

TClientDataSet, we hardly know ye?

By Kris Golko

In Kylix, the new portable database library, dbDirect, will implement only the provide-resolve model!

First impressions

When TClientDataSet was introduced in Delphi 3, I was as indifferent to it as anybody else. It seemed to be associated with the multi-tier world, which at the time was not on my horizon. Early experiments showed it rather likely to crash and requiring an expensive licence to use. Now, the more I know about TClientDataSet, the more I like it and the more new opportunities I see.

What's the ClientDataset?

I think of TClientDataSet as just an in-memory database table. It's inherited from TDataSet, so you can use all the data aware controls with it; it's an advanced and well-designed descendant of TDataSet. It wouldn't be of much use, though, if it didn't have a way to exchange its data with the world. One way to do this, is to use SaveToStream and LoadFromStream procedures to save and load data from a stream; a disk file can be accessed as a stream. To spare you a few lines of code, the SaveToFile and LoadFromFile procedures are also provided (SaveToFile opens a disk file as a stream, calls SaveToStream, and closes the stream). These procedures allow a TClientDataSet's data to be stored in a disk file. This is a fine solution for a very small single-user database.

For anything more advanced, data is exchanged using a provider component. The provider component has methods to exchange data with a database: TClientDataSet can thus be small and independent of a database flavour. A provider component can be hosted on another machine, so applications using TClientDataSet can be distributed.

Navigational versus provide-resolve model

In a navigational model, tables are available as a randomly accessible array of rows. You can scroll through rows and typically you select rows randomly, editing one row at a time. A navigational model is associated with pessimistic locking, the lock for a row or table must be obtained before any edits can be started. If a lock can't be obtained, an operation is aborted and the error reported. An example of the implementation of a navigational model in Delphi is TTable.

An alternative approach is illustrated by the following scenario. A number of rows are retrieved from the database, and then they are edited locally. There are changes to some rows and, as rows can be inserted and deleted, eventually, edited data is put back into the database. In the meantime other users can update their changes to the database and rows to be updated might keep different values or be deleted.

The process of updating data to the database with regard to changes made by other users is called **resolving**. In the process of **resolving**, decisions have to be taken, whether rows changed by someone else should be overridden, merged or ignored, etc. so it typically involves user interaction. The process of procuring data is called providing, that is why it's called the **provide-resolve** model.

Both models have their advantages and disadvantages, for example provide-resolve doesn't support pessimistic locking, but it's generally agreed that, in most situations, the provide-resolve model is better. This model goes particularly well with database servers (Oracle Database, MS SQL Server, InterBase, etc.).

How do they do it in Delphi?

Delphi 1 was designed with support for local tables (Paradox and dBase) in mind and supported only the navigational model. What about SQL-based database servers, which are based in a completely different paradigm? There's some inconspicuous trickery to make a SQL query result set to look like a table. This is subject to considerable criticism, as it doesn't work seamlessly.

To overcome these problems, Delphi 2 introduced cached updates, which actually are implementations of the provide-resolve model. The cached updates are designed to fit snugly into the SQL-based environment, but it works as well with Paradox/dBase.

Delphi 3 introduced MIDAS. MIDAS actually implements the provide-resolve model. It is based on two core components: TClientDataSet and TProvider. Since Delphi 3 cached updates were not intended to be substantially developed, further development was moved to MIDAS instead.

MIDAS 3 bundled with Delphi 5 is more than just an upgrade, it's a substantial redevelopment. The TProvider has been renamed to TDataSetProvider to reflect the idea that data can be provided to TClientDataSet from non-database sources as well.

In Kylix, the new portable database library, dbDirect, will implement only the provide-resolve model!

Mobile applications - briefcase model

Here comes an example of how flexible the TClientDataSet is. When TClientDataSet has retrieved data from the database, it can disconnect from the database and still be able to edit data (that sort of in-memory table, anyway). It can also save data to a file, so even if the computer gets shut down, data can be loaded from the file to the TClientDataSet, editing can continue and the data can eventually be reconciled with the database. These capabilities make easy development of applications based on the following scenario.

