Speeding Up Client/Server

by David Selwood

So you've implemented your client/server application and you want more speed? Faster execution can always be achieved and various methods are presented in this article. The main objectives are to minimise network traffic and to make improved use of the VCL and the BDE. Example code is given and a client/server application which implements the discussion is on the disk. The example uses the `IBLOCAL` aliases given with Delphi and its `CUSTOMER` and `COUNTRY` tables with the `SYSDBA` username and the default password of `masterkey`. Some of the following will also apply to local transactions (ie Paradox and dBASE tables).

Faster Driving

The drivers you use for accessing a database and server determine IO speed. Always use the latest 32-bit drivers for both network protocol and database access. ODBC drivers are common, but they can be slow since they are not BDE native. The rule is that if you can access the database via a native BDE driver (ie SQL Links) then do so.

The `driver` and `aliases` pages should be reviewed with the BDE Configuration Utility. If the table structures are going to remain consistent then `ENABLE SCHEMA CACHE` should be `True` and a schema cache directory should be defined. This will improve BDE performance since it will not be constantly fetching schema information from the server. For each alias the `SCHEMA CACHE SIZE` should be set to the maximum number of tables your application will have open; this allows all tables to be cached. The `SCHEMA CACHE TIME` should be `-1` which means leave schema information in the cache until the database is closed. The `SQLQRYMODE` should be `NULL` which indicates that a query executes at the server first. Should the query fail it is attempted by the BDE, if the BDE fails an exception is raised. The `SQLPASSTHRO MODE` should be `NOT SHARED` to avoid clashing between pass-through SQL and Delphi's methods.

If you are using TCP/IP and the server and Delphi application are on the same machine then use the standard TCP/IP loopback address of 127.0.0.1 to communicate. It is also worth checking the TCP/IP configuration: it is often configured to send a packet when the packet is full on the IP layer; this can be reconfigured if this is the case.

Sharing Your Data Access

Delphi 2 and 3 allow Data Modules, which provide a mechanism for storing non-visual components in a single module which can be accessed by any unit/form. The sharing of data access components will minimise logging-on to databases and also opening and closing of tables. Data Modules also save coding time as units will not have to re-declare components. To access these components add to the `uses` clause within the `implementation` section the name of the data module unit. In Delphi 2 and 3 the components in the data module can be accessed by component drop down list box properties. If using Delphi 1 you provide this functionality by having all your non-visual data access components on a form or hard-coded in a separate unit. This will require hard coding in that when the unit is initialised the `datasource` property of the visual database components should be set up to point to the correct datasource component:

```
DBGrid1.DataSource:=
  MainForm.DataSource1;
```

If you encounter violation errors this may be because you are trying to access components before they are created. To resolve this select the menu item `Project|Options` and under `Auto-Create Forms` drag the unit which contains the non-visual components to the top of the list so that it is created first.

Get Connected

Setting the `TSession.KeepConnections` property to `true` will allow temporary `TDatabase` connections to remain connected even when inactive; this will save time in not having to re-establish connections. The `TDatabase` component has a similar property called `KeepConnection` and this should be set to `true` to avoid multiple logging off and back on when tables are closed and re-opened. Should an application not be expecting to use the server for some time then close the `TDatabase` connection as this will save resources on both the PC and network. This allows other users to access the server if connections are limited.

Transactions

Delphi assumes an *implicit* transaction environment which is similar to the desktop local database architecture. Put simply this means that when a record is posted, the record is committed to the database automatically. This is fine for local databases, but on servers it will result in heavy network traffic, since each record is committed separately. The committing of individual records and the required network IO and server processing for each record will slow your application down. The advantage is minimum record conflicts between users. Also, since this is the default transaction control the programmer does not perform any extra work.

The other method is *explicit* transaction control. For explicit control the application must start and commit or rollback transactions explicitly. This allows batch operations to be performed on records and results in less network traffic. However, explicit control can result in conflicts when the same records are being operated

on in a multi-user environment. Because of the saving in network traffic and the option of transaction rollback, this extra power is a worthwhile investment.

The example program allows implicit and explicit transactions to be applied. I suggest you have two copies of this program running at once and experiment performing updates on the same record.

Explicit Transaction Control

Three mechanisms are available for explicit transaction control. These are: firstly, use the `TDatabase` component to start, commit or rollback transactions, secondly use the `TQuery` component and thirdly the `CachedUpdates` property of `TTable`, `TQuery` or `TStoredProc`.

