnect();

throw ApplicationException(); //This error is defined by the application
}
...

```

Viewing attributes

There are two methods you can use to view connection profile attributes. They are used for different purposes: one is used when a connection is currently allocated, the other is used when there is no connection allocated.

Method	Behavior
VISSessionManager:: Connection::getAttributes()	This method returns the values of configuration profile attributes for a connection that is currently allocated. This method is in the <code>Connection</code> interface.
VISSessionManager:: ConnectionPool:: getProfileAttributes()	This method may be used to query attributes in a connection profile without allocating a connection. You may want to use this method to evaluate which connection profile you want to use. This method is in the <code>ConnectionPool</code> interface.

For more information about `getAttribute()` and `getProfileAttributes()`, and for a list of the connection attributes common to all supported databases, see `VISSessionManager` module in the *VisiBroker for C++ API Reference*.

Obtaining Session Manager information

To obtain information such as the version of the Session Manager, whether the `hold()` method is supported or not, or the database's threading policy, use the following methods:

```

string* getInfo(in string info_type)
boolean isSupported(in string support_type)

```

The following information types are common to all the Session Managers:

- "version"—Returns the version number of the generic Session Manager. The version number is returned in a 5-field string which is standard in the `VisiBroker` utility `vbver`. This `info_type` does not return specific information about which component you are talking to. This information is to be used for informational purposes.
- "version_rm"—Returns the version number of the Resource Manager-specific component of the Session Manager. This information is to be used for informational purposes.

The following support types are available for all types of Session Managers:

- "hold"—Returns true if the `hold()` method is supported; otherwise, returns false.
- "thread_portable"—Returns true if the connections are restricted to the thread that made the connection; otherwise, returns false.

The following code sample is an example for using `getInfo()`.

```

...
VISSessionManager::Connection_var conn;
CORBA::String_var info = conn->getInfo("version");
...

```

Using hold() and resume()

These methods are used to maintain ownership of a Session Manager Connection when the thread of control returns to a client.

Method	Behavior
<code>VISessionManager::Connection::hold()</code>	This method notifies the Session Manager that the thread of control is leaving the current process and intends to return.
<code>VISessionManager::Connection::resume()</code>	This method is used after a <code>hold()</code> to indicate to the Session Manager that the thread of control for this Connection is now back in process.

Using hold()

The Session Manager requires that it be notified if no thread in the current process is active with respect to this connection. The main reason for this restriction is that if the requester fails or is otherwise unable to return to this process to release its Resources, the Session Manager must be able to clean up any resources used for this connection. If the Session Manager does not have knowledge of whether or not the application is still actively using the connection, it cannot dissociate the transaction and proceed with cleanup.

There is another more subtle reason to use `hold()`. Some database connections are restricted in their use to a particular thread. Here is an example of what may happen:

- A client makes an interface call to the server.
- The client obtains a Connection object through the Session Manager.
- Later, the client makes another interface call to the same server hoping to perform more work on that transaction. This call may occur on a different thread, depending on the behavior of the BOA used in the server process.

Using `hold()` gives the Session Manager a chance to inform the application that making a second interface call to the same server may not be supported for some Session Manager implementations which do not allow the connection to be used on another thread.

Note

Using `hold()` monopolizes the connection and affects performance; use `hold()` only when it is necessary.

The `timeout` parameter specifies the time in seconds that the Session Manager should wait before timing out the connection and cleaning up its resources. As part of the cleanup process, the connection is returned to the ConnectionPool and the transaction is marked for rollback.

Your application can send multiple `hold()` requests with no intervening `resume()` calls. If `hold()` is called twice, the timer is reset with the new value at each call. For example, if you send `hold(60)` at 8:42:30, it would expire at 8:43:30. However, if you subsequently invoke `hold(45)` at 8:42:50, the timer would expire at 8:43:35 because it had been reset by the second `hold()` call.

Note

Some database Session Manager implementations may not support this method. Your application can use `isSupported()` to query whether the Session Manager supports the `hold()` method or not. Also you can find more information about this in [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client.”](#)

Before the Connection object or the corresponding database connection handle can be used again, `resume()` must be called on the Connection object.

Using resume()

The `resume()` cancels the timer associated with the `hold()` and guarantees that the Session Manager will not modify the underlying connection in any way that would cause conflicts with an active application. Calling `resume()` when the Connection has not been placed in the hold state results in a `VISSessionManager::Error` exception, but does not modify the transaction or connection state.

Note

Between the `hold()` and `resume()` calls, the application is not allowed to make any other calls on the Connection object or its associated native database handle. If the `hold()` call timer expires in this interval, the Session Manager has the right to release the connection and mark the transaction for rollback. This is to ensure that Resources held in the application server by that transaction are not left forever if a client dies or never calls again.

Example of a simple integration

The following code sample shows an example program that integrates a DBMS using the Session Manager.

```
...
void applicationWork(CosTransactions::Coordinator *coordinator)
{
    VISSessionManager::ConnectionPool_var pool;
    //get the ConnectionPool reference
    try
    {
        CORBA::ORB_var orb = CORBA::ORB_init();
        CORBA::Object_var initRef =
            orb->resolve_initial_references("VISSessionManager::ConnectionPool");
        pool = VISSessionManager::ConnectionPool::_narrow(initRef);
    }
    catch (CORBA::Exception &ex)
    {
        cout << "Corba exception obtaining reference to ConnectionPool"
            << endl;
        cout << ex << endl;
        throw ApplicationException();
    }
    //Declare the Connection_var on the stack to ensure that it destructs.
    VISSessionManager::Connection_var conn;
    Lda_Def *lda_ptr = 0;
    try
    {
        // Ask the pool for a database connection
        // Use the database profile "quickstart"
        conn = pool->getConnection("quickstart");

        // get a connection handle to use for native Oracle OCI calls
        lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
    }
    catch(const VISSessionManager::ConnectionPool::ProfileError& ex)
    {
        // we received an error with this profile.
        cerr << "Profile error:\n";
        << " " << ex.code
        << ": " << ex.reason
        << endl;
    }
}
```

```

        throw ApplicationException();
        // This would be something an application would define.
    }
}
catch(const VISSessionManager::Error& ex)
{
    cerr << "Session Manager error:\n";
    // print out all the error messages
    for(CORBA::ULong i = 0; i < ex.info.length(); i++)
    {
        cerr << " " << ex.info[i].subsystem
            << "-" << ex.info[i].code
            << ": " << ex.info[i].reason
            << endl;
    }

    throw ApplicationException();
    // This would be something an application would define.
}

//use lda to access Oracle data.
...

// If they got here, no unhandled exceptions occurred.
// Release the connection successfully
conn->release(VISSessionManager::Connection::MarkSuccess);
}

```

XA implementation issues

The XA implementation supports full participation in VisiTransact transactions. When using the XA implementation of the Session Manager, some tasks are different than when you are using the DirectConnect implementation. This section provides information about XA-implementation issues.

Completing or recovering a transaction

The Session Manager automatically registers the XA Resource Director with the VisiTransact Transaction Service during the `getConnection()` call. The Resource Director is ready and waiting for transaction completion (commit or rollback). Once all of the work of the transaction is done and the application invokes `commit()` or `rollback()`, the VisiTransact Transaction Service calls the Resource Director to either commit or rollback the transaction. Depending on the circumstances, the Resource Director may coordinate recovery. The Resource Director handles all recovery between XA Resources (the databases) and the VisiTransact Transaction Service without administrator intervention.

For more information about transaction completion and two-phase commit, see [“Transaction completion”](#) and [“Coordinating transaction completion with Resource objects.”](#) For more information about the Resource Director, see [“Session Manager overview.”](#)

DirectConnect implementation issues

When using the DirectConnect implementation of the Session Manager, some tasks are different than when you are using the XA implementation.

In DirectConnect transactions, the connection has one of the following type of states:

- Available and unassociated. The connection is available for any transaction.

- Available, but associated with a particular transaction. This connection is unavailable for other transactions, but can be acquired and used for the same transactions.
- In use. This connection is unavailable for other threads or transactions.

These last two states provide behavior that serializes access for clients. Access needs to be serialized because two different threads should not be able to use the same connection at the same time. Therefore, two different threads cannot do work on one of these DirectConnect connections at the same time for the same transaction.

Since the transaction state or Resource state is being maintained in a single process, if any element fails, the transaction is rolled back. If the failure occurs during the commit phase, you may not be able to tell if the transaction was committed or not. The `commit()` may receive either `CosTransactions::HeuristicHazard` or `CORBA::TRANSACTION_ROLLEDBACK`, depending on whether the VisiTransact Transaction Service knows what happened.

Completing or recovering a transaction

For DirectConnect transactions, the commit process is a single phase commit. The Resource that is involved is a single phase Resource embedded in the Session Manager. The connection that has been doing the work up until the time to commit will be available in that process (as long as the process stays up) to commit the transaction's work. Once the VisiTransact Transaction Service has been told to commit, it will tell the Resource to perform a single-phase commit. Once the commit happens, the connection is freed and returns to the pool to do work for a different transaction.

If the application server containing the DirectConnect Session Manager goes down, the single phase Resource is forgotten and the transaction is rolled back.

Note

The application must have already invoked `release()` or `releaseAndDisconnect()` before the commit so that the connection can be freed up.

Changing from DirectConnect to XA

If you originally develop your application for a DirectConnect environment and then want to use it in an XA environment, there should be no code changes necessary. There is just one basic rule that you should follow: Conform to the programming restrictions for both DirectConnect and XA. The only change necessary to convert from DirectConnect to XA is to use a connection profile configured for the XA implementation of the Session Manager. You must then deploy an XA Resource Director if one is not already deployed for that database.

16

Pluggable Database Resource Module for VisiTransact

The pluggable database resource module, a.k.a. Pluggable Resource Interface, is a component that implements a set of predefined interfaces to allow transactional applications to use databases as their persistent storage in transactions managed by Borland VisiTransact.

The examples given in this document are for the Oracle9i database, but you can use the Pluggable Resource Interface to manage transactions with the database of your choice.

Concepts

What is pluggable database resource module?

The pluggable database resource module is a component that implements a set of predefined interfaces to allow transactional applications to use databases as their persistent storage in transactions managed by Borland VisiTransact.

Developers who want to use a specific type of database have to implement the module. After a module has been properly implemented and compiled, the Session Manager's Connection Manager will load the module into application.

The pluggable modules are provided in the form of shared libraries. Once the Session Manager's Connection Manager loads a module in, it will interact with the module by the predefined interfaces. Those predefined interfaces enable the Session Manager's Connection Manager to carry out tasks necessary for a transaction, including: obtaining a physical connection from the supported database for the transactional application to manipulate data; informing the database of the association, dissociation, and the decision (commit or rollback) of a transaction; and disconnecting from the database when it is no longer needed.

The predefined interfaces are simple and standard-based, which support two kinds of transactions, that is, local transactions (Direct connections or DCs) and global transactions (XA connections). Direct connections are used when there is only one resource in a transaction. In that case it doesn't need to coordinate the commitment of multiple resources so VisiTransact will send commit or roll back directly to the database. XA connections are used when more than one databases are involved in a

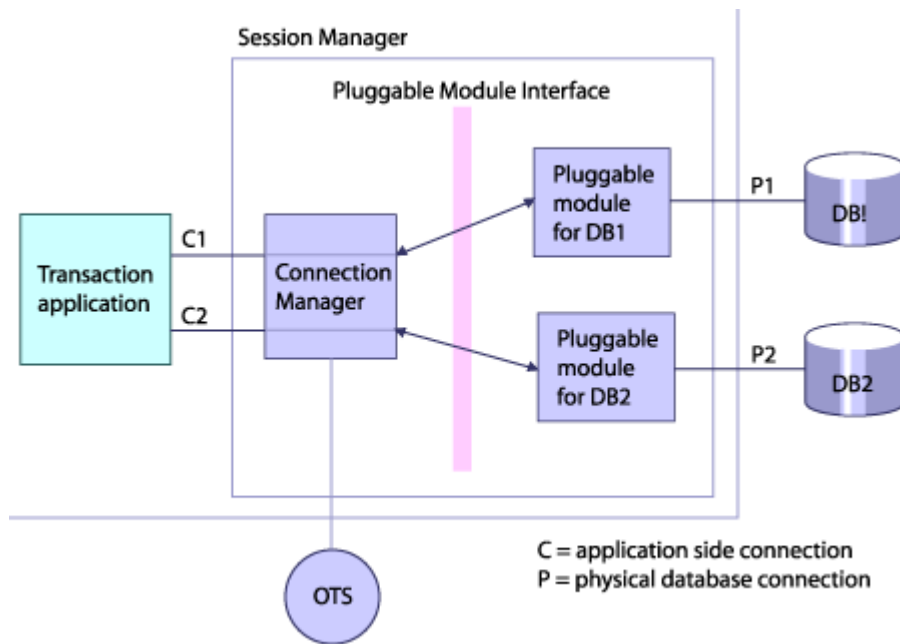
single global transaction. VisiTransact uses the XA interface defined in the X-Open specification to coordinate the databases to complete a global transaction.

Using pluggable module technique of Borland VisiTransact, a developer can easily integrate different databases into VisiTransact-managed transactional applications.

Structural descriptions

The following figure shows the basic model for the pluggable modules.

Figure 16.1 The pluggable database resource support module



The transaction application is a program that usually initializes transactions to carry out safe business tasks. The session manager's connection manager sits between the transaction application and the pluggable modules to take care of the connection and association issues for the access to a database in transactions. The various pluggable modules are loaded by the application in time of need, through which all specific databases can be reached.

The session manager's connection manager also caches the live connections for reuse, thus improving the performance. It also communicates with VisiTransact to register resource objects.

For XA connections a separate component, resource director, is used.

Connection Management

When a transactional application for the first time starts a transaction and get a connection with a specified profile name, the session manager's connection manager will load the right pluggable module into the process according to the profile. The session manager's connection manager makes connection to the database through the module and wraps the physical link into a standard connection object, and then associates it with the transaction. After that, it returns the connection object to the application.

Once the connection object is got, the application is safe to use it to update transactional data in the database. When a part of work is finished, the application must release the connection. This allows the session manager's connection manager to either recollect the resources allocated to the connection object or make the current connection available to other tasks.

Each time a connection object is created, it is associated with a specific configure profile which contains information necessary for the Session Manager Connection Manager to make a physical connection to a database. The Session Manager Connection Manager also associates some attributes with the connection object, such as transaction context, internal connection states, and timeout.

A connection obtained from the Session Manager Connection Manager is valid until it is released, disconnected, or finished with the completion of the associated transaction.

When a pluggable connection object is released, it is pooled for reuse. However the pooling and reuse mechanism is different between the DC and XA connections.

For a DC connection, the following table gives the detail about how a connection is used, pooled and reused.

Interface call (clients)	Session Manager Connection Manager	Plug-in module	Database
getConnection() getConnectionWithCoordinator()	It searches the pool for available connection, if found, associates it with the transaction, and then returns it. If no connection available in the pool, it will load the plug-in module in and get a connection through the module, and then create a new connection object and return it to the client.	It will be loaded into the session manager for the new connection. When the session manager reuses a connection, it doesn't call into the plug-in module.	It accepts the connect request from the plug-in module. Then the client may use the connection to update data in the tables.
release()	It dissociates the connection from the transaction and put the connection to the connection pool.	-	-
releaseAndDisconnect()	It first dissociates the connection with the transaction, rollback the current transaction and then disconnect from the database through the plug-in module.	It is called to rollback the transaction and disconnect from the database. (the rollback call is from VisiTransact)	It accepts the rollback request and the disconnect request from the plug-in module and release the connection.
hold()	It sets the connection to be in hold state. Any subsequent calls (except resume) will cause an exception. After the timeout expires, the session manager will restore the connection.	-	During the hold state, the database receives no request.
resume()	The connection is resumed and the client can use it again.	-	The database may receive native calls from the client to update the tables.
transaction commit (from VisiTransact)	-	The commit() interface will be called.	The database will commit any changes in this transaction.
transaction rollback (from VisiTransact)	The transaction is rolled back and the connection is put to the connection pool. (for reuse).	The rollback() interface will be called.	The database will roll back any changes made in this transaction.

For an XA connection, the following table shows the mechanism.

Interface call (clients)	Session Manager Connection Manager	Plug-in module	Database
getConnection() getConnectionWithCoordinator()	It searches the thread local pool for a reusable connection. If none available, create a new connection in the pool and associate the connection (as well as the thread) with the transaction.	To establish a XA connection, the xa_switch() interface will be called to get an pointer to the xa switch structure. Then the xa_open_string() will be called to obtain the string for the opening of a resource manager in the database.	An xa connection will be opened and associated with the current calling thread.
release()	The calling thread is dissociated with the transaction. The connection object is made available for reuse in the pool.	No specific interface will be called. The necessary calls will be made through the xa switch.	The opened resource will be suspended from the current transaction.
releaseAndDisconnect()	The calling thread is dissociated with the transaction and the connection is closed.	The xa_close_string() interface will be called for the string used to close the connection. Other necessary calls will be made through the xa switch.	The xa connection for the resource in the database is closed.
transaction commit (from VisiTransact)	The resource director will get a connection on behave of the transaction and complete the 2-PC commit using xa interfaces. The connection object will later be recollected in the pool and made ready for reuse.	No specific interface will be called. The necessary calls will be made through the xa switch.	The database will receive XA calls from the resource director and commit the changes made to the data.
transaction rollback (from VisiTransact)	The resource director will get a connection on behalf of the transaction and roll back the transaction using xa interfaces. The connection object will later be recollected in the pool and made ready for reuse.	No specific interface will be called. The necessary calls will be made through the xa switch.	The database will receive xa calls from the resource director and roll back the changes made to the data.

One major difference between XA and DC connections are their thread models. For DC connections, once the application get a connection from a thread, it can pass the connection object to any thread in the process as long as the specific database allows it to do so; for XA connections, the Session Manager Connection Manager obtains connections for different threads and then associates each connection object, as well as the thread who requires the connection, with the global transaction managed by VisiTransact. Passing an XA connection object across threads may get unexpected results and therefore is strongly discouraged.

Writing a Pluggable Module

The Connection Profiles

Each connection that the pluggable modules provides is associated with a configure profile, which contains the necessary information for the Session Manager Connection Manager to get a connection. This information is given in the following table:

Name	Value	Meaning
Profile name	a string of maximum length 63 (ASCII characters)	The profile name is the name of the file that stores these configuration information. Meanwhile it uniquely represents a kind of connections within the application.
Database type	a string of maximum length 63 (ASCII characters)	This is a informative string that tells which database the pluggable module supports.
version information	a string of maximum length 63 (ASCII characters)	Informative string showing the version info of the database.
pluggable module name	a string of maximum length 63 (ASCII characters)	The name of the pluggable module. The session manager will load the module of this name into the process when necessary.
connection parameter	a string of maximum length 256 (ASCII characters)	A string parameter the session manager passes to the <code>getITSDataConnection()</code> call in the pluggable module to get a new connection. This will give the module a way to customize different types of connections it can produce. (please refer to the example included in the product for detail.)

When an application calls `getConnection()` interface (defined in `VISessionManager.idl`), it must supply a configure profile name for the session manager connection manager to load the right module and make the connections. So before starting your application, the corresponding configure profiles must be created.

To create a profile, use the `smconfigsetup` tool included in the product.

To start the `smconfigsetup` tool, follow these steps:

- 1 Start `osagent`.
- 2 Start `smconfig_server`.
- 3 Start `smconfigsetup`.

After starting the `smconfigsetup` tool, it will give you a list of options that you can use to manage your profiles. Select option 7 to create a configure profile for the pluggable modules. Then you can follow the prompt to give all your information defined in the above table in order. The tool will save the profile in the specified location for session manager.

The Interface Definition

The interfaces that a pluggable module need to implement are defined in a single header file.

In this header file, a function that Session Manager Connection Manager used to get a connection object from and a connection class are defined. A pluggable module does not need to implement all the interfaces. Some of the interface are compulsory, some are optional, based on the type of connection the module is going to support.

The Single Function

Any pluggable module must implement this function.

The function `GetITSDataConnection()` is defined as follows:

```
extern "C"
ITSDataConnection* GetITSDataConnection(const char* param);
```

This function, when called by Session Manager Connection Manager, must return a new object that represents a new connection. If an existing connection is reused, the session manager connection manager never calls the function again for it. The function uses C calling convention.

This function takes a string as its only parameter. Users are free to specify this parameter in a configuration file to control the properties of the connections corresponding to the profile, if any. The Session Manager Connection Manager will get this parameter from the profile and passes it as the argument to this function.

The return value is a pointer to an object of type `ITSDataConnection` which contains the connection related interfaces that a pluggable module should implement.

This function can be taken as the entry point of a pluggable support module - the first call by the module loader (which is the Session Manager Connection Manager). The session manager connection manager relies on this function to get an object for connection management.

The ITSDataConnection class

This class is defined below.

```
class ITSDataConnection
{
public:
    virtual void connect() = 0;
    virtual void disconnect() = 0;
    virtual void rollback() = 0;
    virtual void commit() = 0;
    virtual xa_switch_t* xa_switch() { return 0; }
    virtual const char* xa_open_string() { return 0; }
    virtual const char* xa_close_string() { return 0; }
    virtual void* native_handle() { return 0; }
};
```

The methods in `ITSDataConnection` class can be divided into three groups:

- 1 native handle acquisition interface
- 2 local transaction connection and completion interface
- 3 global transaction connection and completion interface

Native handle acquisition interface

```
void* native_handle();
```

This function is used to get access to the native APIs for a database supported by the module. The return value is a void pointer, allowing the implementation to return anything necessary to manipulate data in the database. A transactional application can obtain this pointer through `getNativeConnectionHandle()`, in which the Session Manager Connection Manager will call the `native_handle()` and return the pointer back to the application.

Any pluggable module must implement this function.

Local transaction connection and completion interface

Pluggable modules that support the local transaction must implement these functions.

These four methods is used by Session Manager Connection Manager to inform the database of the start and completion of local transactions.

void connect();

When it is called, it establishes the connection to the database and tells the database that a local transaction begins.

void disconnect();

When it is called, it means the connection, if established, is no longer needed. So the connection can be closed.

void rollback();

It tells the database to commit the transaction.

void commit();

It tells the database to rollback the transaction.

Global transaction connection and completion interface

Pluggable modules that support global transactions must implement the functions.

The session manager uses X-open's XA interface to talk to a XA conformable database.

xa_switch_t* xa_switch();

All the Session Manager Connection Manager need from the pluggable module is a pointer to a xa_switch_t data structure which contains all the XA APIs as defined in the xa.h. The xa_switch() function is just for this purpose. Whenever being called, it must returns a valid pointer to this data.

Usually the specific database implements and exposes the xa_switch_t to its clients. The name of that data struct varies from database to database. For example, Oracle9i implements its xa_switch_t as a global variable named xaosw.

This function is also used by Session Manager Connection Manager to judge the type of a connection. If the function returns zero, the session manager will treat the connection as DC type, otherwise it takes the connection as XA type.

Pluggable modules that support global transactions must implement the function and must not return zero.

const char* xa_open_string();

When called, it returns a string used as argument to xa_open() call.

const char* xa_close_string();

When called, it returns a string used as argument to xa_close() call.

The two methods are called by the session manager to get database specific parameters to open or close a XA connection to a database. The returned string from the xa_open_string() call will be used in the call on xa_open() and the returned string from the xa_close_string() is used in xa_close().

Once called for an XA connection, the session manager will keep the returned values for later use. The implementation does not need to keep the validity of the returned pointer all the time.

Building and Running

- 1 Include the required header files.

Compiling the pluggable module doesn't need any VisiTransact specific libraries. However, you have to include the itsdataconnection.h in your source file. The xa.h is

a standard XA header that needed to be in your include path. Usually databases which support the XA interface provide the `xa.h` in their installation directory.

2 Make sure the needed libraries are available.

No VisiTransact specific libraries are needed during the compile time. However, you may need the database specific libraries. For example if you want to support the oracle9i database in your pluggable module and use OCI, you have to put the oracle9i client library in your library path and link it with your code.

3 Build the pluggable module.

The pluggable modules must be in form of shared libraries. Different compilers have different flags that control the build target type. Please refer to our examples to see the flags needed to build a shared library.

Running Applications using the pluggable modules

A transactional application need not have the knowledge of the pluggable modules. However, it may need to know the interface of the native handle for accessing data in databases. Building the application does not need to link with any pluggable modules as those modules, when needed, will be dynamically loaded into the process in the run time. Therefore, before you start the application make sure that the pluggable modules are in the library path so that the Session Manager Connection Manager can successfully load them in.

Programming restrictions

When using the DirectConnect profile, the following operations should not be called in the application: “Connection Operation”, “Transaction Operations”, and “Implicit Operations”. Similarly, there is another set of programming restrictions when using the XA profile. See “[Programming restrictions](#)” and “[Programming restrictions](#)” for more information.

Known limitations

In any plug-in session manager, the `VISessionManager::Connection::isSupported()` API will have static return values. In the case of DC connection, `isSupported("hold")` will return true and `isSupported("thread_portable")` will return true. In the case of XA connection, `isSupported("hold")` will return false and `isSupported("thread_portable")` will return false.

In any plug-in session manager, the `VISessionManager::Connection::getInfo("version_rm")` will now return NULL. As the information is not applicable in the case of a plug-in session manager.

17

Using the VisiBroker Console

This chapter provides information about the VisiBroker Console including: managing transactions with the VisiTransact Transaction Service, tracking heuristic completions, viewing error messages, and configuring a connection to a database for data access using the Session Manager Configuration Server.

Overview of the VisiBroker Console

The VisiBroker Console provides you with an easy way to graphically monitor transaction status, view heuristic logs, view message logs, and configure database access. The VisiTransact functionality of the VisiBroker Console is divided into the following three sections:

- Transaction Services section
- Session Manager Profile Sets section

Transaction Services section

The Transaction Services section lets you manage VisiTransact Transaction Service instances and their transactions over the network. Also, you can monitor the status of and control the completion of transactions running under the VisiTransact Transaction Service instance you choose to monitor.

When you select the Transaction Services section, it displays all the instances of the VisiTransact Transaction Service that are running at that particular OSAGENT_PORT.

When you select an instance of the VisiTransact Transaction Service, the right panel displays three tabs. You can move between these tabs to:

- **View transactions.** The Transactions tab lets you view information about the transactions for this instance of the VisiTransact Transaction Service. You can also view transaction details, force the transaction to roll back or commit, or shutdown the VisiTransact Transaction Service.
- **Monitor heuristics.** The Heuristics tab lets you view information about heuristic outcomes for VisiTransact transactions. You can view the specific details for each participant of a transaction that had a heuristic outcome. For more information about heuristics, see [“Managing heuristic decisions.”](#)
- **View log messages.** The Message Logs tab lets you view the error, warning, and information messages in the Message Log for all VisiTransact components on the same node as that instance of the VisiTransact Transaction Service.

Session Manager Profile Sets section

The Session Manager Profile Sets section lets you create, configure, and edit connection profiles for the Session Manager. The Session Manager provides pre-configured database connections to applications and XA Resource Directors.

Starting the VisiBroker Console

Before you start the VisiBroker Console, make sure that you start either an instance of an VisiTransact Transaction Service, or an instance of an Session Manager Configuration Server. See [“Starting a VisiTransact Transaction Service”](#) and [“Starting the Session Manager Configuration Server”](#) for instructions.

Starting a VisiTransact Transaction Service

The VisiTransact Transaction Service may have been started by the OAD. However, you can start the Transaction Service manually by using this command:

```
prompt>ots
```

For a complete list of the options for the OTS command, see [“ots”](#).

Note

If the VisiTransact Transaction Service you want to manage is not running or is not on your network, then it will not appear in the list of VisiTransact Services you can manage with the VisiBroker Console. You can use the **osfind** utility to determine if the instance of the VisiTransact Transaction Service is running on your network.

Starting the Session Manager Configuration Server

The Session Manager Configuration Server may have been started by the OAD. However, you can start the Session Manager Configuration Server manually by using this command:

```
prompt>smconfig_server
```

For a complete list of the options for the **smconfig_server** see [“smconfig_server”](#).

Launching the VisiBroker Console

In Windows, you can click the Borland Management Console icon, found in the Borland Deployment Platform program group, and select the VisiBroker icon in the left-side navigation bar on the Console.

Alternatively, at the command prompt in either Windows or UNIX, type the following command:

```
vbconsole
```

The VisiBroker Console screen displays.

Using the Transaction Services section

You use the features in the Transaction Services section to monitor and manage transaction information for the VisiTransact Transaction Service you selected, to resolve the status of a transaction by committing or rolling back the transaction, and to shutdown an instance of the Transaction Service, monitor heuristics, and view messages.

For more information on using the Transaction Services section, see one of the following sections:

- Locating an instance of the Transaction Service
- Monitoring transactions
- Refreshing the transaction list
- Displaying details for specific transactions
- Controlling specific transactions
- Filtering the transaction list
- Viewing heuristic transactions
- Viewing heuristic details
- Viewing the message log
- Filtering the message logs

Locating an instance of the Transaction Service

To view a list of transactions for a specific VisiTransact Transaction Service, you must choose from the list of VisiTransact Transaction Service instances running on your network. If you want to switch to another instance, you can select a VisiTransact Transaction Service from the ones listed under Transaction Services. However, you can view only the transactions for one instance of the VisiTransact Transaction Service at a time in the Transactions tab.

Monitoring transactions

The Transactions tab displays a list of transactions for the VisiTransact Transaction Service you select. This lets you keep track of which transactions have not completed. From this list of transactions, you can view the current status and periodically, you can refresh the view to track the most current information. Use the following instructions to monitor transactions:

- 1 Select the instance of the VisiTransact Transaction Service you want to monitor from the list under Transaction Services. A list of transactions for that instance displays in the table.

The list provides you with the following information for each transaction:

- Transaction Name
- Status
- Host of the Transaction Originator
- Age in seconds
- The number of times transaction completion was attempted

- 1 To sort the list of transactions in ascending order, click on the header of the column you want to sort.

The status bar at the bottom of the screen provides additional information about the number of transactions, filtering status, and current system activity.

Refreshing the transaction list

You can refresh the list of transactions by clicking the Refresh button on the toolbar in the main window. Transactions that have completed will no longer display in the table.

Displaying details for specific transactions

You can use the Transactions tab to view the details of a specific transaction. This detailed information can provide you with insight so you can resolve the status of the transaction.

To view detail for a specific transaction select the row from the list of all the transactions.

The bottom table in the Transaction view tab displays the following information for each participant in the selected transaction:

- IOR of the Participants
- Host of the participant
- Vote for Prepare
- Outcome

The PrepareVote column contains the participant's vote. The possible values are:

- Unknown
- ReadOnly
- Commit
- Rollback

The last three values will only appear if the transaction has been prepared.

The Outcome column contains the outcome of the commit phase for a participant. The possible values are:

- None
- Commit
- Rollback
- HeuristicCommit
- HeuristicRollback
- HeuristicMixed
- HeuristicHazard

Note

For more information on heuristics and the heuristic logs, see [“Managing heuristic decisions.”](#)

Controlling specific transactions

In the Transactions tab, you can resolve the status of a transaction that has not completed by using the Force Rollback or Stop Completion functions.

- **Force Rollback** can be used on any transaction that has not finished the prepare phase of completion.
- **Stop Completion** tells the VisiTransact Transaction Service to stop attempting to complete the transaction. To stop the completion of a transaction, click Stop Completion.

Resolving hung or in-doubt transactions

If the transaction does not complete in a reasonable amount of time or if the outcome is in doubt, you can resolve it by either stopping or rolling back the transaction. Use the following instructions to resolve the transaction:

- 1 Use the instructions in [“Displaying details for specific transactions”](#) to view the details of the transaction in question.
- 2 Resolve the transaction by choosing either to rollback or to stop committing.

- To rollback the transaction, click Force Rollback.

If the transaction has progressed beyond the prepare phase of completion but still exists, then the Transaction Is In Second-Phase dialog box displays indicating that you cannot rollback the transaction.

If the transaction no longer exists, then the Transaction Not Found dialog box displays indicating that the completed transaction cannot be rolled back.

- To stop the completion of a transaction, click Stop Completion.

If the VisiTransact Transaction Service does not know about this transaction (for example, possibly because it has already completed) then the Unknown Transaction dialog box displays indicating that you can no longer resolve the transaction.

Filtering the transaction list

You can filter the transactions that you view in the Transactions tab by clicking Filter Transactions.

The Filter dialog box lets you set the filter value option in minutes and seconds. Only those transactions older than the age filter value will be displayed.

To cancel the filter option, click Filter Off from the Transactions tab.

Viewing heuristic transactions

The Heuristics tab lets you view the transactions that had heuristic outcomes and were placed in the heuristic log. To view this information:

- 1 Click the Heuristics tab.

VisiTransact produces one heuristic log per instance of the Transaction Service, referred to as the **heuristic.log** file. By default, the path to this directory is <VBROKER_ADM>/its/transaction_service/<transaction_service name>/heuristic.log. The Heuristics tab provides the following information for each item listed:

- transaction's name
- originator
- time of the heuristic outcome

- 1 To cancel the list of transactions from loading in the tab, click Cancel Refresh.

Viewing heuristic details

You can use the Heuristics tab to view the details of a specific heuristic completion. These details provide you with information so you can resolve the transaction.

To view the detail for a specific heuristic decision, select the row in the list of all the heuristic decisions.

The bottom table in the Heuristics view tab displays the following information for the selected transaction:

- Participant
- Host
- PrepareVote
- Outcome
- Last Exception (This is the last exception that occurred before the heuristic occurred.)

Note

The number appearing in the header corresponds to the heuristic's place in the list of heuristics. The number for the first heuristic in the list is 1 (one).

The PrepareVote column contains the participant's vote. The possible values are Commit or Rollback. The Outcome column contains the outcome of the commit phase for a participant. The possible values are HeuristicCommit, HeuristicRollback, HeuristicHazard, HeuristicMixed, or None, which means that there were no heuristics for that participant.

Note

For more information on heuristics and the heuristic logs, see [“Managing heuristic decisions.”](#)

Viewing the message log

A message log is created for every physical machine running a VisiTransact Transaction Service. The message logs are located in `<VBROKER_ADM>/its/message.log`.

To view a list of messages for the node on which the selected Transaction Service resides, you must use the Message Log tab. The Message Log displays these messages:

- Error
- Information
- Warning

Filtering the message logs

You can filter the messages in the message log by clicking Filter Messages.

This displays the Filter dialog box.

You can filter messages as follows:

- By specifying a time window
- By specifying the type of message to be displayed
- By entering other criteria in the Source, Category, Code or Host fields (information entered in these fields is case sensitive)

To stop filtering, click Filter Off.

Trimming the message log

You can permanently remove entries from the message log. All messages older than the date and time you select will be removed permanently from the message log. To trim the message log:

- 1 Click Trim Message Log to view the Trim dialog box.
The Trim dialog box opens.
- 2 Set the date and time criteria and click OK. The date reads as month, day, and year.
The time reads as hours, minutes, and seconds.

Using the Session Manager Profile Sets section

The Session Manager Profile Sets section lets you access the Session Manager Configuration Server. Do not confuse this with the Session Manager.

The Session Manager Configuration Server reads and writes connection profiles to the persistent store file. Once you have gained access to the Session Manager Configuration Server, you can create and configure a new connection profile or edit an existing connection profile. For more information about the Session Manager Configuration Server, see [“Integrating VisiTransact with databases using the Session Manager.”](#) For more information on using the VisiBroker Console to configure connection profiles, see the following sections:

- Gaining access to the Session Manager Configuration Server
- Creating and configuring a new connection profile
- Editing an existing connection profile

What are connection profiles?

A connection profile consists of all the required connection attributes to make a connection to a particular database. For more information, see [“Session Manager overview.”](#)

Attributes required to create Session Manager Configuration Profiles are specific to the type of database to which you are connecting. For example, to integrate VisiBroker VisiTransact with an Oracle DBMS, you must use the Session Manager for Oracle OCI. Then you may create a connection profile using a combination of attributes that are common to all databases, plus specific ones for Oracle.

Currently, the Session Manager provides connectivity with Oracle9i databases. See [“XA Session Manager for Oracle OCI, version 9i Client”](#) and [“DirectConnect Session Manager for Oracle OCI, version 9i Client”](#) for more information.

The Pluggable Resource Interface provides the capability to create a Session Manager to integrate with the database of your choice. See [“Pluggable Database Resource Module for VisiTransact”](#) for more information.

Gaining access to the Session Manager Configuration Server

Make sure that you have started the Session Manager Configuration Server, as described in [“Starting the Session Manager Configuration Server”](#).

To gain access to the Session Manager Configuration Server, choose a Session Manager Configuration Server under Session Manager Profile Sets.

The profile names and details for connection profiles managed by the selected Session Manager Configuration Server appear in a table in the right hand panel.

Creating and configuring a new connection profile

Before you can create a new connection profile, you must gain access to the Session Manager Configuration Server, as described in [“Gaining access to the Session Manager Configuration Server”](#).

Note

Alternatively, you can use the `smconfigsetup` utility to create connection profiles. See [smconfigsetup](#) for more information.

To create and configure a new connection profile, perform the following steps:

- 1 To create a new profile, click New in the right hand panel.

The New Profile dialog box appears.

Note

You can create a new profile based on an existing one by selecting the profile you wish to copy and clicking Duplicate. You can keep the same attribute settings or change them. However, you must give this new profile a unique name.

- 2 Enter a unique profile name in the New Profile Name field.
- 3 Select a database type from the DB Type drop down list box.

The database type indicates the kind of database and transactional access type for the connection profile. Each database type has particular attributes associated with it. Entering the database type will determine which attributes display in the Connection Profile Editor screen.

- 4 Click OK.

The Connection Profile Editor dialog box appears.

- 5 Fill in the Database Name field.

This field requires a value that is database dependent.

- 6 In the User Name and Password fields, enter the user name and password for the database.
- 7 Click Save.

Once you have saved, the values are written to the persistent store file and any Session Manager or XA Resource Director that has access can read from that file.

Editing an existing connection profile

Before you can edit an existing connection profile, you must gain access to the Session Manager Configuration Server. For information on how to gain access, see [“Gaining access to the Session Manager Configuration Server”](#).

To edit an existing connection profile, perform the following steps:

- 1 To edit an existing profile, select the profile you want to edit from the list.
- 2 Click Open.

The Connection Profile Editor dialog box appears.

- 3 Make your changes in the Connection Profile Editor screen
- 4 Click Save.

See [“Modifying connection profiles used by Session Manager clients”](#) for details on when the new values will take effect.

Filtering the connection profiles

When you first access connection profiles for a Session Manager Configuration Server, it displays profiles that are common to all database types.

- 1 To filter the profiles by database, select a database type from the DB Type drop down list box.

Only profiles for the selected database type display in the tab.

Each database type has particular attributes associated with it. For example, if you select Oracle as the DB Type, you will only see the profiles (containing attributes) associated with Oracle databases. If there are attributes specific to Oracle databases, columns are added to the right of the common attribute columns.

- 2 If you want to see profiles for all databases, you can select All in the DB Type drop down list box.

Deleting a connection profile

To delete a connection profile, select it and click Delete. A confirmation dialog box appears to confirm your decision.

Refreshing the list of connection profiles

Click Refresh to update the list of connection profiles for the selected Session Manager Configuration Server.

18

Server Application Model

This section describes the server application model supported by VisiTransact, as well as the XA configuration. With this model, the transaction logic becomes transparent to application business logic.

Server Application transaction and database management

The OMG Object Transaction Service (OTS) specification version 1.4 or below standardizes the following aspects of distributed transactional applications:

- A CORBA distributed transaction application management model, which is an OMG version of the X/Open DTP model, expressed in form of OMG IDL.
- A transactional interoperable protocol, in terms of IDL interfaces and service context, between application clients, transaction coordinators and participating transactional resource managers in the DTP model. This protocol is supported by various OTS implementations and by JTS implementations.
- An implicit and an explicit transactional application programming model, presented in IDL interfaces and local objects (CosTransactions::Current).

OTS does not address application server-side database integration and the implicit transaction programming model. On the server side, by OMG OTS specification, applications are responsible for database connection and transaction control explicitly through the XA interface.

Using the ITS server application model, transaction control (and database connection) does not need to be handled by servant business logic implementation, but becomes an attribute setting specified as POA creation policy.

Requirements before reading this section

This section assumes the following knowledge:

- **Database and Embedded SQL** You must know how to use database tools (such as Oracle sqlplus) to create, browse and manipulate database tables. You should be able to program in embedded SQL, and build applications with the database-provided embedded SQL to C++ pre-compiler (such as Oracle proc).
- **XML and DTD** You must know how to use XML to describe XA configurations and you must understand Data Type Definition (DTD).

- **OMG and the Distributed Transaction Process (DTP) of X/Open** You should know the DTP architecture concepts and terminology, as well as the client side implicit transaction programming model (using the transaction `Current` interface to start and end a transaction). You do not need to read through and understand the entire OMG OTS specification.
- **XA and database connection configuration** You must be able to make minor modifications to XA and database configurations.

Understanding the terminology of Container Managed Transaction (CMT), a concept of EJB and CCM, and the concepts of implementation, deployment, and application assembly in either EJB or CCM will help you use the information in this section.

Concepts and terminology

The following terminology is used in this section:

- **Client** A CORBA application. See Client-initiated transaction (CIT) below for more information.
- **AP** A CORBA client application that can initiate a transaction.
- **Server** A CORBA server application that implements business logic. See RM and Server-initiated transaction (SIT) below for more information.
- **TM** A Transaction Manager that coordinates global transactions. Typically, it is a stand-alone server process (such as a VisiTransact OTS server). In-proc TM is also supported in VisiTransact (ots.dll/so), but is not recommended.
- **RM** A Resource Manager. In OMG OTS, RM usually refers to a database server. RM can also refer to an application server that uses SQL to access a database.
- **1PC** One phase commit, involving a single RM, committed without a preparation stage.
- **2PC** Two phase commit, involving multiple RMs, committed with a preparation stage.
- **Global transaction** A transaction that can involve multiple RMs. Usually, but not necessarily, a global transaction needs to be coordinated by a TM and use a 2PC protocol to commit. See Local Transaction Optimization (LTO) below for more information. By default, all transactions described in this section are global transactions (initiated either by the client or the server).
- **Local transaction** A transaction, which only involves one single RM, restricted in one thread of control, and not coordinated by TM.
- **Client-initiated transaction (CIT)** Also known as a Client Demarcated Transaction, this term refers to a global transaction initiated and terminated by a client. Client-initiated transactions must be coordinated by a TM.
- **Server-initiated transaction (SIT)** Also known as a Server Demarcated Transaction, this term refers to a global transaction initiated and terminated by a server PMT engine transparently when handling a client request. The boundary of this transaction is the given client request. The transaction starts before performing the business logic, and ends before replying to client. A server-initiated transaction is global, but not necessarily coordinated by a remote TM.

Note

SIT is a popular transaction model that is well documented and widely used in EJB and CCM (CORBA Component Model)

- **Local Transaction Optimization (LTO)** A technique that allows a server to initiate and terminate a global transaction locally without involving a TM, if it only accesses one local RM (database). Server would only export a SIT to TM and evolve it into a true global transaction when server makes a forward call to a second RM (another transaction object outside the process). J2EE document presents a possible

scenario of LTO, but not applicable to OTS implementation. Therefore, VisiTransact's LTO uses a different technology to ensure it is OMG compliant, interoperable and applicable to other OTS implementations.

- **PMT POA Managed Transaction.** A server side transaction and database integration engine. PMT separates and hides all database connection and transaction details from application business logic developers. With PMT, servant implementation business logic does not need to hardcode database connection and transaction logic within its implementation. Database connection and transaction control are independent of business logic, and can be configured and reconfigured during application assembling. With PMT, a given business logic implementation can involve both CIT and SIT. In addition, different operation signatures of a given object can be configured with different transaction attributes.
- **CosTransactions::Current** A single object for client applications, in an implicit transaction model, to initiate and manipulate thread-specific global transaction. A server implementation can also use this object to retrieve thread-specific information about the global transaction it currently involves. The CosTransactions::Current can be retrieved from ORB using the `resolve_initial_references()` method.
- **PMT::Current:** A single object for a server application to retrieve information about thread specific database connection and transaction arranged by PMT engine. The retrieved information, such as connection name, is necessary for the SQL AT clause. Other retrieved information is useful for PMT diagnosis. PMT::Current object instance can be retrieved from `PMT::Current::instance()`.
- **XA:** An API standardized by X/Open. XA API drivers (usually shared libraries) are provided by database vendors. PMT uses these drivers to manipulate database connections and transactions. With PMT, XA (as well as database connections and transactions) are transparent to application developers. It is the responsibility of the application assembler to configure XA correctly. See "[XA resources configuration](#)" for more information.
- **Session Manager (SM)** A service used in previous releases. To use this service in this release, you must set the `vbroker.its.its6xmode` property to `true`. See "[VisiTransact properties](#)" for more information.
- **Resource Director (RD)** A service used in previous releases. To use this service in this release, you must set the `vbroker.its.its6xmode` property to `true`. See "[VisiTransact properties](#)" for more information.

Scenarios of global transaction and PMT

Client-initiated global 2PC and 1PC transactions

In the OTS and X/Open DTP model, a distributed transaction means a global transaction initiated by a client. In this case, a client contacts a TM to initiate and end

(commit or rollback) a transaction, regardless of whether it is 2PC (figure 1) or 1PC (figure 2).

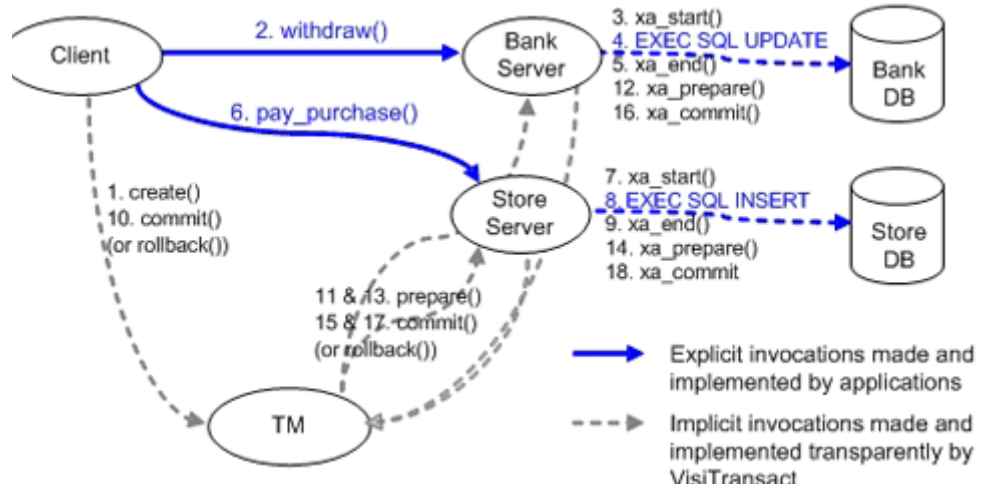
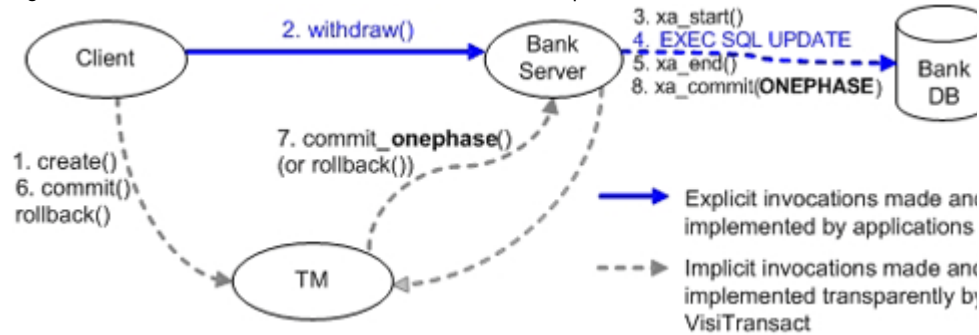


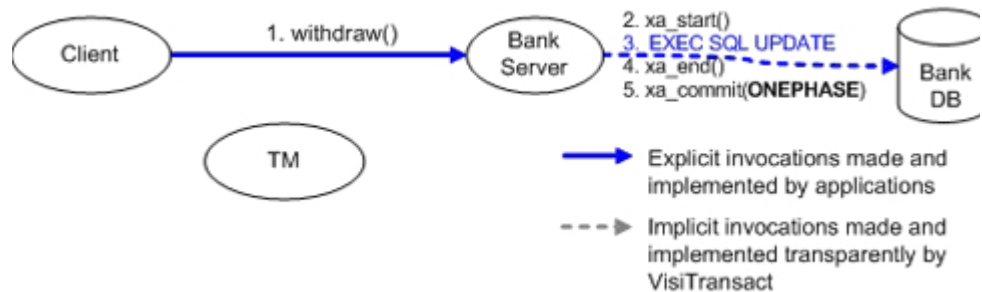
Figure 18.1 Client-Initiated Transaction with Two RM Participants



Transparent server-initiated transactions with PMT

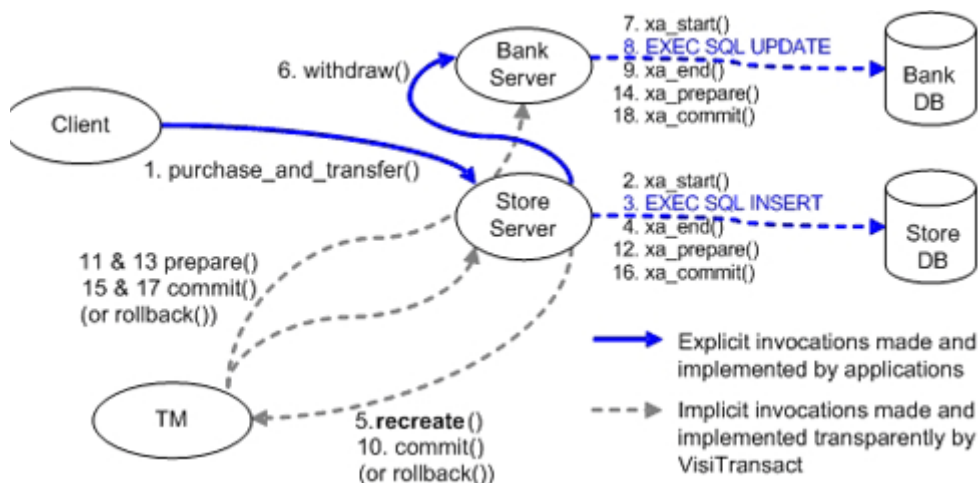
The PMT hides all transaction aspects from business logic developers and provides an optimized way to perform client-initiated or server-initiated global transactions.

For example, in a server-initiated transaction, PMT uses the local transaction optimization to initiate a global transaction locally, as shown in the following figure.



The locally-initiated global transaction is exported to an external TM before the server makes a forward invocation to another external transactional object, as shown in step 5 of the following figure.

Figure 18.2 Server-Initiated Transaction with One RM Participant, with Local Transaction Optimization



PMT completely hides database access and transaction details. The server side implementation only needs to write Embedded SQL (or ODBC/CLI) to access an unspecified database, without specific database connection and transaction control statements. All database connection and transaction management tasks are performed, implicitly, by PMT integrated in the server-side ORB and POA engine. Consequently, the `deposit()` code is simplified, as shown in the following example:

```
void BankImpl::deposit(const char* id, float amount) {
    EXEC SQL BEGIN DECLARE SECTION;
        const char* account_id = id;
        float deposit_amount = amount;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL UPDATE account_table
        WHERE account_id = :account_id
        SET balance = balance + :deposit_amount;
}
```

There is neither connection management nor transaction management code. This same business logic implementation can be used transparently in either client-initiated or server-initiated transactions.

PMT overview

PMT is set up programmatically. The server setup and transaction attribute configuration must be arranged in conventional POA creation code, namely POA creation policy.

PMT is modeled after the widely used Container Managed Transaction (CMT) of EJB and CCM. Therefore, most CMT concepts and features are directly applicable to PMT.

Note

Applications should not explicitly suspend or resume a transaction or obtain a transaction coordinator/terminator on a POA implicit managed transaction (PMT).

PMT transaction attribute values

In PMT, servant implementations only implement business logic. Details of transactions they are going to involve are determined by the transaction attribute assigned to specific objects and methods described in the policy of the POA `PMT_ATTRS_TYPE`. Using the POA policy, you can configure the transaction attributes as follows:

- *PMT_NotSupported*
The propagation context is not copied to transaction current. POA neither joins the client's transaction (T1), nor starts a server to initiate a new global transaction (T2). This is the default PMT attribute. This setting should be used for non-transactional methods to avoid the overhead of associating a current worker thread with a global transaction.
- *PMT_Required*
POA joins or propagates the client-initiated global transaction (T1), if the request from the client carries a global transaction context. Otherwise, POA initiates and ends a new global transaction (T2) for that request. This is the most useful PMT attribute setting for transactional methods. It ensures the business logic will always be performed with an XA connection and within a transaction. This attribute is referred as "AUTOTRAN" in classic TP products.
- *PMT_Supports*
POA joins or propagates the client-initiated global transaction (T1), if the request from client carries a global transaction context. Otherwise, if the request is not within a CIT, the POA does not start a transaction. When combined with a null XA resource (described in "[XA resources configuration](#)"), this PMT setting is typically used for transaction propagation.
- *PMT_RequiresNew*
POA does not join or propagate a global transaction of a client, but always initiates and ends a new global transaction (T2) on each client request. To improve performance, use this PMT setting for all business logic with only a perform read (query) operation on backend databases.
- *PMT_Mandatory*
POA always joins or propagates the global transactions of a client, if it is in a context. Otherwise, if the client did not start a transaction, POA raises an exception.
- *PMT_Never*
POA raises an exception if it is in a client transaction context.

Note

PMT only applies to remote requests. Co-located invocations (although dispatched via POA), remain in client's transaction, if any, regardless of the PMT setting. When POA initiated server transaction (T2) has not yet been exported to an external TM, calling method on transaction Current within servant implementation is prohibited.

The following table summarizes the PMT transaction attribute mode and its semantic behavior.

PMT Attribute Mode	Client's Transaction	POA's Transaction
NotSupported	None T1	None None
Required	None T1	T2 T1
Supports	None T1	None T1
RequiresNew	None T1	T2 T2
Mandatory	None T1	TRANSACTION_REQUIRED T1

PMT Attribute Mode	Client's Transaction	POA's Transaction
Never	None T1	None INVALID_TRANSACTION

In a programmatic approach, an application can specify transaction attributes for a given POA and for given objects, at POA creation time, by specifying the PMT attribute policy.

The value of the PMT attribute POA creation policy is a sequence of PMTAttr structure, defined as follows:

```

module VISTransactions {
    ...
    enum PMTMode {
        PMT_NotSupported = 1,
        PMT_Required = 2;
        PMT_Supports = 3;
        PMT_RequiresNew = 4;
        PMT_Mandatory = 5;
        PMT_Never = 6;
    };

    struct PMTAttr {
        CORBA::OctetSequence    oid;

        string                    method_name;
        PMTMode                    mode;
        string                    xa_resource;
    };

    typedef sequence<PMTAttr>    PMTAttrSeq;
};

```

In this definition of a PMTAttr structure:

- The **oid** field specifies the ID of the object this PMT attribute setting applies to. If the oid is an empty sequence (zero length), this attribute setting applies to all objects of this POA. See the list of dynamic rules for more details.
- The **method_name** field specifies the request's operation name this PMT attribute setting applies to. If method_name is set to *, this attribute setting will apply to all request operations sending to objects of this POA.
- The **mode** field specifies the mode of this PMT mode attribute setting.
- The **xa_resource** field specifies the name of a preconfigured XA resource to be associated. For more information, see [“XA resources configuration”](#). If this field is an empty string or null, a PortableServer::POA::InvalidPolicy exception is raised on create_POA(). Literal `null` is a special reserved xa-resource name. This name cannot be used to name a physical XA resource in an XA resource descriptor, but can only be used as value of the xa_resource field of PMTAttr. When a request condition matches one of the given PMTAttr with xa_resource field equal to null, the PMT engine does not associate the request processing worker thread with any physical XA connection. Instead, the PMT engine only ensures that the associated OTS context is propagated if the servant implementation method makes a forward invocation to the next tier. See the `oci` example in the `<installation_directory>\examples\vbroker\Transaction` directory.

It is possible that none, one, or two PMT attribute settings apply to a request. The PMT engine uses the following rules, in sequence, to decide which PMT mode or attribute should be applied:

- 1 For oneway methods, pseudo methods, or IDL interface attribute set/get methods, PMT mode NotSupported is applied, regardless the PMT attribute setting.

- 2 PMT attribute, its oid field exactly matches the request target's object id and its method_name field exactly matches the request's operation name, is applied.
- 3 PMT attribute, its oid field exactly matches the request target's object ID, and has a wildcard (*) in the method_name field, is applied.
- 4 PMT attribute, its method_name field exactly matches the request's operation name, and has an empty oid field (indicating a wildcard), is applied.
- 5 PMT attribute, which has an empty oid field (indicating a wildcard) and a wildcard (*) in the method_name field, is applied.
- 6 PMT mode NotSupported is applied if none of the other rules applies.

PMT is independent of OMG standardized POA OTS policy. A Server side transaction engine first checks the target POA's OTS policy against the received request context, and decides whether to raise an OMG-specified exception (INVALID_TRANSACTION or TRANSACTION_REQUIRED). If no exception is raised, the request is forwarded to PMT.

- If an OTS policy is not specified as an attribute of the POA creation element, and a PMT policy is specified (other than with an empty PMTAttr sequence), it implies OTS ADAPTS.
- If neither an OTS nor a PMT policy is specified (or if the PMT policy is specified with an empty PMTAttr sequence), it implies an OTS policy of NONE, and the OTS component is not added to exporting IORs.

A simple example

```

CORBA::PolicyList policies;
policies.length(1);

PortableServer::ObjectId_var objId=
PortableServer::string_to_ObjectId("account_object");
PMTAttrSeq pmt_seq;
pmt_seq.length(1);

pmt_seq[0].oid           = (CORBA::OctetSequence&) objId;
pmt_seq[0].method_name   = (const char*)"withdraw";
pmt_seq[0].mode          = VISTransactions::PMT_Required;
pmt_seq[0].xa_resource   = (const char*)"account_storage";

CORBA::Any policy_value;
policy_value <<= pmt_seq;
policies[0] = orb->create_policy(VISTransactions::PMT_ATTRS_TYPE,
policy_value);

// Create myPOA with the right policies
PortableServer::POA_var myPOA = rootPOA->create_POA("account_server_poa",
poa_manager,
policies);

```

In this example:

- A POA, which is PMT-enabled and has the name `account_server_poa`, is created.
- Invocations on the target object with an ID of `account_server_poa`, an operation equal to `withdraw` is performed with the `PMT_Required` policy. It will either join the client-initiated transaction (T1), or, if the client did not initiate one, POA initiates a new global transaction (T2).
- An xa-resource with the name `account_storage` is used by this transaction.

PMT::Current and connection name

OMG OTS defines the `CosTransactions::Current` object for an application to retrieve information and manipulate thread specific client-initiated transactions.

PMT also provides an additional object, `PMT::Current`, for applications to retrieve information about the transaction and connection associated to the thread by POA.

```
class PMT_Current {
public:
    static const PMT_Current* instance();

    const char*    resourceName() const;
    const char*    connectionName() const;

    // XA diagnoses
    const xid_t*   xid() const;
    int            rmid() const;

    // PMT diagnoses (these two method does not raise exception)
    int            attribute() const;
    int            decision() const;
};
```

The name of the connection associated to the current work thread is returned by `connectionName()` of the current object. This name can be used to instruct an embedded SQL statement to use a specified connection, with either the `AT <conn_name>` clause or the `SET CONNECTION <conn_name>` statement, as shown in the following example:

```
void BankImpl::deposit(const char* id, float amount) {
    EXEC SQL BEGIN DECLARE SECTION;
        const char* account_id = id;
        float deposit_amount = amount;
        const char* conn = current->connectionName();
    EXEC SQL END DECLARE SECTION;

    EXEC SQL AT :conn UPDATE account_table
        WHERE account_id = :account_id
        SET balance = balance + :deposit_amount;
}
```

The connection name is similar to the concept of connection handle in Call Level Interface (CLI).

The `AT` clause in the previous example is optional in some cases. For example, Oracle has the concept of *default connection*, which is the connection last opened by the thread-of-control. If an embedded SQL does not have the `AT` clause, Oracle uses the default connection. Other databases, such as Sybase, do not have the concept of default connection, and Borland recommends using the `AT` clause or `SET CONNECTION` statement with those databases.

The following additional `PMT::Current` methods are used for diagnostic purposes.

- `const char* PMT::Current::resourceName() const;`
Returns the XA resource name used by associated XA connection. See [“XA resources configuration”](#) for more information.
- `const xid_t* PMT::Current::xid() const;`
Returns the XID of the transaction associated with the current thread. If no transaction is associated, this method raises a `CosTransactions::unavailable` exception.
- `int rmid() const;`
Returns the XA resource manager ID of the XA connection associated with the

current thread. If no XA connections and no transactions are associated, this method raises a `CosTransactions::Unavailable` exception.

- `int attribute() const;`
Returns the PMT attribute mode of the PMT attribute that matches or wildcard matches with the current {POA, oid, method-name} triplex. If PMT is not enabled on the POA, the return value is 0.
- `int decision() const;`
Returns a value 1 or 2, indicating the current thread is associated with a client or server-initiated transaction. If PMT is not enabled on the POA, the return value is 0.

XA resources configuration

xa-resource-descriptor

In VisiTransact, XA resources are also configured using an XML description named `xa-resource-descriptor`. An `xa-resource-descriptor` is the root element of an `xa-resource-descriptor` XML file, which typically has following structure:

```
<?xml version="1.0"?>
<!DOCTYPE xa-resource-descriptor SYSTEM "xa-resdesc.dtd">

<!-- an example of xa-resource-descriptor -->
<xa-resource-descriptor>
    ...

</xa-resource-descriptor>
```

The `<xa-resource-descriptor>` root element can have one or multiple `<xa-resource>` sub-elements, and follow by zero or multiple `<xa-resource-alias>` elements with the following structure:

```
<!ELEMENT xa-resource-descriptor
    (xa-resource+,
    xa-resource-alias*)
    &gt;
```

xa-resource

An `<xa-resource>` defines and configures an XA resource supplier. Its sub-elements, `<xa-connection>` define connections to be opened on the given `xa-resource`. The DTD of an `<xa-resource>` is:

```
!ELEMENT xa-resource
    (xa-connection+)
>&
ATTLIST xa-resource
    name          CDATA          "default"
    xa-library    CDATA          #IMPLIED
    xa-switch     CDATA          #REQUIRED
    xa-conn-scope (thread|process) #REQUIRED
>
```

An `<xa-resource>` specifies one or more `<xa-connection>` sub-elements. It can be used to configure the following attributes:

- *name*
Specifies a unique name for this `xa-resource`. This name is used by the PMT `<transaction>` element to decide which `xa-resource` the dispatched request should be associated with. The default value is `default`.

- *xa-library*
Specifies the library file name of the XA API library, provided by database vendor. If this attribute is left out, the engine will try to resolve the XA from the application executable module itself.
- *xa-switch*: Specifies the symbol of the `xa_switch_t` variable. For example, for Oracle XA, the symbol is `xaosw`, for Informix, the symbol is `infx_xa_switch`, and for DB2, the symbol is `db2xa_switch`.
- *xa-conn-scope*: Specifies the scope of XA connection provided by the XA library. This depends on the XA API library that is used, and on the used XA open string (the `info` attribute in the `<xa-connection>` element).

Database	xa-library	xa-switch	xa-conn-scope
Oracle 9 and 10	libclntsh.so/sl/a Oraclnt9.dll	Xaosw	"thread", if info contains "+Threads=true" "process" if info contains "+Threads=false"
Informix 7	[lib]infxa.[extension]	infx_xa_switch	"thread"
DB2 8	[lib]db2.[extension]	db2xa_switch	
Direct Connection Driver			"process"

Avoid the `xa-conn-scope="process"` mode for Oracle XA because the Oracle XA library is not thread-safe when `"Threads=false"`.

xa-connection

The `<xa-connection>` element specifies a given XA connection's name and `xa_open` info string. The DTD of `<xa-connection>` element is:

```
<!ELEMENT xa-connection
  EMPTY
>
<!ATTLIST xa-connection
  name CDATA          #IMPLIED
  info CDATA          #REQUIRED
>
```

- *name*
Name of the connection. This name is returned from `PMT::Current::connectName()` method when the connection is associated with the thread. You must ensure that this name is consistent with the name assigned in the `info` string.

XA API	Info substring specifying connection name
Oracle XA	"DB=<name>"
Informix XA	"CON=<name>"
Sybase XA	"N=<name>"
DB2 XA	"DB=<name>"

- *info*
The string passed to `xa_open()`. The information specified by this string is XA provider dependent. The following table shows typical setting templates:

Using the information you provide, the PMT XA engine opens XA open connections. If

XA API	Typical info string template (items inside the [] are optional)
Oracle XA	"Oracle_XA+Acc=P/[<uid>]/ [<pwd>]+SqlNet=<dblink>+SesTm=<timeout>[+Threads=<true/false>] [+LogDir=<dir>][+DbgFlag=<0x0 to 0x7>][+DB=<conn_name>]"
Informix XA	"[DB=<dbname>][;USER=<uid>][;PASSWD=<pwd>][;RM=<server>][;CON=<conn_name>]"
Sybase XA	"-U<uid> -P<pwd> [-L<logfile>] [-T<traceflg>] [-V12] [-O<1 -1>] [-N<lrm>]"
DB2 XA	"[UID=<uid>][,PWD=<pwd>][,TPM<tpm>][,DB=<conn_name>]"

the value of the `xa-conn-scope` attribute of `<xa-resource>` is `process`, `VisiTransact` opens one specified connection and associates it to one thread at a time. If the value of this attribute is `thread`, `VisiTransact` opens a connection per-work-thread when associating the given worker thread to a transaction.

xa-resource-alias

An `<xa-resource-alias>` element defines an alias name to a previously-defined `<xa-resource>` element:

```
<!ELEMENT xa-resource-alias
    EMPTY
>
<!ATTLIST xa-resource-alias
    name          CDATA #REQUIRED
    xa-resource   CDATA #REQUIRED
>
```

- *name*
The `xa-resource`'s alias name
- *xa-resource*
The actual `xa-resource` that this alias points to.

When the name of an `xa-resource-alias` is referred to by a PMT `<transaction>` element's `xa-resource` attribute, the actual `xa-resource` is used.

An example of XA resource descriptor

The following example shows a comprehensive description of an `xa-resource-descriptor`:

```
<?xml version="1.0"?>
<!DOCTYPE xa-resource-descriptor SYSTEM "xaresdesc.dtd">

<!-- an example of xa-resource-descriptor -->
<xa-resource-descriptor>

    <!-- 1. list of xa resources -->
    <xa-resource
        name="oracle"
        xa-switch="xaosw"
        xa-conn-scope="thread"
    </xa-resource>

    <!-- 2. list of xa connections -->
    <xa-connection
        info=
            "Oracle_XA+Acc=P/scott/
            tiger+SesTm=10+SqlNet=ora92a+Threads=true"
    />
```

```

</xa-resource>

<!-- 3. list of resource alias(es) -->
<xa-resource-alias
  name="default"
  xa-resource="oracle"
/>

