Surviving Client/Server:
Three-Tier Applications With MIDAS

by Steve Troxell and Scott Samet

When I first started programming, I cut my data storage teeth on the venerable sequential access text file. Add new records to the end of the file and read all the records from the top of the file until you found the one you wanted. You want fields? Parse the string into its constituent data elements. A really slick program would use a random access file which could jump a specified number of records into a file and grab a single record lying anywhere within the file. Ooooo! There were no indexes, other than what you might handcraft on your own in the form of another data file. There were no record locks for multiple user applications unless you rolled your own. There was no ODBC, SQL Links, DB-LIB, BDE, or any other kind of acronymic drivers between you and your data. It was *real* programming back then. If you could read bytes, you could mess with the data. Not this sissy stuff we do today...

Over the years we've learned how to deal with database engines like dBase and FoxPro. Then we got a grip on client/server and SQL. And just when we think we've finally got it licked, they throw another curve ball at us and call it three-tier architecture.

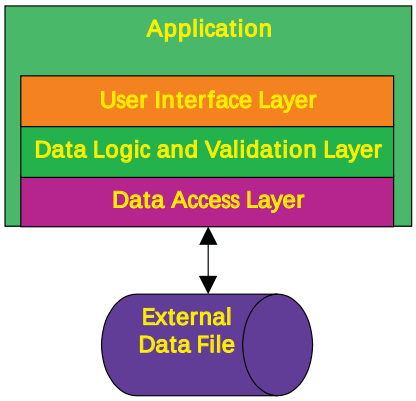
Apparently, our front-end applications aren't good enough even to talk to a database server directly. They want us to develop a middleman app to serve as go-between for the keeper of the data and all the peasant client apps trying to get the real work done. It's kind of like handing over your vacation request to the boss's secretary, who returns it to you sometime later with the boss's signature on it. You don't know how the signature got there, or even if it's *really* the boss's signature. But you don't care, you got a signed vacation request and you're off to the Bahamas. Just remember to take your laptop.

So what is all this talk about three-tier development and just what is Delphi going to do to help us out here? Borland has conjured up some very sophisticated components to facilitate three-tier development with Delphi. This technology comes under the name MIDAS, standing for **M**ulti-**T**ier **D**istributed **A**pplication **S**ervices, and that's what this month's column is all about. For this topic, I've leaned a great deal on Scott Samet, whom some of you may recognize from Team B. I'm fortunate enough to be working semi-directly with Scott now, who was investigating MIDAS before I had a chance to. I am grateful for his help in putting together the information in this article.

Tiering Your Applications

First, let's get a grip on the terminology of *tiers*. The notion of a system being tiered has nothing to do with the number of physical machines the system employs. The determining factor is the number of independent processes or applications that are talking to each other to make the entire system work. For example, a Delphi application using the BDE to access Paradox data is a single-tier system. A web browser fetching HTML pages from a web server is a two-tiered system.

A single-tier architecture is one in which a single application contains all the code to present the user interface, apply the logic, and manipulate the data within an external data file (see Figure 1). Even if some of these layers are implemented within DLLs, it is still a single-tier architecture because there is only one running process handling all the work. For example, within a typical Paradox application, all the code for accessing the data stored in the Paradox tables, as well as the code for applying logic to that data and presenting it to the user, is contained within the same application. If you want to display all the customers in Germany on the screen, then your application must execute code to search the Paradox tables, find the records containing customers in Germany, then display the matching records on the screen. Note that even if the data resides on a remote machine, this is still a single-tier model.



➤ *Figure 1: Single-Tier Architecture*

With the advent of client/server technology came the notion of two-tiered systems. Here we have a database server, running as a separate application, in charge of direct access to the data. Separate client applications connect to this server, making requests to receive data from or apply data modifications to the database. We have two independently running processes communicating with each other, and hence two tiers (see Figure 2). The significance of the separate server tier is that many different client applications, possibly even written in different languages, can share services and logic provided by the server. Also, there can be a significant reduction in network

traffic because most of the record filtering logic can be performed by the database server and only the matching results returned to the client workstations. Finally, the performance of the database server tier can be scaled independently of the client workstations by beefing up the server hardware to deal with an increased load.

