CHAPTER 8

Transactions

In an enterprise system, maintaining the integrity of data across various applications and machines is critical. Regardless of the scope of the application, at least some aspects of transaction processing have to be implemented to guarantee the integrity of the data. However, developing code to handle data integrity can be very challenging. In this chapter, we will look at the issues involved and examine how the transaction support under COM+ helps simplify component development.

THE STOCK TRADER

We need an example to explore the transaction support under COM+. As I have dabbled in trading stocks over the Internet, I would like to use a simple stock trading system as an example. As for those lessons learned when trading stocks, I will leave that for another book.

Our brokerage firm, *MyBroker.com*, allows clients to trade stocks over the Internet. In order to do so, a client has to maintain an account with the firm.

Figure 8.1 identifies the requisite components to set up our stock trading system. The figure also illustrates the interaction between these components.

The trading system is based on a three-tier Windows DNA strategy.

The presentation layer (the first tier) is a Web-based user interface.

The business logic (the second tier) consists of three components: `AccountMgmt.DLL`, `StockExchange.DLL`, and `TradeMgmt.DLL`.

The data for the clients and the stocks is stored in two different databases (the third tier): `AccountsDB` and `StocksDB`.

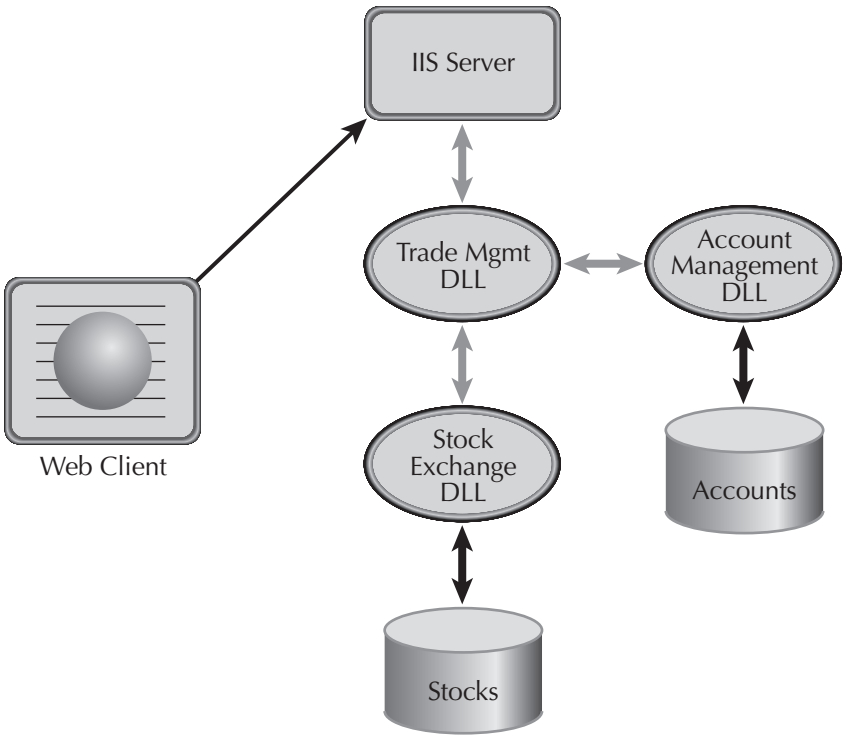

Figure 8.1 A sample stock trading system.

The account management component maintains the clients' accounts. The stock management component maintains the stocks' database. The trade management component lets the client buy a stock. It interacts with the other two components to update the respective databases.

The Database Layer

In this application, the Microsoft Data Engine (MSDE) will be the database server. It is a Microsoft SQL Server 7.0-compatible data storage server and is packaged with Microsoft Office 2000. Details of installing MSDE can be found in the MSDN article "Creating and Deploying Access Solutions with the Microsoft Data Engine" [Smi-99] by Scott Smith.

Using MSDE

MSDE comes with a command line program, `osql.exe`. We will use this program to create our databases. To use this program interactively, you can type:

```
osql.exe -U userid
```

where `userid` is the identification of the user allowed to access the database server. In our example, the `userid` is `sa` and the password field is empty.

Program `osql.exe` also lets you run SQL statements in a batch mode. Simply create a file containing the SQL statements and run the program specifying the filename as a parameter, as shown in the following example:

```
osql.exe -U sa -i MyQueryFile.sql
```

That's it as far as using MSDE goes. Now let's create our databases!

The Accounts Database

The `AccountsDB` database maintains account balances for our clients. It contains one table, `Accounts`, that defines two fields, `Client` (the client's name) and `Balance` (funds that can be used to purchase stocks). Table 8.1 shows the data stored in this table.

Table 8.1 `AccountsDB` **Database**

Client	*Balance*
Don	100000
Chris	90000
Richard	80000

To create this database, run the following SQL statements:

```
create database AccountsDB
go
use AccountsDB
create table Accounts ([Client] varchar (15) NOT NULL,
  [Balance] int NOT NULL)
create unique index Client on Accounts([Client])
insert into Accounts Values ('Don', '100000')
insert into Accounts Values ('Chris', '90000')
insert into Accounts Values ('Richard', '80000')
go
quit
```

The `go` SQL statement explicitly forces the execution of preceding SQL statements. For more information on SQL statements, consult the MSDE documentation.

The Stocks Database

The `StocksDB` database maintains information on the stocks that are currently traded on our fictitious stock exchange. It contains one table, `Stocks`, that defines three fields, `Symbol` (for stock symbols), `Shares` (number of outstanding shares for the stock that may be purchased at the market price), and `MarketPrice` (current market price for the stock). Table 8.2 shows the data stored in the `Stocks` table.

Table 8.2 `StocksDB` **Database**

Symbol	*Shares*	*MarketPrice*
MSFT	50000	95
INTC	30000	75

To create this database, run the following SQL statements:

```
create database StocksDB
go
use StocksDB
create table Stocks ([Symbol] varchar (5) NOT NULL,
  [Shares] int NOT NULL,
  [MarketPrice] int NOT NULL)
create unique index [Symbol] on Stocks([Symbol])
insert into Stocks Values ('MSFT', '50000', 95)
insert into Stocks Values ('INTC', '30000', 75)
go
quit
```

Now let's take a look at the various components of the business logic.

The Business Logic

The Account Management Component

Component `AccountMgmt.DLL` is used to update the `AccountsDB` database. It has just one interface, `IAccountMgr`, that supports just one method, `Debit`. The interface is defined as follows:

```
interface IAccountMgr : IDispatch
{
  HRESULT Debit([in] BSTR bsClient, [in] long lAmount);
};
```

Method `Debit` decreases the account balance of the specified client by the amount specified. The implementation is shown in the following code fragment:

```
STDMETHODIMP CAccountMgr::Debit(BSTR bsClient, long
lAmount)
{
  try {
    ADOConnectionPtr spConn = OpenAccountsDB();
    long lCurrentBalance = GetBalance(spConn, bsClient);
    if (lCurrentBalance < lAmount) {
      return Error(_T("Not enough balance"),
        GUID_NULL, E_FAIL);
    }
    long lNewBalance = lCurrentBalance - lAmount;
    UpdateBalance(spConn, bsClient, lNewBalance);
  }
  catch(_com_error& e) {
    return Error(static_cast<LPCTSTR>(e.Description()),
      GUID_NULL, e.Error());
  }

  return S_OK;
}
```

The code snippet here uses Microsoft's Active Data Objects (ADO) to manipulate the database. ADO simplifies programming by isolating the details of underlying ODBC (Open Database Connectivity) drivers and/or native OLE DB drivers. In the simulation program, ADO uses a native OLE DB driver called `SQLOEDB` to access the MSDE database. Covering ADO is beyond the scope of this book. However, the code snippets that I will be presenting should illustrate the use of ADO interfaces clearly. More information on ADO can be found on the Microsoft platform SDK documentation. In particular, the SDK article, "Migrating from DAO to ADO using ADO with the Microsoft Jet Provider" [Hen-99] has a great introduction to ADO.

