Quick Start Guide to FlashFiler

Introduction

This guide is intended for new users to FlashFiler who are especially interested in converting an existing Delphi application using TTables (probably with a Paradox (PX) backend) to one with a FlashFiler (FF) Client/Server backend. Most of the material contained in this guide came about from my own experiences in doing just such a task. Those experiences draw upon a significant amount of material posted to the FF Newsgroup (NG) – an invaluable source of information, tips and essentially real-time help (but more of that later).

This document is designed to get you up and running as quickly as possible with FF via the technique of suck-and-see. The first section is about getting up to speed with the "simplest" FF application – a single executable. (This is called using the embedded server and is the FF way of compiling the database engine directly into an application resulting a single-exe for distribution. No DLL's, no server to install on-site and configure. Its even easier than Delphi and the BDE – just compile and go.)

The section following that will get you quickly up and running with the full multi-user C/S version of FF. It is not necessary to have read the section on the single-exe applications first. Instructions are given on how to set up a generic data module to enable connection to a remote server (remote server = you have ffServer.exe running separately to the main application, usually on a different PC on your network) and then a brief discussion on how to manage your project effectively within the IDE so that you do not fall foul of the dreaded "connection to server no longer valid" message.

The document then has a brief discussion of ffServer settings and the terminology used in FlashFiler. These bits come directly from postings to the NG by TP, TPX'ers and some of the more experienced FF users. Many thanks to everyone who participates in the NG – you have all helped to make this document possible.

Regards,
Geoff.

A Single Exe Application – The Embedded Server

FF is a true client server DBMS. However, one of its many features is that the engine can be compiled directly into a project so that your application can be distributed as a single exe without the need to install, configure, and maintain the ffServer DBMS backend.

Tests have shown that over a wide range of database operations and table structures that an FF application built this way (called an embedded server) is roughly 60% faster than a single exe equivalent PX application. This is actually quite amazing as you would not expect a Client/Server backend to outperform PX in single-exe mode. There are a couple of reasons for this:

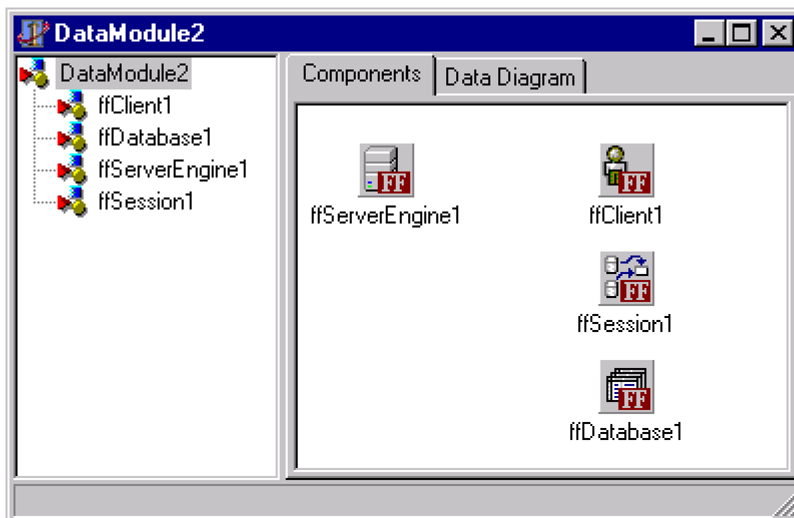
1. Client/Server architecture, being necessarily more robust and having more features than PX, carries an overhead for this extra table integrity.
2. PX has been around for a longer time than FF and thus it is reasonable to expect that it will show the benefit of being "tweaked" with all sorts of internal optimisations.

All things considered, if FF is faster than PX, and you have access to all the extra benefits of a Client/Server DBMS, then its not surprising that it is recommended you consider using FF as a total replacement for PX.

In this section we begin with a detailed step-by-step guide on setting up the data communications for a single-exe application. This data module is developed so that it is application independent. That is, once you have created the data module, you need only to copy the resulting files to a new directory and then create a new project and include the data module.