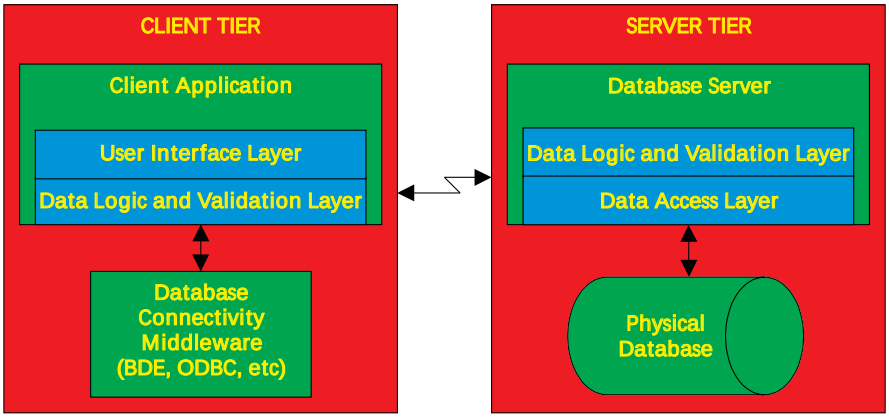
Three-tier systems go one step further in isolating the access to the database by creating an application server between the client front-end and the database server. In this model, the client app does not directly connect to the database server, but instead connects to a 'middle tier' application server, which in turn connects to the database server (see Figure 3).

Can We Talk?

In the two-tier model, the client applications typically communicated with the database server via a native database API or ODBC driver. This driver would normally be implemented in a DLL hosted on each client workstation. This is also how our application server communicates with the database server in a three-tier model. But how do the client applications get in touch with the application server?

Multi-tier applications in Delphi are based on COM objects. In actuality, the application server constituting the middle tier is an OLE Automation server, the client applications are OLE clients. So our objects are able to talk to each other by virtue of COM. For an excellent primer on the wonderful world of COM, check out Dave Jewell's series *Delphi Meets COM* starting in Issue 28 of *The Delphi Magazine*.

COM itself is restricted to inter-object communication on a single machine. To make multi-user, multi-tiered database applications work, we rely on Distributed COM (DCOM), which extends COM's capabilities across multiple machines on a network. In fact, since DCOM is based on TCP/IP, it even allows different machines to communicate via the Internet. This is a significant point in multi-tier development. The in-house workstations can interface with the same application server as a browser-hosted application. This opens quite a few opportunities for leveraging the enterprise data to customers, remote offices, and even off site workers.