Method `Debit` calls `OpenAccountsDB` to open the `AccountsDB` database. It then calls `GetBalance` to obtain the balance for the specified client. Finally, it calls `UpdateBalance` to update the account balance for the client. The implementation of these methods is as follows:

```
// File StdAfx.h
...
#import "c:\program files\common files\system\ado\msado15.dll"\
  rename ( "EOF", "adoEOF" )
typedef ADODB::_ConnectionPtr ADOConnectionPtr;
typedef ADODB::_RecordsetPtr ADORecordsetPtr;

#define CHECKHR(hr) \
  { if (FAILED(hr)) _com_issue_error(hr ); }

// File AccountMgr.cpp

ADOConnectionPtr CAccountMgr::OpenAccountsDB()
{
  ADOConnectionPtr spConn;
  HRESULT hr =
    spConn.CreateInstance(__uuidof(ADODB::Connection));
  CHECKHR(hr);

  // Use native OLE DB driver for MSDE when connecting to
  // to the database
  _bstr_t bsDSN = "provider=sqloledb;database=AccountsDB";
  _bstr_t bsUser = "sa";
  hr = spConn->Open (bsDSN, bsUser, (LPCTSTR) NULL, NULL);
  CHECKHR(hr);
  return spConn;
}

long CAccountMgr::GetBalance(ADOConnectionPtr spConn,
  BSTR bsClient)
{
  ADORecordsetPtr spRS;
  HRESULT hr = spRS.CreateInstance(__uuidof(ADODB::Recordset));
  CHECKHR(hr);

  // Construct a SQL query
  TCHAR buf[256];
  _stprintf(buf,
    _T("SELECT * FROM Accounts WHERE [client] = '%S'"),
      (LPCWSTR) bsClient);

  // Get the recordset
  _variant_t vConn = static_cast<IDispatch*>(spConn);
  hr = spRS->Open(buf, vConn, ADODB::adOpenKeyset,
    ADODB::adLockPessimistic, ADODB::adCmdText);
  CHECKHR(hr);

  return spRS->Fields->Item["Balance"]->Value;
}
```

```
void CAccountMgr::UpdateBalance(ADOConnectionPtr spConn,
  BSTR bsClient, long lBalance)
{
  // Construct a SQL statement to update the balance
  TCHAR buf[256];
  _stprintf(buf,
    _T("UPDATE Accounts SET Balance = %ld WHERE
      [client] = '%S'"), lBalance, (LPCWSTR) bsClient);

  // Execute the SQL statement
  _variant_t vRecordCount;
  spConn->Execute(buf, &vRecordCount,-1);
}
```

In the code above, if any ADO call fails, an exception of type `_com_error` is issued and further processing is stopped.

The Stock Exchange Component

This component supports one interface, `IStockMgr`. The interface is described as follows:

```
interface IStockMgr : IDispatch
{
  HRESULT BuyStock([in] BSTR bsSymbol, [in] long lShares,
    [out, retval] long* plValue);
};
```

Method `BuyStock` checks to see if the requested stock symbol and the requested number of shares are available in the `StocksDB` database. If a match is found, it reduces the number of available shares in the database and returns the total value of the trade, which is the product of the number of requested shares and the market price of the stock. The implementation for this method follows:

```
STDMETHODIMP CStockMgr::BuyStock(BSTR bsSymbol,
  long lRequestedShares, long *plValue)
{
  try {
    ADOConnectionPtr spConn = OpenStocksDB();

    long lAvailableShares, lMarketPrice;
    GetStockInfo(spConn, bsSymbol, lAvailableShares,
      lMarketPrice);
    if( lAvailableShares < lRequestedShares) {
      return Error(_T("Not enough shares"),
```

```
        GUID_NULL, E_FAIL);
    }
    // Reduce the available number of shares
    lAvailableShares -= lRequestedShares;
    UpdateAvailableShares(spConn, bsSymbol, lAvailableShares);
    *plValue = lRequestedShares * lMarketPrice;
  }
  catch(_com_error& e) {
    return Error(static_cast<LPCTSTR>( e.Description()),
      GUID_NULL, E_FAIL);
  }

  return S_OK;
}
```

The code here references two methods, `GetStockInfo` and `UpdateAvailableShares`. Their implementation is similar to those we saw earlier for the account management component. Consequently, the implementation is not shown here.

The Trade Management Component

Our final component of the business logic, the trade manager, is responsible for the overall management of buying stocks. It supports an interface, `ITradeMgr`. The interface is defined as follows:

```
interface ITradeMgr : IDispatch
{
  HRESULT BuyStocks([in] BSTR bsClient, [in] BSTR bsSymbol,
    [in] long lShares);
};
```

Interface method `BuyStocks` attempts to buy the specified number of shares for the specified stock on behalf of the specified client. To accomplish this, the trade manager interacts with the other two components of the business logic. The implementation for this method is as follows:

```
STDMETHODIMP CTradeMgr::BuyStocks(BSTR bsClient, BSTR bsSymbol,
  long lShares)
{
  try {
    //
    // First operation - Obtain the stocks.
    //
    IStockMgrPtr spStockMgr(__uuidof(StockMgr));
    long lAmount = spStockMgr->BuyStock(bsSymbol, lShares);
```

```
    //
    // Second operation - Debit the clien't account balance
    //
    IAccountMgrPtr spAccountMgr(__uuidof(AccountMgr));
    spAccountMgr->Debit(bsClient, lAmount);
  }catch(_com_error& e) {
    return Error(static_cast<LPCTSTR>( e.Description()),
      GUID_NULL, e.Error());
  }

  return S_OK;
}
```

The Simulation

Consider the case when one of the clients, say, Don, wants to buy 100 shares of `MSFT`. The following `VbScript` code shows the logic:

```
set TradeMgr = CreateObject("TradeMgmt.TradeMgr")
TradeMgr.BuyStocks "Don", "MSFT", 100
```

When this code is executed, 100 shares of `MSFT` are removed from the `StocksDB` database, and Don's account balance is debited by $9500 (recall that the market price of MSFT was $95).

To verify that the transaction succeeded, you can query the databases by running `osql.exe`. This is left as an exercise for you.

Now, consider another scenario. Let's say another client, Chris, wants to buy 1000 shares of `MSFT`. The following `VBScript` code shows the logic:

```
set TradeMgr = CreateObject("TradeMgmt.TradeMgr")
TradeMgr.BuyStocks "Chris", "MSFT", 1000
```

When this code is executed, 1000 shares of `MSFT` are removed from the `StocksDB` database. However, Chris' account balance will not get debited, as he does not have sufficient funds; he is short by $5000.

This transaction has created a problem for us. A thousand shares of `MSFT` have just disappeared from the exchange. At this rate, our stock exchange will pretty soon run into the ground!

An obvious solution is to modify the code so that the stocks are inserted back into the market in case of a failure. However, this solution is neither practical nor maintainable for transactions that involve many complex operations to perform. Moreover, it violates many important requirements for a transaction.

Let's examine the requirements for a transaction.

TRANSACTION THEORY

For our brokerage firm example, a "buy stock" transaction consists of the following two operations:

- Reduce the number of available shares for the specified stock in the `StocksDB` database.
- Debit the balance in the `AccountDB` database.