With `TDatabase` the table or query should be linked to the `TDatabase` component and the `StartTransaction` method should be executed. To commit changes to the database execute the `TDatabase` `Commit` method and to cancel use `Rollback`. Experiment by using the `StartTransaction`, `Commit` and `Rollback` buttons in the example program. It is important that if the `TDataSource` is in edit or insert state then this state should be set to `browse` by posting the edit or cancelling the edit; if this is not performed, expect strange results. Notice that code placed within the `Commit` and `Cancel` buttons events perform this: see Listing 1. If the user starts editing or inserting a record and the `StartTransaction` method on `TDatabase` was not performed then Delphi will work implicitly (ie the default method) and the current record will be posted automatically.

SQL Passthrough

The other implementation for explicit transaction control is pass through SQL via the `TQuery` component. This is when SQL is passed directly to a server for execution. Using this method one SQL script can contain many transactions and it allows the server to be used to its full potential as special server transaction commands can be utilised. The disadvantage of unique server commands is the application will be server dependent. The BDE has a `UniDirectional` property which if `false` allows the results of the `TQuery` (the dataset) to be moved forward and backwards. If the user or program code is only going to move forward set the `UniDirectional` property to true. This will save time and memory by avoiding caching. When performing queries where parameters are required don't build static queries (ie passing the SQL to be executed into the `TQuery` `SQL` property). Static queries require the SQL to be passed to the BDE and then to the server and you or Delphi will have to prepare the query. Instead use queries with parameter passing: dynamic queries. These queries only have to be passed and prepared once.

```
procedure TfrMainForm.buStartTransactionClick(Sender: TObject);
begin
  DataModule2.Database1.StartTransaction;
  buStartTransaction.Enabled:= False;
  buRollback.Enabled:= True;
  buCommit.Enabled:=  True;
end;
procedure TfrMainForm.buCommitClick(Sender: TObject);
begin
  if DataModule2.DataSource1.State in [dsEdit,dsInsert] then
    DataModule2.Table1.Post;
  DataModule2.Database1.Commit;
  buStartTransaction.Enabled:= True;
  buRollback.Enabled:= False;
  buCommit.Enabled:=  False;
end;
procedure TfrMainForm.buRollbackClick(Sender: TObject);
begin
  if DataModule2.DataSource1.State in [dsEdit,dsInsert] then
    DataModule2.Table1.Cancel;
  DataModule2.DataBase1.Rollback;
  buStartTransaction.Enabled:= True;
  buRollback.Enabled:= False;
  buCommit.Enabled:=  False;
end;
```

➤ *Listing 1: TDatabase transaction control*

Cached Updates

A `CachedUpdates` property is available on `TTable`, `TQuery` and `TStoredProc`. If this `boolean` property is `true` updates applied in the latter components will be cached.

To apply the cached updates use the `ApplyUpdates` method which sends the cached updates in one batch to the server. This minimises network traffic. Cached `Updates` should be used with implicit transactions (that is, when you are not using `TDatabase.StartTransaction` etc) and the programmer should use `ApplyUpdates` for groups of related fields.

Explicit transactions save on network I/O and cut down the number of intervals when a user is left waiting for a server update to be applied. This does cost something! The longer the time between a transaction starting and being committed to the database the greater chance that a lock may have been applied by another user or that the data has been altered. To reduce this a trade off should be made. I suggest that when data is updated frequently by different users, database updates are performed frequently to minimise contention. If data is not updated frequently then the updates to the database can be less frequent.

Updating Mode

When Delphi performs an update on a SQL database it achieves this by using the SQL command `UPDATE` with a `WHERE` clause to make sure the correct record is updated. The `WHERE` clause not only makes sure the correct record is updated, but if the record has changed between Delphi reading the original record and performing the update then the update will fail and raise an exception. Both `TTable` and `TQuery` have a `UpdateMode` property that states the type of SQL `WHERE` clause to be generated. The default value is `WhereAll` and this is the slowest, since Delphi uses all the columns to find the record being updated and could produce a SQL statement which is too long. The `WhereKeyOnly` value is by far the fastest as only the table key is used to find the correct record being

updated. However the problem with WhereKeyOnly is that if simultaneous updates are performed exceptions will not be raised unless the key was modified. This method should be used if only one transaction is allowed write access to the table. I recommend that WhereChanged is used as Delphi generates the WHERE clause for only key fields and any columns that have changed. This is faster than the default of WhereAll and still generates exceptions if the data in the columns to be updated is out of synchronisation.