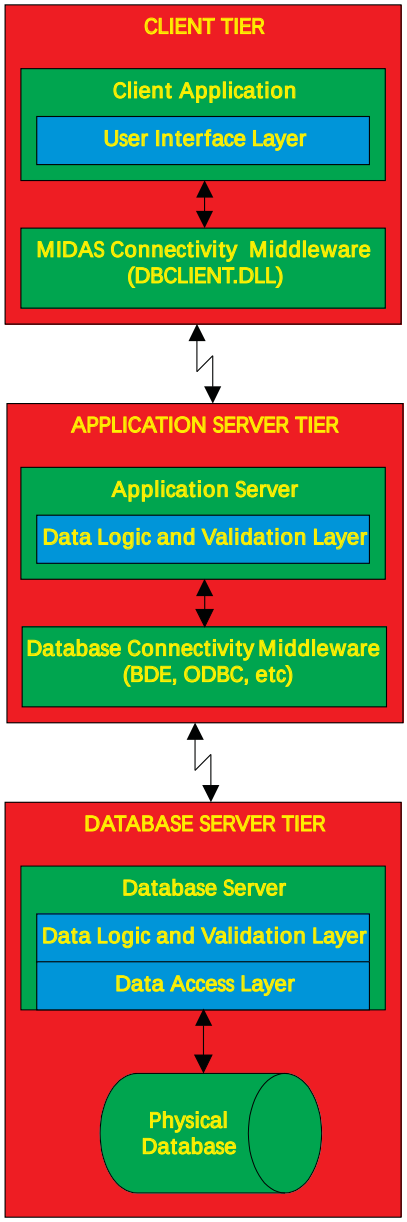
➤ *Figure 2: Two-Tier Architecture*

DCOM is native to Windows NT 4.0 but can also be installed on Windows 95 by downloading the appropriate files from Microsoft's website. Running server objects on NT machines and client objects on Windows 95 machines should pose no problems. However, using a Windows 95 machine as a DCOM server requires a bit more configuration.

Borland's MIDAS technology in Delphi is a layer on top of DCOM. MIDAS facilitates the handling of distributed datasets through `TProvider` (in the server) and `TClientDataSet` (in the client) Delphi components. While these components use DCOM to communicate with each other, MIDAS further encapsulates their services within the `DBCLIENT.DLL` library. MIDAS client applications must be deployed with the `DBCLIENT.DLL` library. MIDAS server applications must be deployed with `DBCLIENT.DLL`, `IDPROV32.DLL`, `STDVCL32.DLL` and, of course, the BDE which is used to access the database server.

Advantages Of Three-Tier Design

Ok, three-tier sounds interesting, but why bother? Actually there are a number of benefits to be gained from this architecture.



➤ *Figure 3: Three-Tier Architecture*

Since the application server is the only piece that directly connects to the database, it is the only machine requiring the database

connectivity middleware such as BDE, SQL Links, ODBC or native database client DLLs. All the installation and configuration of the database access middleware is removed from the numerous client workstations and centralized on one or more application server machines. Client configuration of Delphi applications is drastically simplified by only requiring the single DBCLIENT.DLL library. The total client footprint is also reduced since, at 154Kb, DBCLIENT.DLL is usually vastly smaller than all the attendant database connectivity libraries.

It is significantly easier to centralize business rules and data manipulation logic within the application server than it is to do so within database triggers and stored procedures. You have all the power and speed of Delphi at your disposal to write any code that should be shared across all client applications. This is much like writing shared code within a Delphi DLL, but in this case the shared code can be run on a separate machine, distributing the system load.

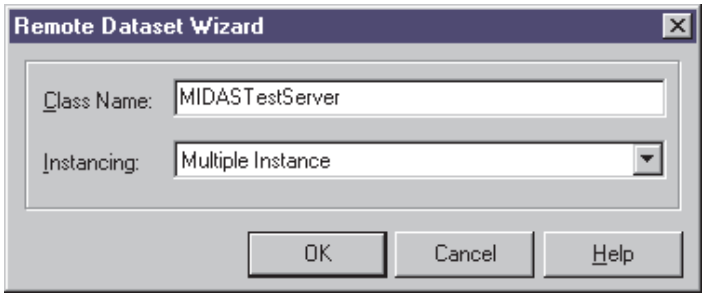
Web-based applications, or web extensions to an existing system, are easier to deploy. Processor intensive work is done on the server end of the connection while a thin client is hosted by the web browser to act as the user interface. MIDAS support is planned to be added to JBuilder, so a JBuilder application can access a MIDAS application server just as easily as a Delphi desktop application.

Automatic load balancing is possible by having identical server objects running on several different machines. Clients can attach to an 'object broker' which will delegate the use of the available server machines to even out the total load. In a similar manner, fail-over safety can be provided by switching to a different server machine when one becomes unavailable or goes down.

My First Multi-Tier Database Application

So how do we go about spinning all this magic with Delphi? Well, our development will obviously be in two parts: the server project and the client project. Life is just a whole bunch easier if we start with the server application. What we must first realize is that we are creating a server *application*, which contains a class definition for an OLE Automation server *object*. The server object descends from a special COM-enabled descendant of `TDataModule`, which is built for us by the Remote Dataset Wizard. The server application wraps around the server object and gives us a user interface into the workings of the server object, if need be. Typically, production server applications will have no user interface.