A transaction must be such that it entirely succeeds or entirely fails. This implies that all of the operations involved in the transaction must be updated successfully or nothing should be updated at all. This all-or-nothing proposition of a transaction is called *atomicity*.

A transaction must be *consistent*. Any individual operation within a transaction may leave the data in such a state that it violates the system's integrity. In our case, after the completion of the first operation, some shares have been taken out of the market. After the completion of the second operation, either the system should roll back to the original state (restore the shares that were taken out), or, upon success, go to a new state that still maintains the overall integrity of the system.

Now consider the case of concurrent transactions. Suppose that 100 shares of a stock X are available in the `StocksDB` database, and transaction A, consisting of two operations, is in progress. The first operation has added 50 shares of X into the `StocksDB` database. This change should be committed only if the second operation succeeds. However, before the second operation completes, another transaction, B, tries to obtain 125 shares of X from the database. Transaction B is able to use the uncommitted changes from transaction A; it actually sees 150 shares of X in the database. This is problematic. What happens if the second operation of transaction A fails and thus the first operation has to be rolled back? Transaction B has been infected with data that never really existed.

To avoid such problems, the system should *isolate* the uncommitted changes. Transaction B should only be able to see the data in the state before transaction A begins or in the state after transaction A completes, but not in some half-baked condition between the two states.

Finally, a transaction must be *durable*, that is, when a transaction is committed, the data sources involved must guarantee that the updates will persist, even if the computer crashes (or the power goes off) immediately after the commit. This requires specialized transaction logging that would allow the data source's restart procedure to complete any unfinished operation.

Atomicity, consistency, isolation, and durability; a transaction should support these properties. This is the ACID test for transactions.

Most transactions are not reversible. However, some irreversible transactions can be undone by applying an equal but opposite transaction. An example of such a pair of operations is registering and unregistering a COM server. A transaction that can undo another transaction is referred to as a compensating transaction.

At this point, it is worth considering what implications this has on the underlying components. How on earth can you ensure that the changes in the system can be unwound if the transaction is aborted at some point? Even for our relatively simple example with just two operations, it is not a trivial task. Think about a transaction that involves many such operations.

Fortunately, COM+ provides the infrastructure to ease dealing with transactions.

Let's see it in action.

COM+ Support for Transactions

At 25,000 feet, the support for transaction under COM+ is very straightforward. A transaction spans across many objects (that may access different databases). If an operation from any one object involved in the transaction fails, it indicates its failure status to COM+. The COM+ infrastructure can then roll back the changes to all the databases involved in the transaction.

A little clarification is in order. All the operations in a transaction need not always deal with databases, though a database operation is the most frequent use-case. Sending an e-mail or copying a file, for example, could also be considered as part of the transaction, and may require a rollback. Generally speaking, rollbacks are applied to any transactional resource, a database being just one such resource.

Two basic questions arise:

- How does the COM+ infrastructure know if an object will participate in a transaction?
- How does the object indicate, or *vote*, on the status of its operation?

Let's tackle the first question.

Configuring Transactions

COM+ defines an attribute on a component called the *transaction attribute*. By setting this attribute, a component tells COM+ to manage transactions on its behalf. When the component's object is activated, COM+ looks at the transaction attribute to determine the type of transaction protection it must provide in the object's context.

Why can't COM+ assume that a component will always participate in a transaction?

Forcing every object to participate in a transaction is not practical. The overhead of adding transaction protection to the context object is not acceptable for a component that has no interest in supporting a transaction.

The transaction attribute can be set from the Component Services snap-in. The property-page for setting transactional attributes is shown in Figure 8.2.

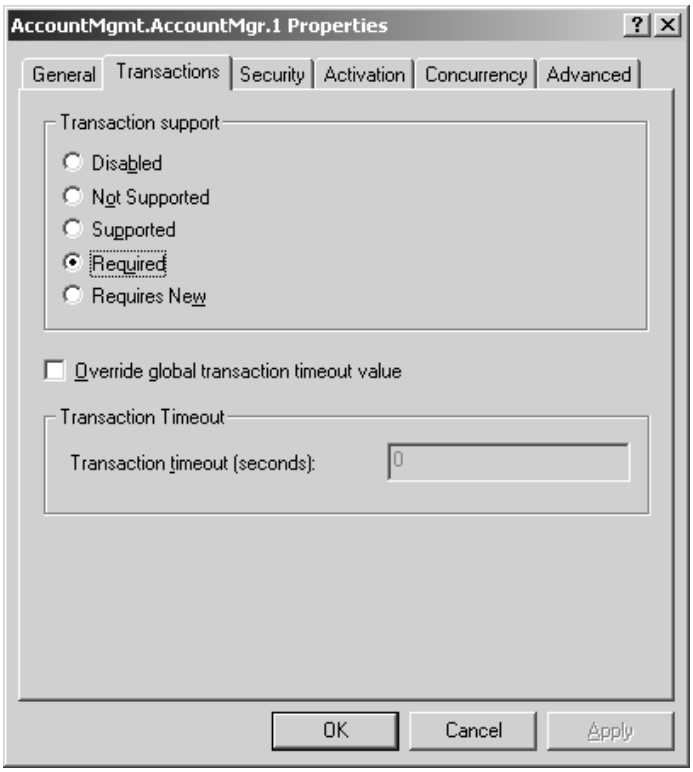

Figure 8.2 Transactional settings on a component.

The transaction attribute can be set to one of the following values:

- **Required:** This value implies that a component *must* have a transaction in order to do its work. If the component's object is activated within the context of an existing transaction, the transaction is propagated to the new object. If the activator's context has no transactional information, COM+ will create a brand new context containing transactional information and attach it to the object.
- **Required New:** Sometimes an object may wish to initiate a new transaction, regardless of the transactional status of its activator. When the *required-new* value is specified, COM+ will initiate a new transaction that is distinct from the activator's transaction. The outcome of the new transaction has no effect on the outcome of the activator's transaction.
- **Supported:** A component with this value set indicates that it does not care for the presence or absence of a transaction. If the activator is participating in a transaction, the object will propagate the transaction to any new object that it activates. The object itself may or may not participate in the transaction.

  This value is generally used when the component doesn't really need a transaction of its own but wants to be able to work with other components.
- **Not Supported:** The component has no interest in participating in a transaction, regardless of the transactional status of its activator. This guarantees that the component's object will neither vote in its activator's transaction nor begin a transaction of its own, nor will it propagate the caller's transaction to any object that it activates. This value should be chosen if you wish to *break* the continuity of an existing transaction.

  Not supported is the default value for all components.
- **Disabled:** If a component will never access a transactional resource, setting the transaction attribute to *disabled* eliminates any transaction-related overhead for the component. This attribute simulates the transaction behavior of a non-configured component.

The transaction attribute on a component can also be specified in the IDL file. The SDK defines the following constants. These constants are defined in the header file `<mtxattr.h>`.

- `TRANSACTION_REQUIRED`
- `TRANSACTION_REQUIRES_NEW`

- TRANSACTION_SUPPORTED
- TRANSACTION_NOT_SUPPORTED

The transaction attribute can be specified on the `coclass` entry in the IDL file, as shown here:

```
import "oaidl.idl";
import "ocidl.idl";
#include <mtxattr.h>
...
[
  uuid(0AC21FA4-DB2A-474F-A501-F9C9A062A63E),
  helpstring("AccountMgr Class"),
  TRANSACTION_REQUIRED
]
coclass AccountMgr
{
  [default] interface IAccountMgr;
};
```

When the component is installed, the Catalog Manager (see Chapter 5) automatically configures the component with the value specified in the IDL file. However, the administrator can override this value from the Component Service snap-in at any time.