Setting up a FlashFiler Comms Data Module

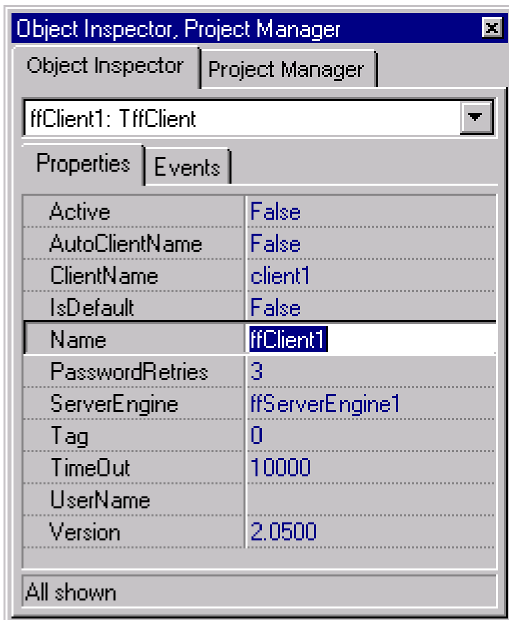
In the Delphi IDE you will need to create a new project. Now follow these steps to create the data module shown in the figure below:



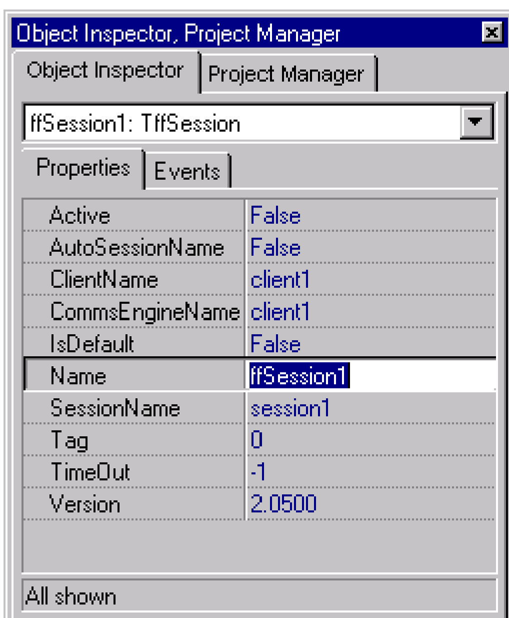
Creating the FlashFiler Embedded Server Data Module

1. Create a new Data Module in the IDE.

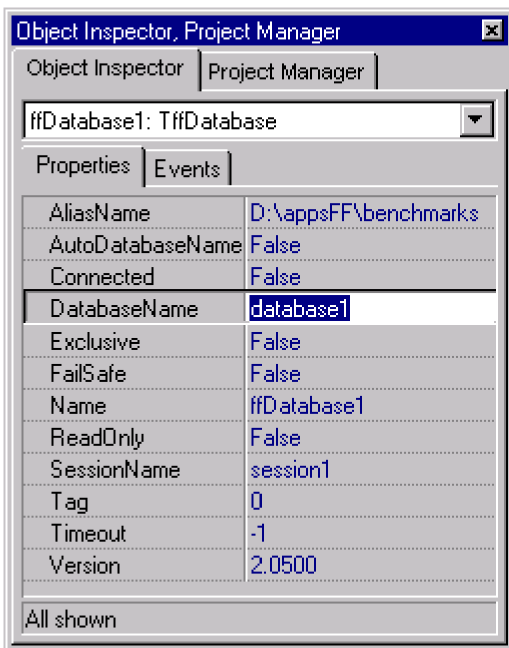
2. Drop on an embedded server engine – this is the first component on the FlashFiler Server tab of the VCL. Do not bother setting any properties.
3. Now add in turn client, session and database components (located on the FlashFiler Client tab of the VCL) to the data module as shown in the figure above.
4. Connect the ffClient1 component to the server engine (from the pull down list of the ServerEngine property) and name the client, eg., client1. The properties of the client should now look as follows:



5. Connect the ffSession1 component to your client component (pull down list of the ClientName property) and give it a name, eg., session1. The properties of your session should now look as follows:



6. Connect the ffDatabase1 component to the session component (from the pull down list) and give it a name, eg., database1. Finally, you will need to point the AliasName property to where your data tables reside, in this case, I've got mine sitting in D:\appsFF\benchmarks. Your database component should now look as follows:



You are now ready to go! Its that straightforward! The only thing that you may want to do to the data module is change the MaxRAM property setting of the server engine to a larger value (obviously depends on your application) but I find 10 Meg is rather conservative and usually up it to about 48 Meg. (Be aware, though, that if this exceeds the available memory on the end user PC, then the windows OS will automatically swap it out to disk with its virtual paging system – something I would recommend against!)