➤ *Figure 4*



Start a new Delphi project then select `File | New | Remote Data Module` from the menu. This starts the Remote Dataset Wizard as shown in Figure 4. We must pick a class name for our OLE automation object. This is the name by which our OLE clients will identify the server they wish to connect to. Then we must pick the type of instancing we will use.

Server Class Instancing

The `Instancing` selection defines how the server object is instantiated as clients make connections. There are three choices: internal, single and multiple instancing.

Internal instancing means the interface to the COM object is not available outside the application server. This is how you would create an 'in-process' server since the COM interface is only available within the same process space. We won't say much about in-process servers here, as our interest lies in full-blown multi-process systems.

With single instancing, each time a new client connects to the OLE server, a completely separate instance of the server application is launched. So if three clients were connected to the same server object, there would be three instances of the server application running on the server machine. There is a single instance of the server object (the Remote Data Module) per instance of the server application. Note that you can still have any number of clients connected at the same time, they each simply get their own copy of the server application.

Multiple instancing is probably how you'll want to define most multi-tier servers. In this case, there is only one instance of the server application, but as each client connects, a separate instance of the server object (Remote Data Module) itself is created and managed by the server application.

The Remote Dataset Wizard automatically generates the type library files for the OLE object. In the same manner in which a Delphi unit's interface section describes the unit's Delphi objects so other units can use them, a type library describes COM objects in a way that allows other programs to use them. Type libraries are a Microsoft standard binary file that can be used by Delphi, C++, VB and other languages that support COM.

The Remote Dataset Wizard automatically generates the type library in a language neutral binary file with a .TLB extension. It also generates a Delphi implementation of the type library in a source file called <project>_TLB.PAS.

Data Access Within The Server

Within our Remote Data Module we can place all the standard dataset components (`TTable`, `TQuery`, and `TStoredProc`) to gain access to

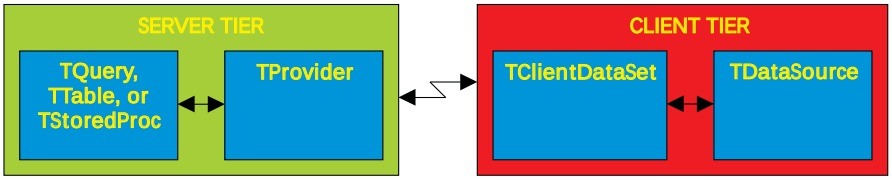
our data. We can organize these components in the same way we would with a standard Data Module in a standard client/server database app. So we simply add dataset components and event handling logic within the Remote Data Module, just as we would for any other database application. Each connecting client is guaranteed to have its own instance of the server's Remote Data Module, so there will never be a conflict with multiple clients accessing the same dataset, possibly even with different parameters. At runtime there will be a private copy of each dataset for each attached client.

Datasets within the server can be manipulated directly by client applications if we export an OLE interface for the dataset. MIDAS accommodates this through the `TProvider` class. For each dataset we wish to export, we must include a `TProvider` class as well. Datasets in a Remote Data Module can implicitly create their own providers and we simply export them by selecting `Export from data module` from the dataset component's context menu. We can also provide our own explicit `TProvider` component by dropping one from the component palette onto the Remote Data Module and connecting it to the dataset component. Again we would export this provider by selecting `Export from data module` from the `TProvider`'s context menu. Unless we export them in one of these two ways, datasets within the server are not visible to clients and only serve as internal working result sets for the server.

Once we have the basic shell of the server app in place, we can compile and run it. The server application must be run once to register it as an OLE Automation object for that machine. Otherwise, our client applications will not recognize our server. OLE Automation objects are registered under the `HKEY_CLASSES_ROOT\CLSID` key in the registry.