Searching

Delphi 2 and 3 provide various methods for searching tables: FindKey, GotoKey and Locate. The former two are restricted in that they only search on key fields, whereas Locate can search on any type of column. Borland quote "If Locate is used you will see performance gains in your application as it uses the fastest possible method to locate matching records." If the search columns include an index then the index will be used for a much quicker search, if not then a BDE filter is applied. If your application uses FindKey or GotoKey then convert to Locate.

Threads

It is sometimes necessary for the client to perform separate queries on different tables and servers at the same time (not a heterogeneous join). If the queries were not executed simultaneously the user would wait for the first query to complete and then for the second query to complete. With simultaneous processing the user will not be left waiting as long, as both queries are processed at the same time on separate servers. To achieve this the TThread component should be used. The TThread component should also be used if you just want to execute a background query. TThread has one drawback in that VCL components are not thread-safe. To resolve this the Synchronize method should be used. Each thread must define its own TSession, TDatabase, TQuery and TDataSource to avoid conflict between threads. The TDataSource should be linked to a visual component (ie TDBGrid) once the query is complete and should be linked in through the primary thread using the Synchronize method.

In the example program if the menu option Two Threads is selected you will be able to execute two SQL threads in parallel (see Figure 1). Notice that the default SQL for these two threads is a cartesian query (ie all rows in one table multiplied by all rows in another table). This is to lengthen the processing of the query. Both the queries may be edited. The main program code for creating the threads is shown in Listing 2. This consists of a new class called TThreadedQuery which accepts the query to be executed and the TDBGrid the query output is to be linked to. The main methods in the class are Create, Execute and LinkDataSourceToTDBGrid. Only the code in the Execute method is performed in the secondary thread and it is here the query is executed. Listing 2 has further comments.

Using Your Server

A basic rule of thumb is: if the server can do the processing then let it. The more tasks that are stored on the server will reduce the amount of network traffic as the client and server will only be transmitting parameters and results. Also, if tasks are stored on the server they will already be prepared and optimised by the server. For example if you have a large or frequently used select statement make sure the statement is stored on the server and not the client.

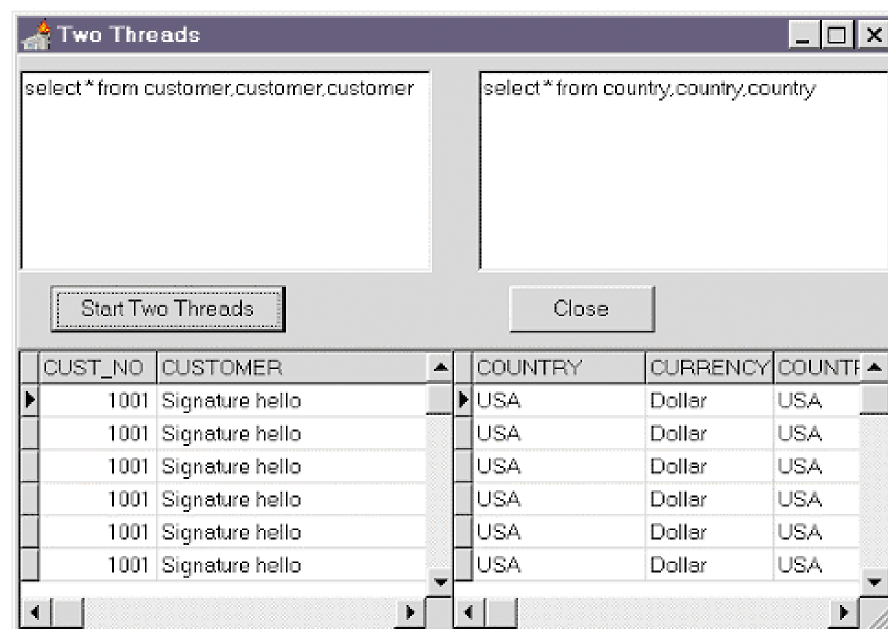
Another advantage with having tasks stored on the server is that the tasks can be made available to other users and systems. Also the tasks allow business rules to be stored with the data: an encapsulation if you like, with access restricted by security. The following mechanisms are provided by all back-end servers and should be considered: views, stored procedures and triggers.