Now, let's get the answer to the second question—how does an object cast its vote in a transaction?

Programmatic Voting

Once the components are set to participate in a transaction, each of the component's objects participating in the transaction has to indicate the outcome of its operation(s) individually.

Recall from Chapter 5 that, for a configured component, the state of the context object is available to the component's object via interface `IContextState`. This interface has a method on it called `SetMyTransactionVote`. Following is its prototype:

```
HRESULT SetMyTransactionVote(TransactionVote txVote);
```

Parameter `txVote` can be set to one of two possible values: `TxAbort` to indicate that an operation failed, and `TxCommit` to indicate that the operation succeeded.

Let's revise the account management code to use `SetMyTransactionVote`. The following code fragment shows the changes:

```
STDMETHODIMP CAccountMgr::Debit(BSTR bsClient, long lAmount)
{
  CComPtr<IContextState> spState;
  HRESULT hr = ::CoGetObjectContext(__uuidof(IContextState),
    (void**) &spState);
  if (FAILED(hr)) {
    return hr;
  }

  try {
    ADOConnectionPtr spConn = OpenAccountsDB();
    long lCurrentBalance = GetBalance(spConn, bsClient);
    if (lCurrentBalance < lAmount) {
      spState->SetMyTransactionVote(TxAbort);
      return Error(_T("Not enough balance"), GUID_NULL,
        E_FAIL);
    }
    long lNewBalance = lCurrentBalance - lAmount;
    UpdateBalance(spConn, bsClient, lNewBalance);
  }
  catch(_com_error& e) {
    spState->SetMyTransactionVote(TxAbort);
    return Error(static_cast<LPCTSTR>(e.Description()),
      GUID_NULL, e.Error());
  }

  spState->SetMyTransactionVote(TxCommit);
  return S_OK;
}
```

Similar changes need to be made to the stock manager and the trade manager components. The following code fragment shows the changes for the trade management component:

```
STDMETHODIMP CTradeMgr::BuyStocks(BSTR bsClient, BSTR bsSymbol,
  long lShares)
{
  CComPtr<IContextState> spState;
  HRESULT hr = ::CoGetObjectContext(__uuidof(IContextState),
    (void**) &spState);
  if (FAILED(hr)) {
    return hr;
  }
```

```
  try {
    //
    // First operation - Obtain the stocks.
    //
    IStockMgrPtr spStockMgr(__uuidof(StockMgr));
    long lAmount = spStockMgr->BuyStock(bsSymbol, lShares);

    //
    // Second operation - Debit the clien't account balance
    //
    IAccountMgrPtr spAccountMgr(__uuidof(AccountMgr));
    spAccountMgr->Debit(bsClient, lAmount);
  }catch(_com_error& e) {
    spState->SetMyTransactionVote(TxAbort);
    return Error(static_cast<LPCTSTR>(e.Description()),
      GUID_NULL, e.Error());
  }

  spState->SetMyTransactionVote(TxCommit);
  return S_OK;
}
```

Let's run the simulation once again. Remember to set all the three components with the `Required` transactional attribute and to reset both the databases to the original values.

For your review, the `VBScript` code for the base client is shown below:

```
set TradeMgr = CreateObject("TradeMgmt.TradeMgr")
TradeMgr.BuyStocks "Chris", "MSFT", 1000
```

When the above code is executed, the transaction will fail, as Chris does not have enough funds to buy the stock. This result is the same as before. However, the difference will become apparent when you examine the `StocksDB` database. The number of shares for `MSFT` has not changed, unlike the earlier simulation where 1000 shares of `MSFT` just disappeared from the database.

COM+ provided automatic transaction support and rolled back the changes when the transaction failed.

This brings us to a new set of questions—how did COM+ know the type of resource a component uses? The simulation program never informed COM+ that it used two MSDE databases. What if the simulation used a normal file as one of the resources? Each type of resource involved in a transaction requires its own specialized rollback. Surely, it is not possible for the

COM+ infrastructure to have an intimate knowledge of rolling back changes for every possible resource type. How, then, can it still support transactions?

It's time to look at the architecture.

THE ARCHITECTURE

Resource Managers

COM+ is an infrastructure. As an infrastructure, it should handle any resource generically; and not know the details of any specific resource.

To access and modify the durable state of a resource in a generic fashion, COM+ relies on a software component called the *Resource Manager (RM).*

A resource manager is a software component that has an intimate knowledge of a specific type of resource, such as a relational database. Under the influence of a transaction, the RM keeps track of the changes made to the resource. If the transaction aborts, the RM can roll back these changes on the resource and bring it back to the original state. A simple RM, for example, may *buffer* the changes made to the resource and persist the changes only if the transaction commits.

There are many commercially available RMs, including the ones for Microsoft SQL Server, Oracle, IBM DB2, Informix, and Sybase. The database server used in the simulation program, MSDE, also provides its own RM.

When a client instantiates an RM, the client gets a *proxy* to the RM. OLE DB drivers and ODBC drivers are examples of RM proxies. The RM proxy provides APIs to access the RM. Typically, the RM proxy provides COM interfaces, although it is not a requirement. ODBC drivers, for example, do not provide COM interfaces.

An RM proxy is typically implemented as part of another software component called the *Resource Dispenser* (RD). Unlike a resource manager that manages the durable state of a resource, a resource dispenser manages the non-durable state of the resource, such as the number of connections to a resource. We will cover resource dispensers in Chapter 11 when we discuss scalability issues.

A transaction can involve many resource managers who may span multiple machines across the network. If some operation in a transaction fails, all the participating resource managers need to be informed so that the changes to the resources can be rolled back. This implies that some service should exist that can coordinate all the resource managers involved in the distributed

transaction. This service does exist and is called the Microsoft Distributed Transaction Coordinator (MS-DTC).

The Distributed Transaction Coordinator

As the name implies, the Distributed Transaction Coordinator (DTC) coordinates a transaction that could potentially be distributed across the network. More precisely, the DTC manages the resource managers. Based on the outcome of a transaction, it informs each of the participating resource managers to either abort or commit the changes to their respective resources.

Each system that needs to use transactions must have the DTC installed. If you install MS SQL Server or MSDE, the MS-DTC automatically gets installed.

The MS-DTC is a Windows NT service that can be started and stopped from the service control panel or from the MS-DTC property page in the Component Services snap-in.

A non-transactional client (transactional clients let COM+ manage their transactions) can obtain the DTC using the SDK API `DtcGetTransactionManager` and explicitly request to begin a new transaction. The following code snippet illustrates the process of using the DTC:

```
CComPtr<ITransactionDispenser> spTxDisp;
HRESULT hr = DtcGetTransactionManager(
  NULL,                                    // host name
  NULL,                                    // TM name
  __uuidof(ITransactionDispenser),         // interface
  0,                                       // reserved
  0,                                       // reserved
  0,                                       // reserved
  (void**) &spTxDisp);                     // [out] pointer

CComPtr<ITransaction> spTx;
hr = spTxDisp->BeginTransaction(
  NULL,                                    // outer component
  ISOLATIONLEVEL_ISOLATED,                 // Isolation level
  ISOFLAG_RETAIN_DONTCARE,                 // Isolation flag
  NULL,                                    // Options
  &spTx);                                  // [out] pointer
```

```
... // Enlist RMs and perform resource updates

if (bSuccess) {
  spTx->Commit(0, XACTTC_SYNC_PHASEONE, 0);
}else {
  spTx->Abort(NULL, 0, FALSE);
}
```

When the non-transactional client requests a new transaction, the DTC (more precisely, a part of the DTC called the *transaction manager*) dispenses the transaction as a pointer to interface `ITransaction`. The client can then enlist other appropriate RMs to participate in the transaction.