The Client Side

The client application in a multi-tiered system does not contain any `TDatabase`, `TTable`, `TQuery` or `TStoredProc` components to support the data accessed through the application server. These components reside in the application server itself. The client application uses a `TRemoteServer` component as a pipeline to the application server, in much the same way the `TDatabase` component served as a pipeline to a database on the database server. We set the `ServerName` property to the name of the application server we have created. If the application server resides on a different machine, we must identify that machine through the `ComputerName` property. We then set the `Connected` property to `True` to establish a live connection to the application server.



➤ *Figure 5*

Any datasets that the client wishes to manipulate from the application server are represented by `TClientDataSet` components. `TClientDataSets` are attached to the corresponding `TProvider` components in the application server (see Figure 5). The provider exports an OLE interface for the dataset component and sends server data to the client in data packets. On the client side, the `TClientDataSet` receives the data packets from the `TProvider` and constructs a local copy of the dataset. The client app can attach a standard `TDataSource` component to the `TClientDataSet` to allow data-aware controls to operate on the data. In short, all our `TQuery`, `TTable`, and `TStoredProc` components in the client application are replaced with `TClientDataSet` components and moved into the application server.

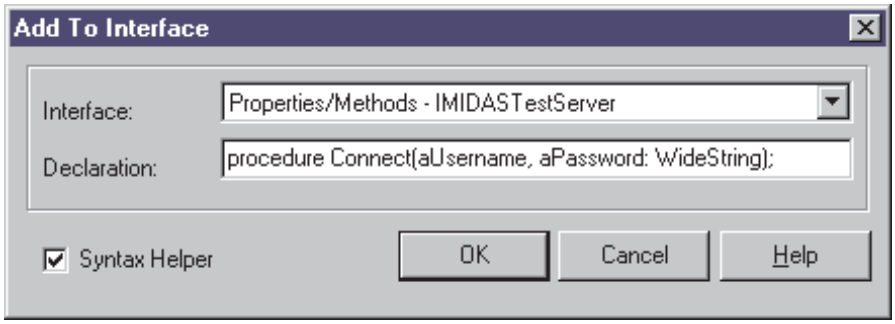
`TClientDataSets` are attached to the `TRemoteServer` through the `RemoteServer` property. This is analogous to how we might attach several dataset components to a `TDatabase` through the `DatabaseName` property in a traditional database application. Then we associate each `TClientDataSet` with a `TProvider` in the application server by setting the `ProviderName` property. Once everything is all hooked up, we have a functioning dataset component receiving its data from the application server.

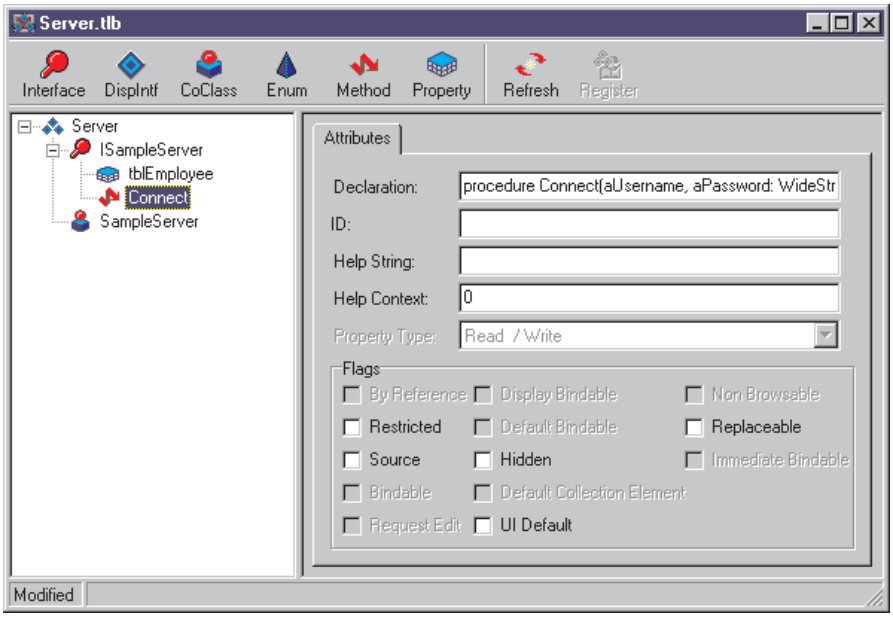
Accessing Server Methods

Datasets are not the only means by which we can pass data between application server and client application. Just as we might declare any number of ad-hoc methods in a standard Data Module to support the logic applied to the data, we can do the same in the Remote Data Module. In order for these methods to be accessible from a client, they must be interfaced.