<xa-resource-alias
  name="account-storage"
  xa-resource="oracle"
/>
</xa-resource-descriptor>

```

In the previous example:

- The `xa-resource-descriptor` contains a single `xa-resource`, with the name `oracle`.
- The `xa-resource` specifies the `xa-switch` symbol `xaosw`, but not the `xa-library` file name. Therefore, VisiTransact resolves the `xa switch` within the current executable module, rather than from an external loaded library. This is a typical usage scenario because it is likely that the application has already linked with the client library of the database, which is likely to contain the needed XA API.
- The `xa-conn-scope` is set to `thread`. This is consistent with the `+Threads=true` substring in the `xa-connection's info` attribute. In this case, VisiTransact opens one dedicated XA connection per worker-thread when associating the thread with a transaction.
- The `xa-connection` element omitted the `name` attribute, as well as the `+DB=<name>` substring in the `info` string. This is a typical usage scenario for an Oracle XA application under thread mode. The embedded SQL assumes the default connection. Applications do not need to use the AT clause.
- An `<xa-resource-alias>` element is defined with the name `default` and points to the `oracle <xa-resource>` defined previously. Whenever a PMT `<transaction>` element is defined with the `<xa-resource>` name `default`, the referenced `oracle xa-resource` is used.
- An additional `<xa-resource-alias>` element is defined with the name `account-storage` and points to the `oracle <xa-resource>` defined previously. Whenever a PMT `<transaction>` element is defined with the `<xa-resource>` name `account-storage`, the referenced `oracle xa-resource` is used.

VisiTransact properties

`vbroker.its.its6xmode=< false|true>`

If set to `false`, all VisiTransact PMT functions and optimizations are enabled. If set to `true`, PMT enhancements and optimizations are disabled, and the following deprecated features are enabled:

- The transactional application uses in-proc OTS.
- The POA is not created with OTS policy, but the object on that POA is inherited from `CosTransactions::TransactionalObject`.
- The application uses `NonTxTargetPolicy` on the client side.
- The application uses the `SessionManager`.
- The VisiTransact OTS server is used by VBJ Java clients and VBJ Servers.

This property is available for performance comparison, bug isolation, and backward compatibility requirements. The default value is `false`.

vbroker.its.verbose=<false|true>

If set to true, VisiTransact prints low level exception and warning runtime information. The default value is `false`.

vbroker.its.xadesc=<xa-resource xml file name>

Specifies the XA-resource configuration file using this property. The default value is `itsxadesc.xml`.

vbroker.orb.enableTransactions=<false|true>

If set to false, it automatically loads up `com.borland.vbroker.CosTransactions.implicit.Init`

RM recovery utility

The two-phase commit mechanism ensures that all nodes either commit or perform a rollback together. During the course of two-phase commit, if a failure occurs because of a network problem, database crash, or an unhandled software error, the transaction becomes in doubt and the resources in the database are locked and are not freed. To solve this problem VisiTransact comes with an RM recovery utility, `rmrecover` (`rmrecover.exe` on Windows), along with automatic TM recovery.

Borland recommends that you execute this utility for each Resource Manager involved in the transaction before restarting a VisiTransact application server that terminated in a failure.

The usage of `rmrecover` is:

```
% rmrecover <xa_resource_desc.xml> [<options>]
```

- `<xa_resource_desc.xml>` is the xa resource configuration used by the RM to connect to the database.
- `<options>` specifies the xa-resource name.

To run the `rmrecover` utility, complete the following steps:

- 1 Modify the user ID and password in `<xa_resource_desc.xml>` to obtain database administrative rights.
- 2 Configure the oracle client library in `<xa_resource_desc.xml>` as appropriate for the operating system:
 - For Windows: `xa-library="oraclient9.dll"`
 - For Unix: `xa-library="libclntsh.so"`
- 1 Start Transaction Service, `ots` (`ots.exe` on Windows) on a specific port `ots - Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>`
- 2 Start the `rmrecover` utility `rmrecover <xa_resource_desc.xml> <xa-resource name> - ORBInitRef VisiTransactionService=corloc::<host>:<port>/VisiTransactionService`

The RM recover utility contacts the database and fetches the list of transactions that are in doubt and either commits or rolls back each transaction.

19

XA Session Manager for Oracle OCI, version 9i Client

This chapter covers issues relating to using the Oracle9i version of the Oracle Call Interface (OCI) database with the XA Session Manager implementation. This chapter contains the following sections:

- Overview
- Oracle9i software requirements
- Oracle9i installation and configuration issues
- Required environment variables
- Session Manager connection profile attributes
- Programming restrictions
- Troubleshooting

Overview

This chapter provides you with information on the specific database issues and requirements for using the Oracle9i version of the Oracle Call Interfaces (OCI) and the Oracle9i database with the VisiTransact using XA transaction coordination. This includes software requirements, installation and configuration information, Session Manager and XA Resource Director configuration attributes, and programming restrictions.

VisiTransact transactional data access occurs through the use of the Session Manager for OCI and the Oracle XA libraries. A connection to the database is established when the application requests a Connection object from the Session Manager. The application can then obtain a native handle, which can be used for making normal OCI calls.

The information covered in this chapter focuses on the specific requirements for accessing an Oracle DBMS with VisiTransact using the standard XA commit protocol. To properly install and configure Oracle, you need to follow the instructions in the documentation shipped with your Oracle database.

Who should read this chapter

System administrators and database administrators responsible for administering this database should read this chapter before installing and configuring the DBMS if it will be used for transaction processing. In particular, refer to the sections identified below. Application developers building applications with VisiTransact should review the information in [“Programming restrictions”](#).

Sections	System administrators	Database administrators	Application developers
Oracle9i Software Requirements	X	X	
Oracle9i Installation and Configuration Issues	X	X	
Required Environment Variables, Troubleshooting	X	X	X
Session Manager Connection Profile Attributes	X	X	
Programming Restrictions		X	X
Troubleshooting	X	X	X

Oracle9i software requirements

You must install the Oracle9i client libraries on every machine on which you will run the XA Resource Director or an application built using the Session Manager. The XA Resource Director and the Session Manager are components of VisiTransact.

The following sections list database client and server requirements by platform.

Client requirements

The following Oracle client components for Oracle OCI must be installed and configured on each node which runs the XA Resource Director or an application built using the Session Manager:

- Oracle OCI, version 9i must be installed on Solaris
- Oracle XA Libraries

Server requirements

The following Oracle server components must be installed and configured on each database server machine:

- Oracle Server, version 9i
- Oracle Distributed Database option

Oracle9i installation and configuration issues

The following sections identify Oracle installation and configuration software issues.

Installation requirements

To install Oracle, you will need the following:

- Oracle installation and configuration guide
- Corresponding release bulletins

Database configuration

Use the **init.ora** parameters, described in the following table, to help configure your database for use with the XA Session Manager for Oracle OCI.

init.ora parameter	Description
transactions	The number of distributed transactions in which the database can concurrently participate.
sessions	The total number of user and system sessions.
processes	See the <i>Oracle9i Server Administrator's Guide</i> for more information on setting this parameter.
distributed_lock_timeout	The amount of time in seconds for distributed transactions to wait for locked resources.

With VisiTransact, the number of distributed transactions is limited by the database **init.ora** parameter `transactions`. Transactions remain active from the time of the first `getConnection()` or `getConnectionWithCoordinator()` call until the commit or rollback is complete. The default setting for `transactions` is generally set too low for use with the Session Manager. The default is system dependent.

With Oracle OCI, each distributed transaction, as opposed to a connection, consumes a database session. Make sure that **init.ora** parameters, `sessions` and `processes`, are set high enough to accommodate the distributed transactions as well as other applications' sessions.

The use of distributed transactions like XA may restrict the use of other Oracle features on some platforms. For instance, the use of the Oracle Parallel Server option may be restricted on some platforms.

Note

See Oracle documentation for information on how to set **init.ora** parameters and for information about the interaction of Oracle XA with other Oracle features, including Oracle Parallel Server and Oracle Replication.

DBA_PENDING_TRANSACTIONS view

The view, `DBA_PENDING_TRANSACTION` is used during recovery processing by the XA Resource Director to synchronize transaction information between the database and the VisiTransact Transaction Service. All users specified as Oracle userids in Oracle9i Session Manager profiles must be granted the `SELECT` privilege on this view.

To make sure that the permissions to the view are correct and that recovery processing can take place, log into Oracle using **SQL*Plus** as the userid for the XA Resource Director and perform the following query:

```
select count (*) from SYS.DBA_PENDING_TRANSACTIONS;
```

If you receive Oracle error "ORA-00942: the table or view does not exist," then the XA Resource Director will not be able to access this view. The user can either logon as user **sys** or **system** or connect internally from the server manager to grant the `select` privilege on this view to the appropriate user.

Required environment variables

The `PATH` environment variable needs to include the path to the Oracle client directory where the database client libraries are installed, as well as the path to the Session Manager libraries.

```
LD_LIBRARY_PATH
PATH
```

Add `$ORACLE_HOME/bin` to your `PATH` and `$ORACLE_HOME/lib32` (or `$ORACLE_HOME/lib` for 64 bit applications) to your `LD_LIBRARY_PATH`. For example, with the Bourne shell:

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${ORACLE_HOME}/lib32
PATH=${ORACLE_HOME}/bin:${PATH}
```

Session Manager connection profile attributes

The following table contains the configuration profile attributes which are specific to the XA Session Manager for Oracle OCI.

Attribute	UI label	Description	Range
heartbeat_retry_period	Heartbeat Retry Period	The number of seconds between heartbeats to the VisiTransact Transaction Service instance after a missed heartbeat. Used to detect the reactivation of an instance of the VisiTransact Transaction Service in order to begin recovery. Used in the Resource Director only.	> 0
heartbeat_watch_period	Heartbeat Watch Period	The number of seconds between heartbeats to the VisiTransact Transaction Service instance. Used to automatically detect VisiTransact Transaction Service instance failures. Used in the Resource Director only.	> 0
oracle_txn_idle_timeout	Transaction Idle Timeout	The timeout (in seconds) that an unprepared transaction can be idle before Oracle rolls back the transaction. When using this attribute, consider the timeout period set for the VisiTransact Transaction Service.	> 0
oracle_xa_logdir	Log Directory Path	Path to the directory where the Oracle XA log files are to be written.	0 to 256 characters long
resource_director_name	Resource Director	The name of the Resource Director to be used.	1 to 128 characters long
native_handle_type	Native Handle Type	The type of the native connection handle requested by the application.	Valid values are Lda_Def and ITSoracle9i_handles

Using the Session Manager with the OCI 9i API

In Oracle9i, the OCI interface has been completely rewritten. With this new interface, several handles are needed in order to executed SQL statements. In order to use this API with the Session Manager, perform the following steps:

- 1 In the connection profile, set the attribute `native_handle_type` to **ITSoracle9i_handles**.
- 2 Include the file, `ora9i_sessmgr.h`, in the application source to define the object **ITSoracle9i_handles**.
- 3 Cast the return value of `Connection::getNativeConnectionHandle()` to the type **ITSoracle9i_handles *** (a pointer to an object of type **ITSoracle9i_handles**).
- 4 Use accessor methods provided by the class **ITSoracle9i_handles** to obtain the various handles needed. These methods are:
 - `OCISvcCtx *getSvcCtx();`
 - `OCIEnv *getEnv();`
 - `OCIError *getError();`

Do not attempt to de-allocate the objects obtained through the **ITSoracle9i_handles** object; these object instances are managed by the Session Manager.

Programming restrictions

The following restrictions apply when programming an application for transaction processing:

- You must use a Connection object in the thread in which it was created.
This is a restriction of the Oracle9i XA implementation that means that you can only use the native connection handle obtained from a Connection object instance in the thread that obtained the object. Using this connection handle on any other threads will cause unexpected results.
- Do not use DDL statements in your application.
This restriction means that DDL SQL statements will not be supported in the Oracle XA application. This is because a DDL SQL statement such as `CREATE TABLE` performs an implicit commit. Any required DDL statements must be performed by a process which does not use the XA protocol.

The following operations, shown in the following table, cannot be used on connections obtained through the Session Manager.

Operations	Disallowed SQL commands	9i API disallowed OCI calls
Connection Operations	CONNECT	OCISvcCtxLogon OCISvcCtxLogoff
Transaction Operations	COMMIT ROLLBACK SAVEPOINT SET TRANSACTION (READ ONLY READWRITE USE ROLLBACK SEGMENT)	OCITransCommit OCITransRollback OCIStmtExecute in OCI_COMMIT_ON_ SUCCESS mode
Implicit Operations	DDL SQL statements (for example, CREATE TABLE, CREATE INDEX)	

Troubleshooting

This section identifies problems that could occur when using the XA Session Manager for Oracle OCI with the Oracle database and provides you with suggestions for troubleshooting the problem.

VisiTransact message log

The VisiTransact message log may contain Session Manager and native Oracle error messages when a connection or transaction error happens.

Using the xa_trc files

If errors occur that indicate problems with the XA code, more information on any Oracle errors can be found in the **xa_*.trc** files. These files will be placed in the log directory specified in the defined connection profile. If a log directory is not specified in the Session Manager connection profile, the **xa_*.trc** files will be placed in the **\$ORACLE_HOME/rdbms/log** directory if **\$ORACLE_HOME** is available, or in the current directory if **\$ORACLE_HOME** is not available, when the process is started.

Note

If a directory is specified but does not exist, there won't be log file and you will not receive any warning.

Distributed update problems

A network or system failure can cause the following types of problems:

- A prepare or commit being processed may not be completed at all nodes of the session when a failure occurs.
- If a failure persists (for example, if the network is down for an extended period), the data exclusively locked by in-doubt transactions (prepared, but not committed or rolled back) is not available for statements of other transactions.

Note

See Oracle documentation for more information on the behavior of distributed updates where one Oracle node serves as a subcoordinator for another Oracle database.

Data access failures

When a user issues a SQL statement, Oracle9i attempts to lock the required data to successfully execute the statement. However, if the requested data is being handled by statements of other uncommitted transactions and continues to remain locked for long periods of time, a timeout occurs.

Lock from in-doubt transaction

A query or DML statement that requires locks on a local database may be blocked indefinitely due to the locked resources of an in-doubt distributed transaction. In this case, the following error message is returned to the user:

```
ORA-01591: lock held by in-doubt distributed transaction <IDt>
```

In this case, the SQL statement is rolled back immediately. The rollback of the SQL statement does not automatically force a rollback of the transaction. The application that executed the statement can try to re-execute the statement later. If the lock persists, the user should contact an Administrator to report the problem, including the ID of the in-doubt distributed transaction.

An in-doubt transaction is a transaction in the prepared state that has not been committed or rolled back.

Transaction timeout

A DML statement that requires locks on a remote database may be blocked if another transaction currently has locks on the requested data. If these locks continue to block the requesting SQL statement, a timeout occurs, the statement is rolled back and the following error message is returned to the user.

```
ORA-02049: timeout: distributed transaction waiting for lock
```

In this case, the SQL statement is rolled back immediately. The rollback of the SQL statement does not automatically force a rollback of the transaction. The application should proceed as if a deadlock has been encountered. The application that executed the statement can try to re-execute the statement at a later time. If the lock persists, the user should contact an administrator to report the problem.

The timeout interval described in the above situation can be controlled with the initialization parameter `distributed_lock_timeout`. This interval is in seconds. For example, to set the timeout interval for an instance to 30 seconds, include the following line in the associated parameter file:

```
DISTRIBUTED_LOCK_TIMEOUT=30
```

With the above timeout interval, the timeout errors discussed in the previous section occur if a transaction cannot proceed after 30 seconds of waiting for unavailable resources.

See [“Database configuration”](#) for a description of the `distributed_lock_timeout` parameter.

Oracle error messages

The VisiTransact Message Log contains Oracle error messages which could help you troubleshoot connections and transaction errors including the following:

Error message	Description	Solution
ORA-12154	The process limit for file descriptors (<code>ulimit</code>) on Solaris is set too low for multithreaded applications.	<p>Check profile name for correct database name.</p> <p>Check the <code>tnsnames.ora</code> file for matching service names entry.</p> <p>Check that the database and Oracle listener process are up.</p> <p>Check that you have set your File Descriptor limit (<code>ulimit</code>), on Solaris, high enough to assure that you can open connections.</p> <p>See the Solaris Operating System documentation for information on setting the <code>ulimit</code> command.</p>

Forcing heuristic completion

Use `COMMIT FORCE <local transaction id>` or `ROLLBACK FORCE <local transaction id>`—where `<local transaction id>` comes from the `dba_2pc_pending` table—to force completion of a heuristic transaction. Refer to *Oracle9i Distributed Database Systems* documentation for more information.

20

DirectConnect Session Manager for Oracle OCI, version 9i Client

This chapter covers issues relating to using the Oracle9i version of the Oracle Call Interface (OCI) database with the DirectConnect Session Manager implementation. This chapter contains the following sections:

- Overview
- Oracle9i software requirements
- Oracle9i installation and configuration issues
- Required environment variables
- Session Manager connection profile attributes
- Programming restrictions
- Troubleshooting

Overview

This chapter provides you with information on the specific database issues and requirements for using the Oracle9i version of the Oracle Call Interface (OCI) and the Oracle9i database with the DirectConnect Session Manager implementation. This includes software requirements, installation and configuration information, Session Manager configuration attributes, and programming restrictions. This is in contrast to other DirectConnect Session Manager implementations described in this manual.

VisiBroker VisiTransact transactional data access occurs through the use of the Session Manager for OCI and the Oracle libraries. A connection to the database is established when the application requests a Connection object from the Session Manager. The application can then obtain a native handle, which can be used for making normal OCI calls.