This COM+ mechanism of letting a non-transactional component handle a transaction manually is referred to as Bring Your Own Transaction (BYOT). An interesting use of BYOT is to manually create a transaction with an arbitrary, long timeout [Mar-00].

The DTC identifies each transaction uniquely as a C structure of type `XACTUOW`. A client can obtain this identification by calling `ITransaction::GetTransactionInfo`. As we will see later, COM+ reinterprets this structure as a GUID.

To commit a transaction, the client calls `ITransaction::Commit`. At this point, the DTC requests each of the enlisted RMs to commit its changes.

What if one of the RMs run into some internal problem and fails to commit?

Two-Phase Commit

To ensure that the all-or-nothing proposition is maintained for a transaction, the DTC mandates that each RM attempt the commitment in two phases, *prepare* and *commit*.

In the prepare phase, the RM should do everything it takes to ensure that the commit phase does not fail. It is up to the developer of the RM to define what "everything" means and how to make it happen. All possible internal problems that an RM can run into should be returned as an appropriate error in this phase.

If no problems are encountered during the prepare phase, the RM saves all of its state information in such a manner that failure to commit can no longer occur. It then returns from the prepare phase with a successful status indicating that it is ready to make the changes permanent.

In the commit phase, the RM applies the just-saved state information and makes the changes permanent. This phase should not fail, unless it runs into some catastrophe such as a power shutdown.

With the breakup of a transaction commitment into two phases, the interaction between the DTC and the RMs gets simplified. The following is the algorithm:

- The client requests the DTC to commit the transaction.
- The DTC sends a *prepare* request to each RM that has been enlisted in the transaction. Upon receiving this request, an RM prepares its internal state.
- If any RM returns a failure status during the *prepare* phase, the DTC informs the rest of the RMs to abort their changes.
- If all the RMs respond positively to the *prepare* request, the DTC requests each RM to commit its changes.

What if the RM runs into a catastrophic failure, such as a power shutdown, in the middle of the commitment phase?

It is the responsibility of the RM to persist its internal state after the *prepare* phase so that it will survive a system failure. Recall that this requirement comes from the *durability* property of the ACID test.

This brief introduction to the DTC and RM is enough for our current discussion. For more information, check out Richard Grimes' book, *Professional Visual C++ 6 MTS Programming* [Gri-99], that has one whole chapter dedicated to the DTC. Also, see Jonathan Pinnock's book, *Professional DCOM Application Development* [Pin-98], for an example of developing your own resource manager and resource dispenser.

Let's see how COM+ supports transactions automatically.

Automatic Transactions through COM+

A configured component indicates its interest in participating in a transaction by means of the transaction attribute.

When an object from such a component is activated, COM+ sets up the object's context to handle transactions.

COM+ automatically begins a transaction when it encounters either of the following conditions:

1. When a non-transactional client activates an object whose component has its transaction attribute set to either the `TRANSACTION_REQUIRED` or `TRANSACTION_REQUIRES_NEW` values.

2. When a transactional client calls an object whose component has its transaction attribute set to the TRANSACTION_REQUIRES_NEW value.

The object responsible for beginning a new transaction is referred to as the *root object* of that transaction. As we will see shortly, this root object has a special role in completing the transaction.

As a corollary, an object whose transaction attribute is set to TRANSACTION_REQUIRES_NEW will always be a root object.

When the root object is activated, COM+ transparently asks the DTC for a new transaction. The DTC returns an ITransaction pointer that COM+ stores in the object's context.

An object that subsequently gets activated within the boundary of this transaction, and is marked as either TRANSACTION_REQUIRED or TRANSACTION_SUPPORTED, will share the transaction.

A collection of one or more contexts that share a transaction is referred to as a *transaction stream*.

To ensure that all the contexts within a transaction stream share only one transaction at a time, COM+ mandates a component that requires a transaction should also require synchronization. Recall from Chapter 5 that COM+ sets up such a component to run under an *activity*.

More precisely, a transaction stream is completely contained inside an activity, but an activity can contain more than one transaction stream.

If an object is participating in a transaction, it can obtain its transaction ID from its context, as shown in the following code fragment:

```
CComPtr<IObjectContextInfo> spInfo;
HRESULT hr = CoGetObjectContext(__uuidof(IObjectContextInfo),
  (void**) &spInfo);
_ASSERT (SUCCEEDED(hr));

GUID tid;
hr = spInfo->GetTransactionId(&tid);
_ASSERT (SUCCEEDED(hr));
```

Note that COM+ returns the transaction ID as a GUID, and not as a XACTUOW structure.

When a transactional object accesses a transactional resource for the first time, the data access layer (such as ODBC and OLE DB) accesses the context's transaction automatically and enlists the corresponding RM with the DTC. In our simulation program, for example, when the account manager object opens the AccountsDB database using ADO, the underlying OLE DB

driver (`SQLOLEDB`) enlists the MSDE resource manager with the DTC in the context of the current transaction. This *auto-enlistment* feature provided by the data access layer simplifies code development and is fundamental to the declarative programming model of COM+.

Each component participating in the transaction casts its vote by calling `IContextState::SetMyTransactionVote`, a method that we have already seen in action.

A transaction completes when the root object of the transaction is deactivated. At this point, COM+ checks to see if all the objects have individually given their consent to commit the transaction. Depending on the consensus, it either calls `ITransaction::Commit` or `ITransaction::Abort` on the current transaction.

The transactional objects themselves do not participate in the two-phase commit process; only the enlisted RMs do. In fact, the transactional objects do not even know about the commitment process, nor do they care.

A transaction also completes when it exceeds its timeout threshold. Transactions are generally designed to be short-lived, as locking a resource for an extended period of time can cause bottlenecks in the system. To ensure efficient performance, COM+ defines a global timeout period for transactions. The default is 60 seconds, but an administrator can change it to any suitable value. COM+ also provides a configuration setting to override the global timeout value for individual components.

If a transaction exceeds its timeout threshold, COM+ will deactivate all the participating objects and abort the transaction.

COM+ 1.0 (the current release) uses a "serializable" level of isolation for transactions. This level of isolation enforces highest level of locking on the underlying resource, thereby providing the highest degree of data integrity. In general, the higher the level of resource locking, the lower the scalability of the application. Under COM+ 1.x (the future release) you will be able to configure the isolation level on a per-component basis. The root object gets to dictate the isolation level for the transaction. Be aware though, that a lower level of isolation increases the chances of incorrect data.

Earlier, I said that, in order to commit a transaction, all the objects participating in the transaction need to cast a positive vote. An obvious improve-

ment that can be made is that, instead of requiring *all* the participating objects to cast a positive vote, it is sufficient that *any one* participating object casts a negative vote. This in fact is the default behavior under COM+. The implication of this is that the developers of a transactional component need not call `SetMyTransactionVote(TxCommit)` on successful operations. They just need to indicate only the failure status (via `TxAbort`).

Lifetime of a Transaction

Consider the following base client `VBScript` code:

```
set TradeMgr = CreateObject("TradeMgmt.TradeMgr")
TradeMgr.BuyStocks "Don", "INTC", 100
TradeMgr.BuyStocks "Chris", "MSFT", 1000
TradeMgr = NULL
```

Recall that Chris does not have enough funds to buy 1000 shares of MSFT; Don, however, does have enough funds to cover 100 shares of INTC. However, if you execute the above code and check the values stored in the database, you will find that even Don was unable to buy the shares he wanted. What went wrong?