When declaring an interfaced method, right click in the Remote Data Module source file and select `Add to Interface` from the context menu (or, alternatively, select `Add to Interface` from Delphi's `Edit` menu). This brings up a dialog (Figure 6) where we enter the complete declaration of the

➤ *Figure 6*





➤ *Figure 7*

method and its parameters. Clicking `OK` on this dialog causes the method to be added into the Remote Data Module source file as well as the Type Library files. At this point you can then proceed to add code to the declared method in the source file.

In the client application, the interface to the remote server is available to us in the `AppServer` property of the `TRemoteServer` component. So the method we just added could be called like this:

```
RemoteServer.AppServer.Connect(
  'AUser', 'APassword');
```

The datatypes of parameters and function results of interfaced methods must be OLE Automation compatible types. In general, this includes `SmallInt`, `Integer`, `Single`, `Double`, `Currency`, `TDateTime`, `WideString`, `IDispatch`, `WordBool`, `OleVariant`, `SCode`, `TColor`, `Byte`, `TSafeArray` and `IUnknown`. There are some additional possibilities outlined in Chapter 41 of the *Developer's Guide* manual shipped with Delphi.

You might notice that if we use parameterized queries as the basis of a dataset on the server, `TClientDataSet` lacks the properties that allow us to set the parameter values. In that case we would have to export a method from the server object to receive and set the desired parameter values from the client. The same is true for most direct manipulation of dataset component properties we might have grown accustomed to in traditional database development.

The Type Library Editor

Another way to flesh out the server object is to access the type library directly through the Type Library Editor (see Figure 7). In Delphi you can get to the Type Library Editor by selecting `View | Type Library` from the main menu, or with the <project>_TLB.PAS file in the editor window, pressing `F12`. This editor allows you to add interfaces, properties, methods and so on to the server object class. By clicking the `Refresh` button on the toolbar, the changes are posted in the .TLB file and the <project>_TLB.PAS file is regenerated to reflect the changes. One thing you should be careful of: when deleting existing properties or methods in the server object, the Type Library Editor is not always good about removing the generated code from the associated Delphi units.

Son Of Cached Updates

Client applications working with distributed datasets are always working with a local copy of the data. Data received from the provider is cached on the local machine. By default, the entire dataset is fetched from the provider at one time, but you can alter this behavior with the `TClientDataSet.FetchOnDemand` property. Client datasets operate like traditional datasets with `CachedUpdates` set to `True`. All modifications made by the user are cached locally until the dataset is committed by calling `ApplyUpdates`. Because of their nature, cached updates always carry along with them the possibility that some other user has modified the same records you have and applied their changes to the database before you did.

Keep in mind that we are not directly connected to the database. When we apply updates, we hand off all our changes to the application server, which in turn tries to apply them to the database server. Any records that cannot be committed successfully (because of constraint violations not already caught by the client, or conflicting data changes made by another user for example), are returned to the client as a dataset and the `TClientDataSet`'s `OnReconcileError` event handler is called for each one.

This handler gives the client application an opportunity to determine how to handle each conflict. Delphi ships with a prefabricated reconciliation dialog which you can add to your apps by selecting `File | New | Dialogs | Reconcile Error Dialog` from the Delphi main menu. This dialog demonstrates the full scope of actions that can be performed via the `OnReconcileError` event handler. Even if you don't use it as-is for conflict reconciliation in your application, studying its source code will give you a good foundation for handling conflicts.