The information covered in this chapter focuses on the specific requirements for accessing an Oracle DBMS with VisiBroker VisiTransact. To properly install and configure Oracle, you need to follow the instructions in the documentation shipped with your Oracle database.

For more information about the DirectConnect Session Manager implementation, see [“Session Manager overview”](#) and [“Data access using the Session Manager.”](#)

Who should read this chapter

System administrators and database administrators responsible for administering this database should read this chapter before installing and configuring the DBMS if it will be used for transaction processing. In particular, refer to the sections identified below. Application developers building applications with VisiTransact should review the information in [Programming restrictions](#).

Sections	System administrators	Database administrators	Application developers
Oracle9i Software Requirements	X	X	
Oracle9i Installation and Configuration Issues	X	X	
Required Environment Variables, Troubleshooting	X	X	X
Session Manager Connection Profile Attributes	X	X	
Programming Restrictions		X	X

Oracle9i software requirements

You must install the Oracle9i client libraries on every machine on which you will run an application built using the Session Manager. The Session Manager is a component of VisiBroker VisiTransact.

The following sections list database client and server requirements by platform.

Client requirements

The following Oracle client components for Oracle OCI must be installed and configured on each node which runs an application built using the Session Manager:

- Oracle OCI, version 9i must be installed on Solaris

Server requirements

UNIX

The following Oracle server component must be installed and configured on each database server machine:

- Oracle9i Server

Oracle9i installation and configuration issues

The following sections identify Oracle installation and configuration software issues.

Installation requirements

To install Oracle, you will need the following:

- Oracle installation and configuration guide
- Corresponding release bulletins

Database configuration

Use the **init.ora** parameters, described in the following table, to help configure your database for use with the DirectConnect Session Manager for Oracle OCI.

init.ora parameter	Description
sessions	The total number of user and system sessions.
processes	See the <i>Oracle9i Server Administrator's Guide</i> for more information on setting this parameter.

With the DirectConnect Session Manager for Oracle OCI, each transaction consumes a database session. Make sure that **init.ora** parameters, `sessions` and `processes`, are set high enough to accommodate the transactions as well as other application sessions.

Each connection opened by the DirectConnect Session Manager will require a session until the transaction completes. Therefore, the `sessions` parameter should be set to a number higher than the maximum number of concurrent DirectConnect transactions that you expect will access the database.

Note

See the Oracle documentation for information on how to set **init.ora** parameters.

Required environment variables

The `PATH` environment variable needs to include the path to the Oracle client directory where the database client libraries are installed, as well as the path to the Session Manager libraries.

UNIX:

```
LD_LIBRARY_PATH
PATH
```

Add `ORACLE_HOME/bin` to your `PATH` and `$ORACLE_HOME/lib32` (or `$ORACLE_HOME/lib` for 64 bit applications) to your `LD_LIBRARY_PATH`. For example, with the Bourne shell:

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${ORACLE_HOME}/lib32
PATH=${ORACLE_HOME}/bin:${PATH}
```

Session Manager connection profile attributes

The following table contains the configuration profile attributes which are specific to the XA Session Manager for Oracle OCI.

The following table contains the attributes for the XA Session Manager for Oracle OCI.

Attribute	UI label	Description	Range
native_handle_type	Native Handle Type	The type of the native connection handle requested by the application.	Valid values are <code>Lda_Def</code> and <code>ITSOracle9i_handles</code> .

Using the Session Manager with the OCI 9i API

In Oracle9i, the OCI interface has been completely rewritten. With this new interface, several handles are needed in order to executed SQL statements. In order to use this API with the Session Manager, perform the following steps:

- 1 In the connection profile, set the attribute `native_handle_type` to `ITSOracle9i_handles`.

- 2 Include the file, `ora9i_sessmgr.h`, in the application source to define the object `ITSOracle9i_handles`.
- 3 Cast the return value of `Connection::getNativeConnectionHandle()` to the type `ITSOracle9i_handles *` (a pointer to an object of type `ITSOracle9i_handles`).
- 4 Use accessor methods provided by the class `ITSOracle9i_handles` to obtain the various handles needed. These methods are:
 - `OCISvcCtx *getSvcCtx();`
 - `OCIEnv *getEnv();`
 - `OCIError *getError();`

Do not attempt to deallocate the objects obtained through the `ITSOracle9i_handles` object; these object instances are managed by the Session Manager.

Programming restrictions

The following restrictions apply when programming an application for transaction processing with VisiBroker VisiTransact and Oracle OCI.

Since the VisiTransact Transaction Service and the Session Manager control connection and transaction management, platforms should not use the disallowed operations shown in the following table:

Operations	Disallowed SQL commands	9i API disallowed OCI calls
Connection Operations	CONNECT	OCISvcCtxLogon OCISvcCtxLogoff
Transaction Operations	COMMIT ROLLBACK SAVEPOINT SET TRANSACTION	OCITransCommit OCITransRollback OCIStmtExecute in <code>OCI_COMMIT_ON_SUCCESS</code> mode
Implicit Operations	DDL SQL Statements (for example, CREATE TABLE)	

Troubleshooting

This section identifies problems that could occur when using the DirectConnect Session Manager for Oracle OCI with the Oracle database and provides you with suggestions for troubleshooting the problem.

VisiBroker VisiTransact message log

The VisiTransact message log may contain Session Manager and native Oracle error messages when a connection or transaction error happens.

Oracle error messages

The VisiTransact message log and the `VISessionManager::Error` exceptions also contain Oracle error messages which could help you troubleshoot connections and transaction errors including the following:

Error message	Description	Solution
ORA-01017	Invalid username/ password	Check connection profile for correct user name and password.
ORA-12154	Cannot resolve service name	<p>Check profile name for correct database name.</p> <p>Check the tnsnames.ora file for matching service names entry.</p> <p>Check that the database and Oracle listener process are up.</p> <p>Check that you have set your File Descriptor limit (ulimit), on Solaris, high enough to assure that you can open connections.</p> <p>See the Solaris Operating System documentation for information on setting the ulimit command.</p>

21

Commands, utilities, arguments, and environment variables

This appendix provides information about arguments for VisiTransact commands and `ORB_init()` and environment variables used with VisiTransact.

Overview of VisiTransact commands

The commands in the next few sections relate to one another as described in the following table.

VisiTransact component	Related commands
VisiBroker Console	<code>"vbconsole"</code> . This command starts the VisiBroker Console.
VisiTransact Transaction Service	<code>"ots"</code> . This command starts an instance of the VisiTransact Transaction Service. <code>"vshutdown"</code> . This command shuts down an instance of the VisiTransact Transaction Service.
Session Manager	<code>"xa_resdir"</code> . This command starts an instance of the XA Resource Director, part of the Session Manager. <code>"smconfig_server"</code> . This command starts an instance of the Session Manager Configuration Server. <code>"vshutdown"</code> . This command shuts down an instance of the XA Resource Director or Session Manager Configuration Server.
Session Manager Configuration Setup	<code>"smconfigsetup"</code> . This utility creates the connection profile for use with the Pluggable Resource Interface for creating a customized Session Manager.

vbconsole

This command invokes the VisiBroker Console, and can be run on any node that has the executable for the VisiBroker Console installed. The VisiBroker Console does not have to be local to the VisiTransact Transaction Service or Session Manager Configuration Server instances that it administers. However, to administer these instances, they must be running when the VisiBroker Console is started.

Syntax

```
prompt>vbconsole
```

Example

```
prompt>vbconsole
```

Arguments

None.

ots

This command starts an instance of the VisiTransact Transaction Service.

Syntax

```
prompt>ots [-Dvbroker.ots.defaultTimeout=<seconds>]
           [-Dvbroker.ots.defaultMaxTimeout=<seconds>]
           [-Dvbroker.ots.name=<transaction_service_name>]
           [-Dvbroker.ots.logDir=<directory_name>]
           [-Dvbroker.log.enable=<Boolean>]
           [vbroker.ots.logPurgeTransactions=<true|false>]
           [vbroker.ots.logSleep=<milliseconds>]
           [vbroker.ots.logCache=<cache_size_in_kilobytes>]
           [vbroker.ots.logUnit=<transaction_log_size>]
```

Example

```
prompt>ots -Dvbroker.ots.defaultTimeout=60 -Dvbroker.ots.defaultMaxTimeout=120
-Dvbroker.ots.name=Sales -Dvbroker.log.enable=true
```

Arguments

The following arguments can be used with this command.

Argument	Description
-Dvbroker.ots.defaultTimeout=<seconds>	Sets the default transaction timeout value for this VisiTransact Transaction Service instance. If not set, this defaults to 600 seconds.
-Dvbroker.ots.defaultMaxTimeout=<seconds>	Sets the maximum allowed transaction timeout value for this VisiTransact Transaction Service instance. If not set, this defaults to 3600 seconds.
-Dvbroker.ots.name=<transaction_service_name>	Sets the name of the VisiTransact Transaction Service instance used when registering its interface with the Smart Agent. The default is <host_name>_ots.

Argument	Description
<code>-Dvbroker.ots.logDir=<directory_name></code>	Names the directory in which logs and logger information are kept. If you do not specify, the default is <code><VBROKER_ADM>\its\<transaction_service_name>\logger</code> .
<code>-Dvbroker.log.enable=<Boolean></code>	To see the debug log statements from this service, set this property to <code>true</code> . For the various source names options for debug log filtering, see the Debug Logging properties section of the <i>VisiBroker for C++ Developer's Guide</i> .
<code>vbroker.ots.logPurgeTransactions=<true false></code>	Indicates whether transaction logs are new files.
<code>vbroker.ots.logSleep=<milliseconds></code>	Indicates the number of milliseconds of sleep time before checking if the cache is full and needs to flush into physical file. The default is 0.
<code>vbroker.ots.logCache=<cache_size_in_kilobytes></code>	Indicates the size of cache before flushing into the physical file. The default is 64k.
<code>vbroker.ots.logUnit=<transaction_log_size></code>	Indicates the size of the log file. The default is 8M.

smconfig_server

This command is used to start an instance of the Session Manager Configuration Server. You use the Session Manager Configuration Server as the agent to create a connection profile that accesses your database.

Syntax

```
prompt>smconfig_server [-Dvbroker.sm.pstorePath=<path>]
                        [-Dvbroker.sm.configName=<name>] [-m{32|64}]
```

Example

```
prompt>smconfig_server -Dvbroker.sm.pstorePath=C:\vbroker\adm\its\
session_manager
-Dvbroker.sm.configName=athena_smcs -m64
```

Arguments

The following arguments can be used with this command.

Argument	Description
<code>-Dvbroker.sm.pstorePath=<path></code>	Provide the path to the directory where the persistent store files are located. By default, the persistent store files are located in <code><VBROKER_ADM>\its\session_manager</code> .
<code>-Dvbroker.sm.configName=<name></code>	Provide the name of the Session Manager Configuration Server you're using. By default, the name assigned to the Session Manager Configuration Server is <code><host>_smcs</code> where <code>host</code> is the name of the host on which you created the Session Manager profile.
<code>-m{32 64}</code>	Generates 32-bit or 64-bit shared plug-in libraries name into the profiles. <ul style="list-style-type: none"> ■ -m32 used for 32-bit naming ■ -m64 used for 64-bit naming

vshutdown

This command can be used to shutdown the VisiTransact Transaction Service, XA Resource Director, and the Session Manager Configuration Server.

If it is used to shutdown an instance of the VisiTransact Transaction Service, it defaults to allow the VisiTransact Transaction Service to wait for outstanding transactions to complete before shutting down, but will not accept any new transactions. You can direct the instance of the VisiTransact Transaction Service to shutdown without resolving transactions by using the optional `-immediate` argument.

Note

You can use this command to shutdown an instance of the VisiTransact Transaction Service that is embedded within an application process provided the `-OTSexit_on_shutdown` argument was passed to the application's `ORB_init()` method. For information about shutting down an instance of the VisiTransact Transaction Service that is embedded in an application process, see [“Arguments for applications with an embedded VisiTransact Transaction Service instance”](#).

Syntax

```
prompt>vshutdown -help
prompt>vshutdown -type <object_type>
                [-name <object_name>]
                [-host <host_name>]
                [-immediate]
                [-noprompt]
```

Example

```
prompt>vshutdown -type ots -name myTxnSvc
```

Arguments

The following arguments can be used with this command.

Argument	Description
-help	Use this argument to display the usage information for this command. If you use this argument, vshutdown ignores all other arguments and just gives you usage information.
-type	Valid types are: <ul style="list-style-type: none"> ■ <code>ots</code> for VisiTransact Transaction Service ■ <code>rd</code> for XA Resource Director ■ <code>smcs</code> for Session Manager Configuration Server If you specify only the type, vshutdown lists all the services of that particular type and prompts you whether to shut them down or not.
-name <object_name>	The name of the object to be shutdown. By default vshutdown looks up all the objects of the specified type and prompts you whether to shut them down or not.
-host <host_name>	The host machine where the service resides that you wish to shutdown. By default vshutdown locates all the objects of a particular type and name (if mentioned) on the network and prompts you whether to shut them down or not.
-immediate	Use this argument to direct the instance of the VisiTransact Transaction Service to shutdown immediately without resolving any outstanding transactions.
-noprompt	Use this argument if you do not want to be prompted for confirmation when you get a list of all object types, names, or hosts to be shut down.

xa_resdir

This command is used to start an instance of the XA Resource Director. You must have already created a connection profile that accesses your database using the VisiBroker Console.

Syntax

```
prompt>xa-resdir -Dvbroker.sm.profileName=<profile>
                [-Dvbroker.sm.pstorePath=<path>]
                [-Dvbroker.sm.configName=<name>]
                [-Dvbroker.sm.connectionIdleTimeout=<seconds>]
```

Example

```
prompt>xa-resdir -Dvbroker.sm.profileName=quickstart
-Dvbroker.sm.pstorePath=C:\vbroker\adm\its\session_manager
-Dvbroker.sm.configName=athena_smcs
```

Arguments

The following arguments can be used with this command.

Argument	Description
-Dvbroker.sm.profileName=<profile>	Provide the name of the Session Manager connection profile you want to use to establish a connection with the database. This is required.
-Dvbroker.sm.pstorePath=<path>	Provide the path to the directory where the persistent store files are located. By default, the persistent store files are located in <VBROKER_ADM>\its\session_manager .
-Dvbroker.sm.configName=<name>	Provide the name of the Session Manager Configuration Server you're using. By default, the name assigned to the Session Manager Configuration Server is <host>_smcs where host is the name of the server on which you created the Session Manager profile.
-Dvbroker.sm.connectionIdleTimeout=<seconds>	Provide the number of seconds a connection may stay idle and unassociated with a transaction before being closed automatically by the Session Manager ConnectionPool. This is used to shrink the number of connections in the pool when they are unused. This parameter defaults to 300 seconds.

VisiTransact utilities

smconfigsetup

The smconfigsetup utility allows you to create connection profiles. The following procedure is provided as a guide when using it to create profiles for use with the Session Manager. The bold type indicates entries from the user. The profile is created upon exit from the smconfigsetup utility.

Creating a profile for use with the Session Manager

To create profile for use with the Session Manager:

- 1 Enter **smconfigsetup** at the command prompt.

```
prompt>smconfigsetup
```

2 Enter the number 1 (one) to create a profile.

```
Do you wish to
(0) Quit
(1) Add a profile
(2) List all profiles
(3) List attributes of a profile
(4) Copy a profile
(5) Delete a profile
(6) Create metadata files
(7) Add pluggable data resources
```

Enter the number of your selection: **1**

3 Enter the number corresponding to the database type.

```
2 known Session Manager implementations:
(0) Oracle OCI 9i DirectConnect
(1) Oracle OCI 9i XA
```

Please enter the database type you are trying to create: **0**

4 Enter a connection profile name.

Please enter the name for the new profile: **quickstart**

5 Enter a database name.

```
Attribute name "database_name"
New value for attribute Database Name (default value ) : its09idb
```

6 Enter the user name.

```
Attribute name "userid"
New value for attribute User Name (default value ) : scott
```

7 Enter the user's password.

```
Attribute name "password"
New value for attribute Password (default value ) : tiger
```

8 Enter the Native Handle Type.

```
Attribute name "native_handle_type"
New value for attribute Native Handle Type
(default value <ITSoracle9i_handles>) : ITSoracle9i_handle
```

9 Enter 0 (zero) to exit the utility.