Recall that a transaction is considered complete only after the root object of the transaction gets deactivated. In the above lines of code, the root object gets deactivated after executing the two `BuyStocks` statements. As a result, both `BuyStocks` statements are considered to be part of the same transaction. When the second `BuyStocks` statement failed, all the changes, including the one from the first `BuyStocks` statement, were rolled back.

An obvious solution is to release the root object after the first call to `BuyStocks` and immediately recreate it before making the second call.

Though the proposed technique will work, releasing an object and recreating it each time is very inefficient.

Fortunately, COM+ offers a better solution.

COM+ provides a way to deactivate an object even if the base client has not released it. To make this possible, COM+ always returns a proxy pointer to the base client, instead of returning the actual reference to the object. This provides COM+ the flexibility to deactivate the actual object while keeping the proxy alive. When the base client makes a method call on the proxy, COM+ can transparently reactivate the object. This is referred to as *just-in-time* (JIT) activation.

JIT is covered in detail in Chapter 11 when we discuss scalability. The important point to note here is that COM+ enforces a component that requires a transaction to have JIT enabled.

COM+ will automatically enforce `JIT Activation` to `TRUE` and `Synchronization` as `REQUIRED` for any component marked as `TRANSACTION_REQUIRED` or `TRANSACTION_REQUIRES_NEW`.

An object that is JIT-enabled contains a bit in its context called the "*done*" bit or, more precisely, the *deactivate-on-return* bit. COM+ checks this bit after its return from each method call. If the bit is turned on, COM+ will deactivate the object. By default, COM+ turns this bit off before entering a method. However, one can change this behavior at the interface method level from the Component Services snap-in.

The *deactivate-on-return* bit can also be set programmatically by using the method `SetDeactivateOnReturn` available on the interface `IContextState`. The following is its prototype:

```
Interface IContextState : IUnknown
{
  ...
  HRESULT SetDeactivateOnReturn(VARIANT_BOOL bVal);
}
```

Using this method, method `CTradeMgr::BuyStocks` can be revised to deactivate the object on return, as shown in the following code fragment:

```
STDMETHODIMP CTradeMgr::BuyStocks(BSTR bsClient, BSTR bsSymbol,
  long lShares)
{
  CComPtr<IContextState> spState;
  HRESULT hr = ::CoGetObjectContext(__uuidof(IContextState),
    (void**) &spState);
  if (FAILED(hr)) {
    return hr;
  }
  hr = spState->SetDeactivateOnReturn(VARIANT_TRUE);
  _ASSERT (SUCCEEDED(hr));

  try {
    //
    // First operation - Obtain the stocks.
    //
    IStockMgrPtr spStockMgr(__uuidof(StockMgr));
    long lAmount = spStockMgr->BuyStock(bsSymbol, lShares);
```

```
    //
    // Second operation - Debit the clien't account balance
    //
    IAccountMgrPtr spAccountMgr(__uuidof(AccountMgr));
    spAccountMgr->Debit(bsClient, lAmount);
  }catch(_com_error& e) {
    spState->SetMyTransactionVote(TxAbort);
    return Error(static_cast<LPCTSTR>(e.Description()),
      GUID_NULL, e.Error());
  }

  spState->SetMyTransactionVote(TxCommit);
  return S_OK;
}
```

With this change in place, if you execute the base client `VBScript` code once again, you will see that this time Don's trade would go through and Chris' trade would fail, just as expected.

Manual Transactions

Allowing COM+ to automatically manage a transaction simplifies component development. However, there are times when the base client would like to control the outcome of a transaction.

To handle this, COM+ provides a component called the `TransactionContext` class, represented by the PROGID `TxCtx.TransactionObject`.

The `TransactionContext` object supports interface `ITransactionContext`. Following is its definition, along with a short explanation for each interface method:

```
ITransactionContext : IDispatch
{
  HRESULT CreateInstance([in] BSTR pszProgId,
    [retval][out] VARIANT *pObject);      // instantiate an object
  HRESULT Commit();                       // commit a transaction
  HRESULT Abort();                        // abort a transaction
};
```

By calling the methods on the `ITransactionContext` interface, the base client can begin a transaction, compose the work of one or more COM+ components in the transaction, and explicitly commit or abort the transaction. This is illustrated in the following `VBScript` code snippet:

```
Dim txCtx
Set txCtx = CreateObject("TxCtx.TransactionContext")

Dim Accounts
set Accounts = txCtx.CreateInstance("AccountMgmt.AccountMgr")
Accounts.Debit "Don", 10

txCtx.Commit
msgbox "Done"
```

Note that an object that is activated by calling `ITransactionContext::CreateInstance` should belong to a COM+ configured component. Each activated object should cast its transaction vote using the context object. However, using the transaction context, the base client also can participate in the voting process.

Also notice the distinction between an object context and a transaction context. An object context relates to an individual object whereas a transaction context is related to the overall transaction.

COMPENSATING RESOURCE MANAGER

A resource manager has to pass the ACID test; it has to guarantee atomicity, consistency, isolation, and durability. Given the intricate footwork an RM has to perform, implementing an RM is not an easy task.

Let's look at the tasks of a typical RM.

- When a client accesses a transactional resource, the corresponding RM should support enlistment with the DTC. The RM may also make a temporary copy of the resource and lock access to the actual resource (so that no other client can use it).
- When the primary client attempts to modify the resource, the RM has to *record* the change and apply the change to the copy (not the actual resource).
- If the DTC asks the RM to *prepare*, the RM has to play back the recorded sequence, and create an internal state for the commit phase. Alternatively, an RM may delay the playback to the commit phase, if it is confident that the updates will not fail.
- If the DTC asks the RM to *commit*, the RM may use the prepared internal state to commit the changes or play back the recorded sequence and apply the changes to the resource.
- If the DTC asks the RM to *abort*, the RM may just discard the prepared internal state (or the recorded sequence).

Given that a large portion of functionality is common from one RM to another, a reasonable question to ask is if there is a way to share this functionality. This would certainly simplify developing an RM.

It turns out that COM+ designers had already thought of this possibility. COM+ provides a framework to develop RMs. An RM developed using this framework is referred to as a Compensating Resource Manager (CRM).

The developer of a CRM has to write two cooperating components called the CRM worker and the CRM compensator.

The CRM worker exposes necessary COM objects to the clients. When the client requests that the resource be modified, the worker simply records the change (using CRM's service).

The CRM compensator reads the recorded changes (supplied by the CRM service) and either commits or aborts the changes.

Note that there is no direct communication between the CRM worker and the CRM compensator. The only data that has to be passed from the worker to the compensator is the sequence of changes applied on the resource.

To facilitate storing the sequence of changes, COM+ provides a component called the *CRM clerk*. The CRM worker instantiates the CRM clerk and starts recording the changes with the clerk. When the transaction closes, COM+ launches the CRM compensator and calls prepare, commit, or abort in whatever combination that is appropriate, and plays back the sequence of records to the compensator.

The CRM clerk supports an interface, `ICrmLogControl`. The following is its prototype:

```
ICrmLogControl : public IUnknown
{
  [propget] HRESULT TransactionUOW([retval][out] BSTR *pVal);
  HRESULT RegisterCompensator(
    [in] LPCWSTR pwszProgId,
    [in] LPCWSTR pwszDesc,
    [in] LONG lCrmRegFlags);
  HRESULT STDMETHODCALLTYPE WriteLogRecordVariants(
    [in] VARIANT *pLogRecord);
  HRESULT ForceLog();
  HRESULT ForgetLogRecord();
  HRESULT ForceTransactionToAbort();
  HRESULT WriteLogRecord(
    [size_is][in] BLOB rgBlob[],
    [in] ULONG cBlob);
};
```

Method `RegisterCompensator` is used to associate a CRM worker with a specific CRM compensator. It is the responsibility of the CRM worker to call this method.