Many SQL servers have added extensions to SQL and no standards exist between vendors on these extensions. If portability is a concern pay careful attention to control flow statements, server exceptions, etc as they are not standardised! Also of importance is locking: Oracle servers implement row locking and implement page locks when X numbers of rows in the current page are locked. Microsoft SQL Server locks the page which the row is on.

SQL

Most SQL servers will try to optimise a query, however with good

➤ *Figure 1*



SQL implementation you can further increase the optimisation. For example use the WHERE clause to reduce the number of rows. When using the OR clause always have the first condition least restrictive. When using LIKE avoid placing a % in the first character of the pattern match, as the server will not use an index, even if one is available. Try to avoid use of ORDER BY unless the result set needs to be sorted, as SQL is set oriented. When using GROUPBY if possible group to keys, to allow the server to work faster. Try to avoid the HAVING clause because HAVING is applied after the data is placed in groups. Try and compute the HAVING clause within the WHERE clause: it's much faster.

Sets

If your application is performing operations on sets of data then use SQL. This is because SQL is a set oriented language rather than record oriented. For example if you wish to delete customers who live in England this could be implemented via Delphi code or SQL (see Listing 3). The SQL implementation is far more efficient as the operation would be performed in one server transaction and most probably uses only one page lock. With the former implementation the data would be passed from the server to the BDE and the BDE would perform the operation on a row-by-row basis. Row-by-row processing should be avoided since each row is processed separately and this increases the number of transactions between the server and client and each row would be locked separately.

Indexes

When desiging tables go to at least Third Normal Form. This will mean you have indexes for all columns that will be used in relational joins. It is also advisable to have indexes for columns that are used by the SQL ORDER BY clause and the Delphi Locate or FindKey methods. When normalising and adding indexes keep the indexes to what is required. Foreign key constraints and indexes lead to slower performance because of the added maintenance. If the server provides clustered indexes these should be considered as they allow indexed sorted records to be physically next to each other. This will increase performance because when the clustered index is used the disk head will be ready to read the next record.

Views And Stored Procedures

Views or stored procedures give the advantages of the query not having to be passed to the server and the view or stored procedure will already be optimised and compiled by the server.

Views can be treated as live queries and if the view is going to be updated use the WITH CHECK OPTION

➤ *Listing 2: TDatabase transaction control*

```
unit UnitThreads;
interface
  uses SysUtils, Forms, Classes, DB, DBTables, DBGrids;
type
  TThreadedQuery = class(TThread)
  private
    FID:                    string;
    FGrid:                  TDBGrid;
    SessionThread:          TSession;
    DatabaseThread:         TDatabase;
    QueryThread:            TQuery;
    DataSourceThread:       TDataSource;
    FExecuteException:        Exception;
    procedure LinkDataSourceToTDBGrid;
    procedure ShowExecuteException;
  protected
    procedure Execute; override;
  public
    constructor Create (const ID, Alias, UserName,
      Password: string; QueryStatement: TStrings;
      Grid: TDBGrid);
      virtual;
    destructor Destroy; override;
  end;
implementation
uses
  UnitStartTwoThreads;
constructor TThreadedQuery.Create(const ID, Alias,
  UserName, Password: string; QueryStatement: TStrings;
  Grid: TDBGrid);
begin
  // Create Thread in a Suspended State
  inherited Create(True);
  // The rest of the Create processing is still using
  // the primary thread so VCL's can be accessed.
  FID:= ID;
  FGrid:= Grid;
  // Create object instances
  SessionThread:=   TSession.Create(nil);
  DatabaseThread:=  TDatabase.Create(nil);
  QueryThread:=     TQuery.Create(nil);
  DataSourceThread:= TDataSource.Create(nil);
  // Give the new Session object a unique name
  SessionThread.SessionName:= 'Session'+ID;
  with DatabaseThread do begin
    // Give the new Database object a unique name
    DatabaseName:=   'DatabaseName'+ID;
    // Link the Session object to the Database object
    SessionName:=    SessionThread.SessionName;
    AliasName:=      Alias;
    Params.Values['USER NAME']:=  UserName;
    Params.Values['PASSWORD'] :=  Password;
    LoginPrompt := False;
  end;
  with QueryThread do begin
    // Link the Database object to the Query object
    DatabaseName:= DatabaseThread.DatabaseName;
    // Line the Session object to the Query object
    SessionName:= SessionThread.SessionName;
    // Assign the actual SQL Query to the Query object
    SQL.Assign(QueryStatement);
  end;
  // Don't allow thread to terminate itself when complete
  FreeOnTerminate:= False;
  // Resume execution of Thread
  Resume;
end;
destructor TThreadedQuery.Destroy;
begin
  QueryThread.Close;
  DataSourceThread.Free;
  QueryThread.Free;
  DatabaseThread.Free;
  SessionThread.Free;
  inherited Destroy;
end;
procedure TThreadedQuery.Execute;
begin
  try
    // Execute the query
    QueryThread.Open;
    // Link the Query Object to the DataSource Object
    DataSourceThread.DataSet:= QueryThread;
    DataSourceThread.Enabled:= True;
    Synchronize (LinkDataSourceToTDBGrid);
  except
    // Save the current exception object
    FExecuteException:= ExceptObject as Exception;
    Synchronize (ShowExecuteException);
  end;
end;
procedure TThreadedQuery.LinkDataSourceToTDBGrid;
begin
  // Link the DataSource Object to the Forms DBGrid
  FGrid.DataSource:= DataSourceThread;
end;
procedure TThreadedQuery.ShowExecuteException;
begin
  // Show error message
  Application.ShowException(FExecuteException);
end;
end.
```