The dialog displays the failed record, and for each field which conflicted shows the original value, the value we tried to apply, and the conflicting value already on the server. The dialog allows the user to correct the data by hand and then re-apply, skip the record, abort the reconciliation process, merge the record as-is into the existing record on the server, and a number of other possibilities.

Briefcase Applications

TClientDataSet's implementation of local caching makes it possible to build 'briefcase' applications, or offline datasets. In this scenario, an application would connect to the application server and retrieve data in a TClientDataSet. The application could then disconnect from the application server and the cached dataset is unaffected. The user can continue to operate on the cached copy of the data for as long as they want. Then they can reconnect to the application server and apply the updates, dealing with any reconciliation problems that may occur with conflicting updates made by other users.

TClientDataSet also provides a handy way to persistently store its cached data with the SaveToFile method. This method writes the cached copy of the dataset into a specially formatted file. The inverse method LoadFromFile allows us to prepopulate a TClientDataSet from a disk file without actually connecting to the application server. These features allow you to create applications that can work with data offline, and reconnect to the application server later to apply changes to the actual database.

For example, upon startup the client app would check for the existence of local data files on the hard disk. If it finds any, it loads them (TClientDataSet.LoadFromFile). The user can connect to the remote server at anytime (TRemoteServer.Connect := True). If we had not previously loaded local data, we would now fetch data from the server (TClientDataSet.Open). We can let the user apply their changes to the server data (TClientDataSet.ApplyChanges), or abandon those changes (TClientDataSet.CancelChanges). At any time the user can disconnect from the server (TRemoteServer.Connected = False), and continue to work on the data. When the client app shuts down, we can detect that changes have been made to the data (TClientDataSet.ChangeCount <> 0). If changes were made, we can save the dataset persistently (TClientDataSet.SaveToFile).

Database Components

Recall that with a multi-instancing class, there is only one instance of the server application, but distinct instances of the OLE Automation server defined in the Remote Data Module are created for each connecting client. Consider what would happen if all clients shared the same server object. Suppose our server contains a TQuery with a parameterized SQL statement of:

```
SELECT * FROM Employee WHERE
  EmpNo = :EmpNo
```

If two clients were simultaneously running this query using two different values for the employee number, then we have two completely different result sets for the query, *but only one* TQuery *component.* That is why each client gets its own instance of the server class: to keep its operating state independent of all other clients.

Now, with regard to BDE access to the database, all these instances of the OLE Automation server are sharing the same session of the BDE. If we use a TDatabase component in our Remote Data Module, we are going to run into problems. The TDatabase component registers its DatabaseName property with the BDE session. Once the second client connects to our server and opens another instance of the same TDatabase component, it will try to register the same database name with the BDE session. Since database names cannot duplicate within the same BDE session, we get a *Name not unique in this context* exception. Keep in mind that this is not an issue with single instancing servers because the BDE sessions are isolated by virtue of the fact that they reside in separate processes running on the machine.

One way to avoid this problem is to put an explicit TSession component in the Remote Data Module and set its AutoSessionName property to True. All the TDatabase and dataset components are associated with explicit session through the SessionName property. All the data access components in the OLE Automation server are now guaranteed to have an independent BDE session, by virtue of the explicit TSession component.

The problem of duplicating database names disappears because each database is isolated within its own session. The sessions themselves are guaranteed to have unique names because of the AutoSessionName property.

Conclusion

Multi-tier application development opens the door to a lot of interesting possibilities in database application development. Delphi provides a great deal of help to us in encapsulating the interactions of the COM objects and in handling distributed datasets via MIDAS. Splitting data access, or any kind of system logic, into a separate application requires more attention to such issues as object-oriented analysis and design and the distinction of user interface and business rule code. In single- or two-tiered systems, it is easier to be sloppy about the logical boundaries of the code. Multi-tier architectures are much less forgiving, and rightfully so.

Next month we'll learn how to use Microsoft SQL Server's own OLE Automation interface to make multi-tier database applications from a third-party interface.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be reached via email as Steve_Troxell@USGroup.com

Scott Samet is also a software engineer with Ultimate Software Group, as well as an active member of Team B.