Parameter `pwszProgId` is the `PROGID` of the CRM compensator that should be associated with the CRM worker. Parameter `pwszDesc` describes the CRM compensator. A transaction-monitoring program can use this description string for display purposes. Parameter `lCrmRegFlags` specifies the possible phases (prepare, commit, or abort) that can be passed to the CRM compensator. For example, if the compensator does not do anything specific to abort a transaction, then `CRMREGFLAG_ABORTPHASE` need not be specified as a possible phase.

Method `WriteLogRecord` can be used to record a change that is being made to the resource. The data is recorded in a form called BLOB (Binary Large Object). A BLOB is a structure that can carry any opaque data as a pointer. The structure of the BLOB is defined as follows:

```
struct BLOB{
  ULONG cbSize;                    // the size of the data
  [size_is(cbSize)] BYTE* pBlobData;    //the actual data
};
```

The CRM worker can record a resource change by passing one or more BLOBS to the method `WriteLogRecord`.

Each call to `WriteLogRecord` results in a single record stored with the CRM clerk. When the transaction completes, the CRM clerk instantiates the CRM compensator and plays back the records in the same sequence as they were originally received.

Associating more than one BLOB with a single record is just a convenience provided to the CRM worker. The CRM clerk internally pastes all the BLOBS together as one big BLOB.

Method `ForceTransactionToAbort` can be used to abort a transaction.

Let's turn our attention to the CRM compensator.

A CRM compensator has to support an interface, `ICrmCompensator`. The following is its prototype:

```
ICrmCompensator : public IUnknown
{
  HRESULT SetLogControl( [in] ICrmLogControl *pLogControl);

  // Prepare phase
  HRESULT BeginPrepare( void);
  HRESULT PrepareRecord( [in] CrmLogRecordRead crmLogRec,
    [retval][out] BOOL *pfForget);
```

```
  HRESULT EndPrepare( [retval][out] BOOL *pfOkToPrepare);

  // Commit phase
  HRESULT BeginCommit( [in] BOOL fRecovery);
  HRESULT CommitRecord( [in] CrmLogRecordRead crmLogRec,
    [retval][out] BOOL *pfForget);
  HRESULT EndCommit( void);
  // Abort phase
  HRESULT BeginAbort( [in] BOOL fRecovery);
  HRESULT AbortRecord( [in] CrmLogRecordRead crmLogRec,
    [retval][out] BOOL *pfForget);
  HRESULT EndAbort( void);
};
```

Data type `CrmLogRecordRead` is a C structure that contains a BLOB (that was previously recorded using `WriteLogRecord`) and some other fields that might be useful for debugging.

The compensator should implement code for all three phases, at least to satisfy the compiler. The DTC enters a phase by calling the `BeginXXX` method on that phase, followed by one or more calls to `RecordXXX`, and completes the phase by calling the `EndXXX` method.

Once a record has been digested in any phase, if the CRM compensator feels that the record serves no purpose to some other phase that it may enter later, it can inform the CRM clerk to lose the record by setting `pfForget` flag to TRUE.

With this brief background, let's build a CRM.

Our CRM will use a text file as a resource. To verify its functionality, we will modify the account manager component from the previous simulation program to use a text file, `W:/DB/Accounts.txt`, as a transactional resource (replacing the MSDE database).

The CRM worker component will support interface `IMyFileDB`, as defined here:

```
interface IMyFileDB : IDispatch
{
  HRESULT Open([in] BSTR bsFilePath);
  HRESULT GetBalance([in] BSTR bsClient,
    [out, retval] long* plBalance);
  HRESULT UpdateBalance([in] BSTR bsClient,
    [in] long lNewBalance);
};
```

With this component in place, the `CAccountMgr::Debit` logic should be modified to use the file-based resource. The revised implementation is shown below:

```
STDMETHODIMP CAccountMgr::Debit(BSTR bsClient, long lAmount)
{
  CComPtr<IContextState> spState;
  HRESULT hr = ::CoGetObjectContext(__uuidof(IContextState),
    (void**) &spState);
  if (FAILED(hr)) {
    return hr;
  }

  try {
    IMyFileDBPtr spConn(__uuidof(MyFileDB));
    spConn->Open("w:/DB/Accounts.txt");
    long lCurrentBalance = spConn->GetBalance(bsClient);
    if (lCurrentBalance < lAmount) {
      spState->SetMyTransactionVote(TxAbort);
      return Error(_T("Not enough balance"), GUID_NULL,
        E_FAIL);
    }
    long lNewBalance = lCurrentBalance - lAmount;
    spConn->UpdateBalance(bsClient, lNewBalance);
  }
  catch(_com_error& e) {
    spState->SetMyTransactionVote(TxAbort);
    return Error(static_cast<LPCTSTR>(e.Description()),
      GUID_NULL, e.Error());
  }

  spState->SetMyTransactionVote(TxCommit);
  return S_OK;
}
```

When `IMyFileDB::Open` is invoked, the CRM worker should first instantiate the CRM clerk and register the associated CRM compensator. The code snippet is shown below:

```
HRESULT CMyFileDB::InitCRM()
{
  if (ISNOTNULL(m_spCrmLC)) {
    m_spCrmLC = NULL;
  }

  HRESULT hr = ::CoCreateInstance(
    __uuidof(CRMClerk),
    NULL,
    CLSCTX_INPROC_SERVER,
    __uuidof(ICrmLogControl),
    (void**) &m_spCrmLC);
```

```
  if (FAILED(hr)) {
    return hr;
  }

  // Register the compensator.
  // Try 5 times if a recovery is in progress
  for(int i=0; i<5; i++) {
    hr = m_spCrmLC->RegisterCompensator(
      L"TextFileDB.MyFileDBCompensator",
      L"My file db compensator",
      CRMREGFLAG_ALLPHASES);

    if (SUCCEEDED(hr)) {
      return S_OK;
    }

    // deal with recovery in progress
    if (XACT_E_RECOVERYINPROGRESS == hr) {
      Sleep(1000); // sleep for a second
      continue; // and try again
    }
  }

  m_spCrmLC = NULL;
  return hr;
}
```

Note that it is possible for the CRM worker to receive an `XACT_E_RECOVERYINPROGRESS` error during the call to `RegisterCompensator`. If this happens, the CRM worker should call the method a few more times until it succeeds.

The data file for our CRM contains clients and their respective balances. The CRM worker loads the file into memory as an STL map. Loading all the data into memory is not always efficient. However, it works for our demonstration.

```
typedef std::map<CComBSTR, long> MYACCOUNTDB;

class CMyFileDB :
  ...
{
  ...

private:
  CComPtr<ICrmLogControl> m_spCrmLC;
  MYACCOUNTDB m_AccountDB;
};
```

I have encapsulated serializing the `MYACCOUNTDB` data type to a file in class `CMyFile` and will not discuss it further. The code can be found on the CD.

There are only two commands that the CRM worker needs to record: the command to open a file and the command to update an account. As the command to obtain the balance does not really change the resource, there is no real need to record it.