```
Do you wish to
(0) Quit
(1) Add a profile
(2) List all profiles
(3) List attributes of a profile
(4) Copy a profile
(5) Delete a profile
(6) Create metadata files
(7) Add pluggable data resources
```

Enter the number of your selection: **0**

Bye!

The profile is created upon exit from the smconfigsetup utility.

Command-line arguments for applications

You can pass arguments to `ORB_init()` which affect the VisiTransact Transaction Service and your application components. The following sections explain these options.

Passing command-line arguments to `ORB_init()` using `argc` and `argv`

As a component of VisiBroker, command line arguments are passed to VisiTransact components through the VisiBroker ORB initialization call `ORB_init()`. Therefore, in order for arguments specified on the command line to have effect on the VisiTransact operation in a given application process, applications must pass the original `argc` and `argv` arguments to `ORB_init()` from the main program. For example,

```
int main(int argc, char * const* argv)
{
    try
    {
        // initialize the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    }
    ...
}
```

The `ORB_init()` function will parse both ORB arguments and VisiTransact arguments, removing them from the `argv` vector before returning.

Arguments for applications that originate transactions

By default, the first time you start a transaction with `Current::begin()`, an instance of the VisiTransact Transaction Service is found using the Smart Agent. You can specify an instance of the VisiTransact Transaction Service to use, and the timeout value for transactions, by using the arguments described in this section.

You pass these arguments at the command line when starting your transactional server manually. Your application handles these command-line input arguments using the `ORB_init()` method as described in [“Passing command-line arguments to `ORB_init\(\)` using `argc` and `argv`”](#).

The following table explains the arguments that can be passed to `ORB_init()` from the command line for applications that originate transactions.

Argument to <code>ORB_init()</code>	Description
<code>-Dvbroker.ots.currentFactory</code>	VisiTransact uses the specified IOR for the requested Transaction Service (<code>CosTransactions::TransactionFactory</code>) to locate the desired instance of the VisiTransact Transaction Service on the network. This argument enables VisiTransact to operate without the use of a Smart Agent (<code>osagent</code>).
<code>-Dvbroker.ots.currentHost</code>	The Smart Agent will find any available VisiTransact Transaction Service instance that is located on the specified host.
<code>-Dvbroker.ots.currentName</code>	The Smart Agent will find the named VisiTransact Transaction Service instance anywhere on the network.
<code>-Dvbroker.ots.currentTimeout</code>	Sets the transaction timeout value for <code>Current</code> . If the transaction is still alive after the timeout expires, the transaction is automatically rolled back.
<code>vbroker.orb.enableTransactions</code>	When set to false, it automatically loads up <code>com.borland.vbroker.CosTransactions.implicit.Init</code>

For example, to start the **billing** C++ transactional server that uses the **Accounting** VisiTransact Transaction Service, you would use the following command:

```
prompt>billing -Dvbroker.ots.currentName=Accounting
```

To start the **Billing** transactional server that uses the **Accounting** VisiTransact Transaction Service, and has a timeout period of 2400 seconds, you would use this command:

```
prompt>billing -Dvbroker.ots.currentName=Accounting -
Dvbroker.ots.currentTimeout=2400
```

If you specify a combination of `-Dvbroker.ots.currentHost` and `-Dvbroker.ots.currentName`, the Smart Agent will find the named VisiTransact Transaction Service instance on the named host.

If you specify the `-Dvbroker.ots.currentFactory` with either the `Dvbroker.ots.currentHost` or `-Dvbroker.ots.currentName`, the Smart Agent will find the VisiTransact Transaction Service instance by IOR only. It ignores all the other arguments.

Arguments for applications with an embedded VisiTransact Transaction Service instance

You can specify an instance of the VisiTransact Transaction Service to use with the arguments described in this section. You can also specify whether your application process will be brought down when the embedded instance of the VisiTransact Transaction Service terminates.

You pass these arguments at the command line when starting your transactional server manually. Your application handles these command-line input arguments using the `ORB_init()` method as described in [“Passing command-line arguments to ORB_init\(\) using argc and argv”](#).

The following table explains the arguments that can be passed to `ORB_init()` from the command line for applications that embed an instance of the VisiTransact Transaction Service.

Argument to ORB_init()	Description
<code>-Dvbroker.ots.defaultTimeout=<seconds></code>	Sets the default transaction timeout value for this VisiTransact Transaction Service instance. If not set, this defaults to 600 seconds.
<code>-Dvbroker.ots.defaultMaxTimeout=<seconds></code>	Sets the maximum allowed transaction timeout value for this VisiTransact Transaction Service instance. If not set, this defaults to 3600 seconds.
<code>-Dvbroker.ots.name=<transaction_service_name></code>	Sets the name of the VisiTransact Transaction Service instance used when registering its interface with the Smart Agent. The default is <host_name>_ots .
<code>-Dvbroker.ots.logDir=<directory_name></code>	Names the directory in which logs and logger information are kept. If you do not specify, the default is <code><VBROKER_ADM>\its\ <transaction_service_name>\logger</code> .
<code>-Dvbroker.ots.exitOnShutdown</code>	If set to true, this will terminate your in-process instance of the VisiTransact Transaction Service and bring down your application process as well when the VisiTransact Transaction Service is shut down remotely using vshutdown or the VisiBroker Console. If it is either not set or set to false, this will deactivate the VisiTransact Transaction Service objects registered with the Smart Agent but will NOT bring down your application process.

Arguments for applications that use the Session Manager

By default, the Session Manager Configuration Server for the machine where the Session Manager connection profile was created is used—**<host>_smcs**. The persistent store classes are located in **<VBROKER_ADM>\its\session_manager** by default.

You pass these arguments at the command line when starting your transactional server manually. Your application handles these command-line input arguments using the `ORB_init()` method as described in [“Passing command-line arguments to ORB_init\(\) using argc and argv”](#).

The following table explains the arguments that can be passed to `ORB_init()` from the command line for applications that use the Session Manager.

Argument to ORB_init()	Description
<code>-Dvbroker.sm.configName</code>	The name of the Session Manager Configuration Server (profileset). By default, this value is <host>_smcs where host is the name of the server on which you created the Session Manager profile.
<code>-Dvbroker.sm.pstorePath</code>	The path to the directory where the persistent store classes are located. By default, this is <VBROKER_ADM>\its\session_manager .
<code>-Dvbroker.sm.connectionIdleTimeout</code>	Provide the number of seconds a connection may stay idle and unassociated with a transaction before being closed automatically by the Session Manager ConnectionPool. This is used to shrink the number of connections in the pool when they are unused. This parameter defaults to 300 seconds.

Environment variables

The following environment variable can be set for VisiBroker VisiTransact.

Environment variable	Description
<code>VBROKER_ADM</code>	Defines the path to the directory where ITS-specific files are stored.

22

Error codes

This appendix provides information about error codes for VisiTransact.

VisiTransact common error codes

The following table lists common error codes for VisiTransact.

Table 22.1 Common error codes for VisiTransact

Error code	Description	Possible causes	Solutions
201	Permission to access the file or directory is denied.	The process does not have the necessary permissions for accessing the file or directory.	Change the file or directory permissions to allow the process to access it.
202	The process cannot open the requested file.	The file is in the wrong directory. The process does not have permission to access the file.	Verify that the file is in the correct directory, and try again. Change the file permissions to allow the process to access it.
203	An error occurred while reading the file.	The process does not have permission to read the file.	Change the file permissions to allow the process to read the file.
204	An error occurred while writing to the file.	The process has read-only permission—it does not have permission to write to the file. The storage is full, and the system does not have enough room to write the changes to the file.	Change the file permissions to allow the process to write to the file. Clean up the storage, and then try again.
801	An error occurred while attempting to list an object of the given type.	No Location Service is available. No Smart Agent is running. The process is experiencing a communication problem.	Verify that a Location Service is available. See the VisiBroker ORB documentation for details. Start the Smart Agent using the osagent command. See the VisiBroker ORB documentation for details. Verify that all required processes are running, and all machines are up, and try again.

VisiTransact Transaction Service error codes

The following table lists the error codes for the VisiTransact Transaction Service.

Error code	Description	Possible causes	Solutions
4000	An instance of the VisiTransact Transaction Service was started successfully.	This is an informational message only.	This message requires no action.
4001	An instance of the VisiTransact Transaction Service is shutting down by request.	An administrator or other individual has shutdown the instance of the VisiTransact Transaction Service using either the vshutdown command, <i>Ctrl+C</i> , or the kill command.	This message requires no action.
4002	The instance of the VisiTransact Transaction Service is prepared to shutdown, but is waiting for outstanding transactions to enter the completion stage before exiting.	The request to shutdown the instance of the VisiTransact Transaction Service was issued without the <code>-immediate</code> argument, allowing the instance to let outstanding transactions enter the completion stage before exiting.	This message requires no action. If you want to shutdown an instance of the VisiTransact Transaction Service without allowing outstanding transactions to enter the completion stage, use the <code>-immediate</code> argument when issuing the vshutdown command.
4003	The instance of the VisiTransact Transaction Service is shutting down without waiting for outstanding transactions to enter the completion stage.	The request to shutdown the instance of the VisiTransact Transaction Service was issued with the <code>-immediate</code> argument, allowing the instance to shutdown without letting outstanding transactions enter the completion stage.	This message requires no action. If you want to shutdown an instance of the VisiTransact Transaction Service and allow outstanding transactions to enter the completion stage, issue the vshutdown command without the <code>-immediate</code> argument.
4004	A HeuristicHazard exception was raised by a Resource. For complete details on this exception, see the output of the heuristic log file.	A Resource made a heuristic decision and does not know the outcome of at least one relevant update.	There is a possible loss of data integrity. Look up the error in the heuristic log, and notify your database administrator of the transaction identifier. Your database administrator will need to locate this error on the Resource and correct any problems manually.
4005	A HeuristicCommit exception was raised by a Resource. For complete details on this exception, see the output of the heuristic log file.	A Resource made a heuristic decision to commit all relevant updates.	There is a possible loss of data integrity. Look up the error in the heuristic log, and notify your database administrator of the transaction identifier. Your database administrator will need to locate this error on the Resource and correct any problems manually.
4006	A HeuristicRollback exception was raised by a Resource. For complete details on this exception, see the output of the heuristic log file.	A Resource made a heuristic decision to rollback all relevant updates.	There is a possible loss of data integrity. Look up the error in the heuristic log, and notify your database administrator of the transaction identifier. Your database administrator will need to locate this error on the Resource and correct any problems manually.
4007	A HeuristicMixed exception was raised by a Resource. For complete details on this exception, see the output of the heuristic log file.	A Resource made a heuristic decision which differs from the outcome of the transaction. Some updates have been committed, others have been rolled back.	There is a possible loss of data integrity. Look up the error in the heuristic log, and notify your database administrator of the transaction identifier. Your database administrator will need to locate this error on the Resource and correct any problems manually.

Error code	Description	Possible causes	Solutions
4008	An exception was caught and ignored during the callback for a specific alarm (listed in the message).	This message could be thrown for various reasons, including running out of system resources.	Ignore this message.
4009	An internal application error occurred.	An internal module of VisiTransact Transaction Manager, used by several VisiTransact components, could not be initialized due to an unknown exception.	Contact VisiBroker Technical Support.
4010	An internal application error occurred, as described in the message.	An internal module of VisiTransact Transaction Manager, used by several VisiTransact components, could not be initialized due to the exception listed in the message.	Contact VisiBroker Technical Support.
4011	An exception occurred while parsing the initialization arguments listed in the message.	The wrong command-line arguments were entered when executing a VisiTransact command.	Verify the command-line arguments, and try again. See “Commands, utilities, arguments, and environment variables.”
4012	An exception occurred while parsing some initialization arguments, but it is unknown which arguments were incorrect.	The wrong command-line arguments were entered when executing a VisiTransact command.	Verify the command-line arguments, and try again. See “Commands, utilities, arguments, and environment variables.”
4014	An initialization failure (specified in the message) occurred while starting an instance of the VisiTransact Transaction Service.	The wrong configuration file was used, or incorrect values were entered for initialization parameters. An internal application error has occurred.	Verify you are using the correct configuration file, and are entering correct values for the initialization parameters. Contact VisiBroker Technical Support.
4015	A runtime exception occurred within a running instance of the VisiTransact Transaction Service.	An internal application error has occurred.	Contact VisiBroker Technical Support.
4016	The default transaction timeout has been changed to the value of the maximum transaction timeout.	The value of the default transaction timeout was higher than the value of the maximum transaction timeout.	Verify that you have coordinated your timeout settings between your applications and any command-line arguments you are using when starting instances of the VisiTransact Transaction Service.
4017	An invalid value was provided for the default transaction timeout. The timeout value was reset to 600 seconds.	When setting the default timeout value, a zero or a negative value was provided. The default timeout value must be at least 1 second.	When setting the default timeout value, make sure to set it to a value greater than or equal to 1 second. The recommended value is 600 seconds.
4018	An unexpected exception was received by the VisiTransact Transaction Service during transaction completion. The VisiTransact Transaction Service will retry transaction completion.	An internal application error has occurred.	Contact VisiBroker Technical Support.
4019	An unexpected CORBA exception was received by the VisiTransact Transaction Service during transaction completion. The VisiTransact Transaction Service will retry transaction completion.	An internal application error has occurred.	Contact VisiBroker Technical Support.

Session Manager error codes

The table lists the error codes for the Session Manager.

Error code	Description	Possible causes	Solutions
6001	The Session Manager could not allocate the necessary memory.	Your system may have run out of memory.	Make memory available by increasing swap spaces and shutting down unnecessary processes.
6002	Profiles are incompatible because they are using different implementations of the Session Manager.	An application is attempting to use one profile which specifies the XA Session Manager for OCI in the same process with a profile which specifies the DirectConnect Session Manager for Oracle OCI.	Ensure that only one type of Oracle connection is used with the Session Manager in one process.
6003	The Session Manager could not load a necessary library.	The PATH or LIBRARY_PATH environment variables may not be set correctly, or the associated libraries may be missing.	Be sure to set environment variables correctly. If this does not correct the problem, you may need to reinstall VisiBroker VisiTransact and the VisiBroker ORB.
6004	The Session Manager could not locate a needed function symbol in the loaded library.	Not all required database libraries could be found. The loaded library file may be corrupted.	Verify that the PATH (Windows NT) and the LD_LIBRARY_PATH (Solaris) environment variables include the path to your database libraries. Check that the database client libraries are available. Reinstall the VisiBroker ORB and VisiBroker VisiTransact software.
6005	The Session Manager could not open a connection to the database.	The database may be unavailable. You may be using an invalid user name or password. The database software may not be correctly installed or configured.	Verify that the database is up and running. Check the Session Manager connection profile to verify that the database name, user name, and password are entered correctly. Verify that your database software is installed and configured correctly.
6006	An error occurred during the initialization phase of an attempt by the Session Manager to connect to the database.	The version of your database software is incompatible with the Session Manager.	Consult “XA Session Manager for Oracle OCI, version 9i Client” and “DirectConnect Session Manager for Oracle OCI, version 9i Client” to be sure that you are using the supported versions of the database software.
6007	A connection association error occurred when the Session Manager attempted to allocate a database connection.	The database server may be unavailable, or may be misconfigured. You may have used an invalid Coordinator when using non-ITS-managed transactions.	Check with the database administrator to ensure that the database server is up and running; check “XA Session Manager for Oracle OCI, version 9i Client” and “DirectConnect Session Manager for Oracle OCI, version 9i Client” for database configuration requirements. Be sure to use a valid Coordinator in any invocations of the <code>getConnection()</code> method.
6008	The limit on the maximum number of connections has been reached.	The maximum number of connections has been reached. The database server is overloaded.	Raise the limit for the maximum number of connections to this database. Reduce the load on the database server.
6009	Another thread already holds the connection to the database for this transaction.	You are using the DirectConnect Session Manager, and have attempted to connect to the database more than once for the transaction.	If you are using the Direct Connect Session Manager, be sure to use only one Connection object at a time for a transaction.

Error code	Description	Possible causes	Solutions
6010	The Session Manager could not retrieve the database's native connection handle from the XA libraries.	The version of your database software is incompatible with the Session Manager.	Consult "XA Session Manager for Oracle OCI, version 9i Client" and "DirectConnect Session Manager for Oracle OCI, version 9i Client" to be sure that you are using the supported versions of the database software.
6011	The thread that acquired the connection is not the same thread as the one currently trying to operate on it.	You may be trying to share a connection among threads.	See "XA Session Manager for Oracle OCI, version 9i Client" and "DirectConnect Session Manager for Oracle OCI, version 9i Client" for restrictions.
6012	The Session Manager could not register the Resource Director specified in the connection profile.	The Resource Director is not running at the specified OSAGENT_PORT. The name of the Resource Director is incorrectly specified in the connection profile.	Make sure that an instance of the Resource Director is running at the specified OSAGENT_PORT. See "Starting an XA Resource Director" . Verify that the name entered in the Resource Director Name field of the connection profile is correct. See "Integrating VisiTransact with databases using the Session Manager" for instructions.
6013	A transaction context was not established when the connection was requested.	The transaction context did not exist when <code>getConnection()</code> was invoked because <code>begin()</code> was not called before it.	Make sure your program invokes <code>begin()</code> before calling <code>getConnection()</code> .
6014	An invalid value was specified for the <code>SMconnection_idle_timeout</code> attribute.	A negative value was specified for <code>SMconnection_idle_timeout</code> .	Specify a positive value for this attribute.
6015	The Session Manager could not validate the explicit Coordinator used.	You are processing a transaction with explicit propagation, and passed a Coordinator object to the Session Manager for a transaction that rolled back already.	When processing a transaction with explicit propagation, you must ensure that the Coordinator object you pass is valid.
6016	An error occurred during the connection release process.	When the Session Manager was releasing the connection, the connection somehow became invalid—perhaps due to a database server crash.	Check that your database server process is still running.
6017	The Session Manager could not perform the requested disconnect from the database.	When the Session Manager was releasing the connection, the connection somehow became invalid—perhaps due to a database server crash.	Check that your database server process is still running.
6018	The Session Manager encountered an unknown attribute in the connection profile.	Perhaps you are using an older connection profile that is incompatible with this version of the Session Manager. The connection profile may be corrupted.	Recreate the connection profile using the VisiBroker Console. See "Integrating VisiTransact with databases using the Session Manager." Recreate the connection profile using the VisiBroker Console. (See above)
6019	A profile attribute that is necessary for obtaining a connection is not set.	The connection profile does not have a value for this connection attribute. The connection profile may be corrupted.	Use the VisiBroker Console to verify the settings in your connection profile. Recreate the connection profile using the VisiBroker Console.
6020	The infotype for an argument passed to the <code>getInfo()</code> method is invalid.	The <code>getInfo()</code> invocation in your program is passing an incorrect infotype.	See <code>getInfo()</code> in the <i>VisiBroker for C++ API Reference</i> for common infotypes accepted by the <code>getInfo()</code> method. See "XA Session Manager for Oracle OCI, version 9i Client" and "DirectConnect Session Manager for Oracle OCI, version 9i Client" for database-specific infotypes that are accepted.

Error code	Description	Possible causes	Solutions
6021	The <code>hold()</code> operation is not supported.	The application called <code>hold()</code> on a connection which does not support that operation.	Use <code>isSupported()</code> to determine connections for which this operation is valid.
6022	The connection profile name passed as an argument to a method invocation is invalid.	A method invocation in your application passes an invalid connection profile name (generally a null, or a string of 0 length).	Verify that the method uses a valid connection profile name. Also use the VisiBroker Console to verify that the connection profile is indeed set up.
6024	The Session Manager was unable to obtain a connection because it reached its maximum wait time for an available connection.	A deadlock has occurred between the current thread (which is requesting a connection) and another thread (which holds the requested connection). To resolve the deadlock, the current thread has been removed from the queue.	To optimize your code for deadlock avoidance, implement a retry loop that invokes <code>getConnection()</code> until it is successful. See “Data access using the Session Manager” for information.
6025	An invalid argument was passed at the command line.	A command-line argument may have been misspelled.	Verify the spelling of all command-line arguments. See “Commands, utilities, arguments, and environment variables.”
6026	An invalid value was specified for a command-line argument.	An invalid value was specified for a command-line argument.	Verify the valid values for all command-line arguments. See “Commands, utilities, arguments, and environment variables.”
6032	A commit has been attempted for this Resource, but has failed.	Illegal database operations were attempted during the commit, and the database rejected these operations.	See “XA Session Manager for Oracle OCI, version 9i Client” and “DirectConnect Session Manager for Oracle OCI, version 9i Client” for troubleshooting information about the Session Manager for your database.
6033	A rollback has been attempted on this Resource, but has failed.	Illegal database operations were attempted during the rollback, and the database rejected these operations.	See “XA Session Manager for Oracle OCI, version 9i Client” and “DirectConnect Session Manager for Oracle OCI, version 9i Client” for troubleshooting information about the Session Manager your database.
6034	The error messages shown are native to the database you are using.	Native connection or transaction management calls to the database failed.	See “XA Session Manager for Oracle OCI, version 9i Client” and “DirectConnect Session Manager for Oracle OCI, version 9i Client” for troubleshooting information about the Session Manager for your database.
6035	An internal error occurred.	An internal application error has occurred in the Session Manager.	Contact VisiBroker Technical Support.
6040	The Session Manager could not register a Resource with an instance of the VisiTransact Transaction Service.	An instance of the VisiTransact Transaction Service is not running at the specified OSAGENT_PORT.	Make sure that an instance of the VisiTransact Transaction Service is running at the specified OSAGENT_PORT.
6042	The specified Resource Director is unavailable.	An instance of the Resource Director is not running at the specified OSAGENT_PORT.	Make sure that an instance of the Resource Director is running at the specified OSAGENT_PORT.
6043	An error occurred during recovery.	The XA Resource Manager for the database is not running. The XA Resource Manager for the database experienced an error.	Make sure the database's XA Resource Manager is running and available. Check the database error log for a more specific indication of the error that occurred.
6046	The XA calls of the Session Manager have returned the error code listed in the message.	A database error occurred.	See “Problem determination” for troubleshooting tips.

Error code	Description	Possible causes	Solutions
6047	The Session Manager for Sybase CTLib has detected two connections with same LRM name, but different user names and passwords.	Two connection profiles in the same server process share the same database name, but have different passwords.	Create a new connection profile for one of these profiles that uses a different LRM. Modify your server process to use the new connection profile where appropriate.
6048	The connection profile has an unrecognized format.	Perhaps foreign files were written to the directory where the persistent store files are located, or the connection profile has been otherwise corrupted.	Recreate the connection profile using the VisiBroker Console.
6049	The Session Manager could not open the connection profile.	There could be a file permission problem.	Be sure that the file has the correct permissions.
6050	The Session Manager encountered an error while reading the connection profile.	Perhaps foreign files were written to the directory where the persistent store files are located, or the connection profile has been otherwise corrupted.	Recreate the connection profile using the VisiBroker Console.
6051	The Session Manager encountered an error while writing to the connection profile.	Perhaps the file or directory does not have the correct permissions. The disk is full, and the Session Manager cannot save the connection profile. You may have specified an incorrect path to the connection profile.	Be sure that the file and directory have the correct permissions. Free disk space so that the Session Manager can save the connection profile to disk. Be sure to specify the correct path.
6052	The specified path is invalid.	You may have specified an incorrect path to the connection profile.	Be sure to specify the correct path.
6053	The specified connection profile already exists.	You have tried to create a connection profile with a name that is already taken.	Choose a different name for this connection profile, or delete the other connection profile.
6054	The specified connection profile does not exist.	Perhaps you have misspelled the connection profile name.	Check that you have correctly specified the connection profile name.
6055	The connection profile could not be removed.	Perhaps the file or directory does not have the correct permissions.	Be sure that the file and directory have the correct permissions.
6056	The XA Resource Director was given an DirectConnect (non-XA) Session Manager connection profile.	You attempted to start an XA Resource Director with a connection profile created for the DirectConnect Session Manager.	Make sure to start the XA Resource Director with a connection profile that was created for the XA Session Manager.
6058	Previous <code>hold()</code> call has timed out; connection must be released.	The timeout value supplied to the <code>hold()</code> call has expired and the connection object may no longer be used.	Call <code>release()</code> on this connection and call <code>getConnection()</code> again if still required.

VisiTransact transaction log error codes

The following table lists error codes for the VisiTransact transaction log.

Error code	Description	Possible causes	Solutions
8001	An internal error occurred in the logger module. See the message log for details.	See the error text in the message log.	If you cannot resolve this error from the error text displayed in the message log, contact VisiBroker Technical Support.

23

Problem determination

This appendix provides information about how you can determine the causes of failures. It focuses on developer-specific issues of problem determination, spending more time on development rather than deployment.

General approaches

A starting point to help research problems you might be encountering is the host's message log located in `vbroker\admits`.

Dealing with problems in transactions

There are a few typical problems that can occur within applications that use transactions and VisiBroker VisiTransact:

- **The transaction experiences a timeout.** There are several situations related to the expiration of a timeout. A transaction may have timed out and rolled back before the transaction originator issued `commit()`, or you may have received a `CORBA::OBJECT_NOT_EXIST` exception when trying to register a Resource. If you experience these problems, make sure that your timeout period is long enough.
- **The VisiTransact Transaction Service disappears.** If the VisiTransact Transaction Service instance is restarted or fails while a transaction is underway, you may temporarily receive a `CORBA::NO_IMPLEMENT` exception when invoking VisiTransact methods.
- **A Resource Director is unavailable.** The Session Manager will throw a `VISSessionManager::Error` exception if it cannot find an XA Resource Director when the Session Manager attempts to obtain a new connection to a Resource Manager. Additionally, the VisiTransact Transaction Service should roll the transaction back if a participating Resource Director is unavailable at prepare time. (This is the same behavior experienced when any other Resource is unavailable at commit time.)

- **You receive a `CosTransactions::NoTransaction` exception.** You will get this exception if there is no transaction context. It means that your application tried to attempt a connection without first beginning a transaction.
- **Session Manager configuration files are unavailable.** You might be requesting the wrong connection profile.

Index

Numerics

2PC summary 77

A

applications in C++ 44
 embedding 45
 standalone 45
approaching transactions 41
arguments 173, 179
attributes
 viewing 117
attributes
 Session Manager Connection Profile for Oracle
 XA 162, 169

B

backward compatibility 89

C

C++ example 13
C++ exmple
 binding 17
 building 24
 committing and rolling back 19
 files 14
 handling exceptions 19
 IDL 15
 invoking methods 18
 objective 15
 obtaining references 18
 ORB initialization 16
 overview 13
 prerequisites 14
 running 25
 Smart Agent 25
 transaction beginning 17
 transaction originator 16
 write the bank object 21
 write the bank server 20
 write the transactional object 22
C++ VisiTransact applications 44
checked behavior, ensuring 68
class
 ITSDataConnection 128
client behaviors
 policy interfaces 38
client requirements
 Oracle7 with DirectConnect 168
client-side, deploying 103
codes
 error 183
command-line arguments 179
commands 173
 overview 173
commit 76
commit vote 77
committing with Terminator 63
compatability 89
completion 67

 coordinating with Resource objects 73
 ensuring 67
 participating 74
 understanding 73
completion issues 120, 121
completion, heuristic 69
configuartion server 108
configuration
 database configuration with Oracle
 DirectConnect 169
 issues with Oracle DirectConnect 168
 issues with Oracle XA 160
 Oracle with XA 161
connection
 releasing 115
Connection deallocation 115
connection handle
 native 114
connection management 124
Connection object
 obtaining 113
connection pool 94
connection pooling
 optimizing 114
connection profile
 configuring 139
 creating 139
 deleting 140
 editing 140
 filtering 140
 refreshing 140
connection profile sets 101
connection profiles 93, 127, 139
 XA Resource Director 103
connection profiles, modifying 102
ConnectionPool reference
 obtaining 112
 using 113
connections, associating with transactions 93
connections, configuring 93
connections, releasing 94
console 133
 locating transaction service instances 135
 overview 133
 session manager 134
 starting 134
 transaction services section 133, 134
context
 multiple transactions 53
context management 42
 direct and indirect 41
context, getting explicit from Current 63
contexts
 explicit transaction 113
control object 61
CORBA compliance 3
CORBA overview 8
CORBA transaction service 8
CORBAservices specification 4
creating transactions 49, 59
Current
 getting explicit context 63

- interface 51
- obtaining reference 50
- using 49, 50

Current extensions 56

D

- data access failures
 - lock from in-doubt transactions 164
 - transaction timeout 164
 - with Oracle OCI and XA 164
- database configuration
 - for Oracle DirectConnect 169
 - Oracle with XA 161
- database connection
 - opening 92
- database integration 2, 99, 101
- database integration with VisiTransact 91
- database integration with XA 100
- database issues
 - for Oracle Call Interface (OCI) 159
 - with Oracle OCI with the DirectConnect Session Manager 167
- database preparation 101
- database resources
 - pluggable 123
- deployment issues 97
- DirectConnect 101
 - session Manager with Oracle OCI 167
- DirectConnect and XA coexistence 98
- DirectConnect issues 120
- DirectConnect restrictions 97
- DirectConnect Session Manager 167
- DirectConnect session managers 96
- directory structure 105
- disappearing transactions 191
- distributed transactions 7
- distributed update problems
 - with Oracle XA 164

E

- embedded instance of VisiTransact 46
- ensuring transactions 55
- environment variables 173, 181
 - required with Oracle and XA 161
 - required with Oracle9i and DirectConnect 169
- error codes 183
 - Session Manager 186
 - transaction log 189
 - transaction service 184
- error messages
 - message log 163, 170
 - Oracle OCI trace files 163
 - Oracle OCI with DirectConnect 171
 - Oracle OCI with XA 165
- example in C++ 13
 - binding 17
 - building 24
 - committing and rolling back 19
 - files 14
 - handling exceptions 19
 - IDL 15
 - invoking methods 18
 - objective 15
 - obtaining references 18

- ORB initialization 16
- overview 13
- prerequisites 14
- running 25
- Smart Agent 25
- transaction beginning 17
- transaction originator 16
- write the bank object 21
- write the bank server 20
- write the transactional object 22

example integration 119

exception

- NoTransaction 191

exceptions 116

- OTS 71

explicit context, getting from Current 63

explicit propagation 62

explicit to implicit 63

explicit transaction contexts 113

extensions to Current 56

F

- failure recovery 78
- filtering message logs 138
- filtering transactions 137
- forcing heuristics 105

H

- handle
 - native connection 114
- header files 46
- heuristic
 - forcing completion with Oracle OCI and XA 165
- heuristic completion 69
- heuristic log, interpreting 82
- heuristic reporting, enabling 70
- heuristic.log file 81
- heuristics 81
 - introduction 81
- heuristics, forcing 105
- heuristics, viewing 137
- hung transactions 136

I

- in_doubt transactions
 - with Oracle OCI and XA 164
- in-doubt transactions 136
- information, obtaining 56, 65
- init.ora
 - with Oracle DirectConnect 169
 - with Oracle XA 161
- init.ora parameters 161
- installation
 - issues with Oracle DirectConnect issues 168
 - issues with Oracle XA 160
 - requirements for Oracle 160
 - requirements for Oracle9i 168
- instance, discovering 54
- integrating applications 44
- integration example 119
- integration preparation 111
- interface
 - native handle acquisition 128

Index

interface definition 127
interface, Current 51
InvocationPolicy interface 38
ITSDataConnection class 128

L

libraries, deploying 103
log, heuristic 81
log, heuristic, interpreting 82

M

marking transactions for rollback 56, 65
message log 138
 Message Log 163
 VisiTransact Message Log 170
message log filtering 138
message log, trimming 138
migration 89
migration from transaction object 89
multiple threads 53
multiple transactions 53
multithreading 4
multithreading transactions 44

N

name, transaction, obtaining 65
native connection handle 114
native handle acquisition interface 128
NonTxTargetPolicy interface 38
NoTransaction exception 191

O

OAD, registering XA Resource Director with 103
obtaining Current object reference 50
obtaining transaction information 56
OCI 159
OMG extensions 4
open standards transaction processing 4
optimizing connection pooling 114
ORA-01017 171
ORA-12154 165, 171
Oracle
 client requirements with XA 160
 configuration
 issues with XA 160
 database configuration for Oracle XA 161
 installation issues with XA 160
 installation requirements for Oracle with XA 160
 programming restrictions with Oracle XA 163
 required environment variables for XA 161
 server requirements with XA 160
 software requirements for XA 160
 troubleshooting with XA 163
 using the XA Session Manager 159
Oracle Call Interface 159
Oracle database integration 2
Oracle OCI
 client requirements with DirectConnect 168
 disallowed OCI calls with XA 163

error messages 163, 170
error messages with DirectConnect 171
error messages with XA 165
forcing heuristic completion 165
server requirements with DirectConnect 168
software requirements with DirectConnect 168
software requirements with XA 160

Oracle requirements
 Oracle with XA 160
Oracle XA
 database configuration 161
Oracle XA Session Manager 159
Oracle7
 software requirements for DirectConnect 168
Oracle9i
 configuration issues
 with DirectConnect 168
 database configuration with DirectConnect 169
 installation issues with DirectConnect 168
 installation requirements for Oracle9i with
 DirectConnect 168
 programming restrictions with Oracle
 DirectConnect 170
 required environment variables with
 DirectConnect 169
 troubleshooting with DirectConnect 170
 troubleshooting with XA 170

OTS
 starting 134
ots 173, 174
OTS exceptions 71
OTS policy interface 37
overview
 CORBA 8
 transaction processing 7

P

performance tuning 105
performance tuning for Session Manager 105
performance tuning for XA 105
persistent store 105
persistent store copied 107
persistent store files 104
persistent store on each node 107
persistent store, deploying 107
persistent store, shared file system 107
pluggable database resources 123
pluggable module, writing 127
Pluggable Resource Interface API
 programming restrictions 130
policy interfaces
 InvocationPolicy 38
 NonTxTargetPolicy 38
 OTS 37
 transaction object 37
policy migration 89
pooling
 connection optimizing 114
pooling connections 94
preparation, Resource 75
profile sets 138

- programming restrictions
 - disallowed calls 163
 - Pluggable Resource Interface API 130
 - using Oracle with Oracle XA Session Manager 163
 - using Oracle9i with Oracle DirectConnect Session Manager 170
- propagating transactions 49, 55, 59
- propagation 42
 - implicit and explicit 42
- propagation, explicit 62
- propagation, explicit to implicit 63

R

- recovery, transaction 95
- recovery from failure 78
- recovery issues 120, 121
- registration, Resource 75
- releasing connections 94
- reporting, enabling heuristic 70
- requirements
 - installation issues with Oracle 168
 - installation with Oracle 160
 - Oracle software with XA 160
- Resource Director unavailable 191
- Resource objects, coordinating completion 73
- Resource preparation 75
- Resource registration 75
- resource registration 93
 - DirectConnect 97
- Resource vote return 76
- rollback 76
- rollback summary 78
- rollback, marking 56