- user plugs into a corporate network with a notebook to access data from a database
- queries a database and saves data in a file on a local hard drive.
- disconnects from a network and departs with a notebook.

- at a remote site, loads the data from a file, edits, inserts and deletes records and saves data again.
- when he's back, he resolves the data to a database

This architecture serves well for applications used by travelling representatives.

What's not obvious is that you don't have to copy the entire database to the notebook. What needs to be saved to a disk file may be a result of a complex query, involving many tables and returning only a limited number of records. Data is still editable, and it's still possible to resolve changes to a database.

Web broker applications

Anybody trying to develop BDE-based Web broker applications would notice two disadvantages of the heavyweight BDE: it takes a lot memory and takes its time to load and unload. To serve multiple Web requests, the Web server has to create multiple copies of an application simultaneously, and it has to start and terminate them often.

One solution is to move to non-BDE databases, some of which have quite a small footprint. The other is to move database access and create "thin client" applications using TClientDataSets. Clients that are Web broker applications load faster and consume far less resources. Only one application server needs to be running and serves multiple clients, and it can be moved to another machine if further reduction of server machine overload is needed.

More reliability!

The danger of corrupting a Paradox table (this also applies to any other file system based database) is attributed to the fact that it is accessed through the local file system of a client machine typically from a network drive. Any problem on the client machine could end up with an incomplete operation on a table or index file. Introducing MIDAS actually shields table files from client computers. As an extra advantage, your application is ready to be converted to Client/Server or multi-tier and is more easily ported to Kylix.

Scaling from desktop to client/server

The problem of scaling Paradox based applications from desktop to Client/Server is a classic one. It looks simple: you migrate the database to InterBase or Oracle using DataPump; you connect your tables and queries to a new TDatabase component, recompile and expect everything will be just fine.

In reality it might look quite different. As I've said before, problems exist with data-aware controls and SQL based tables and, for someone new to the subject, it's a long and frustrating process. It's quite an effort to become familiar with the database server environment and management. It's much less hassle to create a database and populate it with data in Paradox than in InterBase, not to mention Oracle. If a team successfully overcomes all these problems, it still has a long way to go to optimise a database structure to take advantage of features not available in Paradox.

All that can be done in a step-by-step way. The first step is to replace existing local datasets with clientdatasets, and connect clientdatasets to a local dataset. When this task is completed and tested, local datasets can be replaced by datasets accessing data from a database server, and it's done.

Multi-tier applications

Last but not least - this is the primary purpose of creating the TClientDataSet. I've left it till the end, because it's the most complex case. A multi-tier application is a distributed application, i.e. parts of an application are hosted by different machines. MIDAS makes it possible, because TClientDataSet can use remote providers. The typical arrangement is that the database server constitutes one tier. The middle tier or application server hosts database access components, which are made accessible to the world by means of providers. The client application uses TClientDataSet connected to providers from the application server. To make a long story short, the more concurrent users there are, and the more overloaded the network, the more you need to distribute your applications.

Goodbye custom dataset, hello custom provider

If there is a need to access some unsupported database or other data source, an "old fashioned" solution is to create a custom dataset component. The new way is to create a custom provider. To create a custom dataset, you have to override 21 methods; to create a custom provider only six, so you have far less chance of messing things up. On the other hand, there is little use for a custom provider if you don't use TClientDataSet.

Developing a custom provider isn't easy, but developing a custom dataset requires a lot of knowledge as well: it's not an everyday task

Why ClientDataSet?

As you can see from the examples above, TClientDataSet provides a very flexible solution. It represents the current state of knowledge in database development, and I expect to see further progress here.

Final thoughts

With MIDAS 3, the concept reached maturity and there are many reasons to consider it in new development plans. In my snobbish opinion, technologies like dbExpress should be available for developers in the way that JDBC (Java Database Connectivity) and other Java technologies are available. I hope all your MIDAS applications turn to gold.



Kris Golko is an independent Delphi and JBuilder developer, based in Hampshire. He holds the title of Certified Delphi 5 Client/Server Suite Consultant.

You can reach him on krzysztofgolko@yahoo.com.