To facilitate converting the command information into a BLOB, let's define some relevant data structures:

```
enum DBACTIONTYPE {dbOpen = 0x10, dbUpdate = 0x20};

#pragma warning(disable : 4200)       // do not warn on
                                      // zero-sized arrays

#pragma pack(1)                       // Pack the following
                                      // structures tightly

struct DBACTION {
  DBACTIONTYPE actionType;
};

struct DBACTIONOPEN : public DBACTION
{
  DBACTIONOPEN()
  {
    actionType = dbOpen;
  }
  WCHAR pszFileName[0];
};

struct DBACTIONUPDATE : public DBACTION
{
  DBACTIONUPDATE()
  {
    actionType = dbUpdate;
  }
  long lNewBalance;
  WCHAR pszClient[0];
};

#pragma pack()                        // back to default packing
#pragma warning(default : 4200)       // back to default warning
```

Note that packing the data on the byte boundary is important for reinterpreting a BLOB to its original structure.

Also note that I am defining a zero-sized array for a variable-sized string. I am just taking advantage of the fact that data is stored contiguously in a BLOB.

With these structures in place, the `CMyFile::Open` method can be implemented as follows:

```
STDMETHODIMP CMyFileDB::Open(BSTR bsFilePath)
{
  HRESULT hr = InitCRM();
  if (FAILED(hr)) {
    return hr;
  }

  // Open the file
  USES_CONVERSION;
  LPCTSTR pszFile = W2T(bsFilePath);
  CMyFile file;
  hr = file.Open(pszFile, CMyFile::READ);
  if (FAILED(hr)) {
    m_spCrmLC->ForceTransactionToAbort();
    return hr;
  }

  // Log info with CRM that the file is being opened
  DBACTIONOPEN openAction;

  BLOB blobArray[2];
  blobArray[0].pBlobData = (BYTE*) &openAction;
  blobArray[0].cbSize = sizeof(DBACTIONOPEN);
  blobArray[1].pBlobData = (BYTE*) bsFilePath;
  blobArray[1].cbSize = ::SysStringByteLen(bsFilePath) +
    sizeof(OLECHAR); // account for the end of string
  hr = m_spCrmLC->WriteLogRecord(blobArray, 2);
  if (FAILED(hr)) {
    m_spCrmLC->ForceTransactionToAbort();
    return hr;
  }

  // Now load file into memory
  hr = file.Load(m_AccountDB);
  if (FAILED(hr)) {
    m_spCrmLC->ForceTransactionToAbort();
    return hr;
  }

  return S_OK;
}
```

Method `IMyFileDB::UpdateBalance` records its operations similarly. The code is not shown here.

Now let's build the CRM compensator component.

The CRM component that we are building need not take any specific action in the *prepare* or *abort* phase. Consequently, we will focus on just the *commit* phase. Specifically, we will look at implementing two `ICrmCompensator` methods—`CommitRecord` and `EndCommit`.

Method `CommitRecord` decodes the BLOB and, depending on the action type, either loads the file into memory or updates the in-memory copy with the new balances from the clients, as follows:

```
STDMETHODIMP CMyFileDBCompensator::CommitRecord(
    /* [in] */ CrmLogRecordRead crmLogRec,
    /* [retval][out] */ BOOL __RPC_FAR *pfForget)
{
  *pfForget = FALSE; // don't drop the record
  BLOB& blob = crmLogRec.blobUserData;
  DBACTION* pAction =
    reinterpret_cast<DBACTION*>(blob.pBlobData);
  if (dbOpen == pAction->actionType) {
    DBACTIONOPEN* pActionOpen =
      reinterpret_cast<DBACTIONOPEN*>(pAction);
    m_bsFilePath = pActionOpen->pszFileName;

    // load the contents of the file
    USES_CONVERSION;
    CMyFile file;
    HRESULT hr = file.Open(W2T(m_bsFilePath), CMyFile::READ);
    if (FAILED(hr)) {
      return hr;
    }
    hr = file.Load(m_AccountDB);
    if (FAILED(hr)) {
      return hr;
    }
    return S_OK;
  }

  if (dbUpdate == pAction->actionType) {
    DBACTIONUPDATE* pActionUpdate =
      reinterpret_cast<DBACTIONUPDATE*>(pAction);
    long lNewBalance = pActionUpdate->lNewBalance;
    LPWSTR pwszClient = pActionUpdate->pszClient;
    MYACCOUNTDB::iterator i = m_AccountDB.find(pwszClient);
```

```
    if (i == m_AccountDB.end()) {
      return E_INVALIDARG;
    }
    (*i).second = lNewBalance;

    return S_OK;
  }

  return S_OK;
}
```

Method `EndCommit` saves the in-memory copy back into the file, as shown here:

```
STDMETHODIMP CMyFileDBCompensator::EndCommit(void)
{
  // Save the information back to file
  USES_CONVERSION;
  CMyFile file;
  HRESULT hr = file.Open(W2T(m_bsFilePath), CMyFile::WRITE);
  if (FAILED(hr)) {
    return hr;
  }
  file.Save(m_AccountDB);
  return S_OK;
}
```

Congratulations! You have just finished building your first CRM.

The CRM components can be installed as a server application.[1] It is recommended that both the CRM worker and the CRM compensator for a specific CRM be installed in the same application.

For CRM components it is important to turn the "Enable Compensating Resource Managers" option on (from the Component Services snap-in). Otherwise, a call to `RegisterCompensator` will result in a "catastrophic failure" error.

The CRM worker should be marked with the transaction setting as `REQUIRED` (which will automatically force `JIT Activation=TRUE` and `Synchronization= REQUIRED`). The CRM compensator, however, should be marked with the transaction as disabled, the synchronization as disabled, and the JIT turned off.

[1] It can also be installed as a library application. Check the online documentation for more details.

SUMMARY

In this chapter, we first looked at the issues involved while updating multiple transactional resources.

To ensure system integrity, a transaction has to support four properties: atomicity, consistency, isolation, and durability. Collectively, these properties are referred to as the ACID properties.

A resource manager (RM) is a software component that manages the durable state of a specific type of transactional resource, such as a relational database.

A distributed transaction coordinator (DTC) coordinates a transaction across multiple machines over the network. Each RM involved in a transaction is enlisted with the DTC. When the transaction completes, the DTC informs the participating RMs to either commit the changes made to their respective resources or to abort the changes. A transaction is committed using a two-phase protocol.

COM+ simplifies developing components by automatically managing a transaction. A COM+ component can indicate its interest in transactions by a configurable attribute. When such an object is activated, COM+ sets its context to deal with transactions. A participating object has to individually indicate to COM+ if its operations succeeded or failed. If any participating object indicates a failure condition, COM+ aborts the transaction. If all the participating objects vote positively, COM+ commits the transaction.

If a component is marked as requiring a transaction, COM+ automatically enforces that the component is marked as `JIT Activation=TRUE` and `Synchronization=Required`. By forcing JIT activation, a component can achieve transactional correctness without sacrificing efficiency.

Finally, we looked at the infrastructure provided by COM+ to develop a compensating resource manager.

REFERENCES

[Smi-99] Scott Smith, "Creating and Deploying Access Solutions with the Microsoft Data Engine," Microsoft Development Network Online, Microsoft Corp., January 1999. *http://msdn.microsoft.com/library/techart/msdedeploy.htm*

[Hen-99] Alyssa Henry, "Migrating from DAO to ADO Using ADO with the Microsoft Jet Provider," Microsoft Platform SDK, Microsoft Corp., March 1999. *http://msdn.microsoft.com/library/techart/daotoadoupdate.htm*

[Mar-00] Davide Marcato, "Create and Customize Transactions," *Visual C++ Developers Journal*, January 2000.

[Gri-99] Richard Grimes, *Professional Visual C++ 6 MTS Programming*, ISBN 1-861002-3-94, Wrox Press, 1999.

[Pin-98] Jonathan Pinnock, *Professional DCOM Application Development*, ISBN 1-861001-31-2, Wrox Press, 1998.