in the view statement to make sure that the updates satisfies the WHERE clause. Views are faster for updating than TQuerys. The reason for this is that views are already understood by the server, whereas, a TQuery modification will require the BDE to generate a new SQL statement which is based on the original query and then send the update to the server. This takes time.

There are, however, rules for acquiring updateable view tables and the rules are dependant on the server being used. If a view includes a join or uses an aggregate function then it will be read only. Note that if the TUpdateSQL component is used this will allow a read only dataset to be updated. When using parameterised components such as TStoredProc and TQuery you should use the Prepare method if the component is going to be used more than once. The Prepare method prepares the database for parsing and optimization and needs only be called once. If Prepare is not called then Delphi will implicit prepare each time the query is executed.

Listing 4 displays the stored procedure from the Delphi IBLOCAL database. The example program uses this procedure under the menu item Other Options | Stored Procedure. You will find that this procedure is prepared only once in the Data Module OnCreate event. If a query has been explicitly prepared and the query will remain unused for sometime then use the UnPrepare method to release database resources.

Delphi 2 and 3 provide filters on TTable, TQuery and TStoredProc that allow restrictive views on datasets. Filters are applied when the data has been received from the server and placed in the dataset, because of this you should only use filters on small datasets. To limit the size of the dataset use SQL to apply a filter. This will mean only filtered data from the server is transmitted to the client and will save much processing time.

Also note that the FindFirst and FindNext work in the same way as a filter. If possible use ranges instead

```
Delphi Code Implementation
With DataModule2.Table1 do begin
  DisableControls;
  First;
  While Not EOF do Begin
    If FieldByName('COUNTRY').AsString = 'ENGLAND'
      Then Delete
    Else Next;
  End;
  EnableControls;
End;
```

```
SQL Implementation
DELETE FROM CUSTOMER
WHERE COUNTRY = ENGLAND;
```

➤ *Listing 3: Delphi code versus SQL*

```
BEGIN
 SELECT SUM(budget), AVG(budget), MIN(budget), MAX(budget)
 FROM department
 WHERE head_dept = :head_dept
 INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
 SUSPEND;
END
```

➤ *Listing 4: Stored procedure*

of filters as ranges only retrieve rows from the table that are within the range, thus saving processing time. Ranges, however, can only be applied on keys.

Triggers

Triggers are statements that are processed when a table or view goes into a specific state, ie the change of state results in a trigger. The states are INSERT, UPDATE and DELETE. Triggers can be used to performed validation or to update other tables or views automatically. This functionality provide a means for implementing rules with the actual data in that the data will only get updated if the insert, update or delete validates correctly on the server. This removes the client from performing validation. An example of a trigger is in the Customer table where the trigger is CHECK (on_hold is NULL OR on_hold = '*'). This trigger simply checks that the column on_hold for the row being inserted or updated is null or is a *. To verify this run the program IBLOCAL and try updating the on-hold column to contain a value other than null or a *, you will find that an exception is raised if you try to post or move off the record.

Server Tuning

For optimum performance tuning the server is critical and this would be performed by the database administrator. The administrator will be concerned with improving overall performance of the database and will perform tasks such as changing the database page size, adding or removing indexes, fragmentation etc. When developing a client/server application make sure the database administrator is aware of what tables you will be using heavily so that the administrator can tune appropriately. For example if you are using an existing table and are expecting to frequently order the data in a specified column ask if the column can be defined as a secondary index. This will save server time on the SQL clauses using this column.

Other Options

Other options are available which are well known. For example your application can provide automatic login to the database by setting up the TDatabase params USER NAME and PASSWORD properties or by using the TDatabase OnLogin event handler.

Note that the TDatabase LoginPrompt property should also be set to false. Allowing automated login to databases presents a security risk and I recommend that if the user will be using a network supported by Microsoft Windows that the BDE API function DBIGetNetUserName is used. This function

returns the logged in user-name. Your application could then pass the returned username to the TDatabase user name parameter or compare the user name with valid know user-names before providing automated login. An example of this is given in Listing 5.

The field editor should be used to create persistent fields with dataset components. This will improve speed as Delphi will not have to implicitly acquire this information from the server, and an exception will be raised should the dataset configuration change. Also if persistent fields are created you can use TFields to access the fields in a record. Using TField is much faster than using the FieldByName method as FieldByName or the Fields property as they walk the table schema to acquire information. Whereas TField information is compiled within the application.

If performing operations on batches of data then the TBatchMove component may provide the required functionality. TBatchMove allows appending, updating, copying and deleting of batches of data and provides mechanisms for storing errors that occurr when the component is executed. If doing large batch operations it may be advantageous to drop the indexes, perform the batch operation and then re-create the original indexes. This will generally require exclusive use of the table. When the program is scrolling through a dataset use the DisableControls method to avoid constant screen repainting. To check for updates on a server use the Refresh method of TTable, TQuery, or TStoredProc, rather than closing and re-opening the dataset. Refresh is quicker. If a field is read only then use the read only component (ie DBText). Also the label component can be used if the data is not expected to change on the server. The same mechanism should also be consider with combo boxes and list boxes. If the table is read only, consider copying data to a local table or to a memory combobox. This will save much network traffic IO. Try to avoid fetching information that is not required. For example only fetch and display information that the user requires. Often the TDBGrid component is used and this requires lots of resources from the server, network and client. If TDBGrid is used, only include columns that are required; avoid fetching unwanted data.

```
implementation
  uses DBIProcs, DBIErrs, DBITypes;
{$R *.DFM}
function GetNetUserName: String;
begin
  SetLength(Result, dbiMaxUserNameLen + 1);
  // Get network user name and call Check to generate an exection if an error
  Check(DbiGetNetUserName(PChar(Result)));
  SetLength(Result, StrLen(PChar(Result)));
end;
procedure TDataModule2.Database1Login(Database: TDatabase;
  LoginParams: TStrings);
begin
  // Provide automatic login if know user
  if GetNetUserName='DAVID' then begin
  LoginParams.Values['USER NAME'] := 'SYSDBA';
  LoginParams.Values['PASSWORD'] := 'masterkey';
  end;
end;
```

➤ *Listing 5: Network user name and automated TDatabase login*

The BDE allows the programmer to create in-memory tables via DbiCreateInMemTable. These tables since they exist in ram are considerably fast. However they are only suitable for small tables and cannot be indexed or saved to disk. If you require fast access to small tables then in-memory tables will provide this requirement.

General Improvements

Other improvements which are applicable to general Delphi applications are:

- Enable compiler optimisation,
- Disable runtime checking (I advise against this),
- Don't auto-create forms,
- Use constants parameters in procedure/functions,
- Avoid variant type variables,
- Use threads for background processes.

These should give you some ideas to work with!

VCL

It is possible to get carried away making a client/server application run as fast as possible. For example, you could set up a stored procedure on the server to perform inserts of new customers in the customers table. Your application could get the required fields from standard TEdit fields and passed the fields to the stored procedure on the server. This technique is ill-advised as the amount of coding on both the client and server would be outweighed by the benefits. If the VCL provides the functionality, use it.

Conclusion

The ability to achieve faster execution of client server applications or other applications comes at a cost: more time spent on designing, implementing and testing the application.

Hopefully this article will save you time in design and implementation, now that you are aware of the client/server acceleration possibilities.

I do advise you not to go overboard on making an application run faster: always try to avoid coding if the rewards are going to be few. More coding means more maintenance.

Certain areas will always be beyond the developer's control, such as a slow network: all you can do is highlight the problem. Perhaps you could ask for bigger packet sizes, since bigger packets will contain more data and this will reduce the number of packets and acknowledgements required.

David Selwood works as a freelance contractor; you can reach him by email at dselwood@aol.com