Making the single-exe application path independent

Finally, let's add one final run-time enhancement to our data module so that the resulting exe is not restricted the path coded into the AliasName property of the database component. That is, let's make the application automatically change this to point to the same path as the exe itself (here we are assuming that the data tables will reside in the same directory as the exe).

Setting the path on execution of the application

1. Double-click into the onCreate event handler of the data module itself.
2. Enter the following code

```
ffDatabase1.aliasName:=getCurrentDir
```

That's it. When the exe boots, the engine will set itself to current directory. If you want to keep your data tables in a sub-directory away from the exe, then you would modify the above to, for example, `extractFilePath(paramStr(0))+'\data'` or wherever.

Caution and final notes

Ok, seems simple to here, doesn't it? Well, it is. However, if you are not careful a few things may come back to bite you. Here's a list of cautions I have compiled over the months:

- Make sure you set the client component active property to false before compiling, testing or distributing to end users. Why? Because otherwise the database component will try to connect to the aliasName directory and it may not be the final one.
- Always open the data module as the first form in the IDE before doing anything else. Otherwise your table and querry components will have nothing to connect to and you will be wondering why the IDE has gone AWOL.
- Never, ever, EVER close the data module off whilst you are still working in the IDE – even if you want to test-run the app. Leave it open at all times.
- Ditto, make sure you close off all your forms, and the comms data module last, before exiting the IDE. If you are curious, just leave all your forms open and close down the IDE – then start up again and see what happens….
- Make sure in your project options that the comms data module is the first form auto-created. It (obviously) is not going to be the mainform, but it must be created before any other form (unless you want to do a lot of extra programming – which I'll let the masochists discover for themselves).

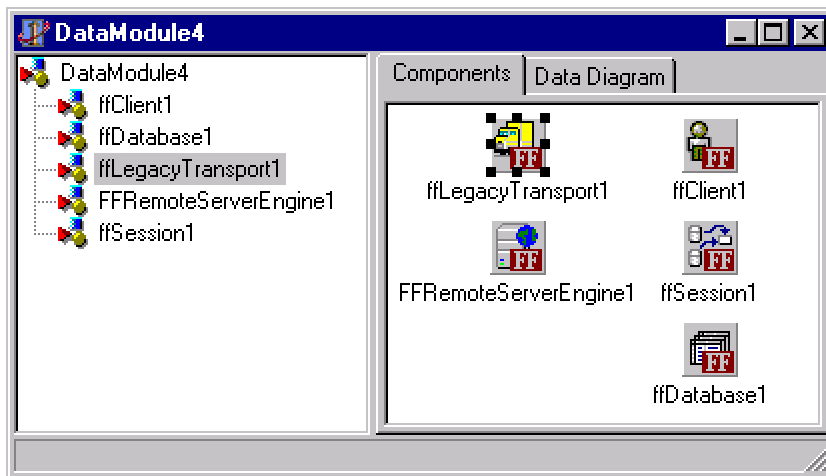
That's about all folks. As the old saying goes, have fun with it!

C/S Applications – Client application separate from the Server

In this section we set up a comms data module that you can use in any application that you are going to deploy as a true C/S system. That is, you will need to install and configure the FF server engine (ffServer.exe) on an appropriately designated file server. Your client applications must then connect to this server with an appropriate transport protocol. In this quick start guide I am only going to look at SUP (Single User Protocol) and TCP/IP (Transport Control Protocol/Internet Protocol) FF models. You use SUP wherever a client app is being run on the same workstation as ffServer and TCP/IP over a network (including the Internet). TCP/IP can be run on the same workstation as ffServer, but the speed advantages that come from using SUP are so significant that I strongly recommend you use SUP as a matter of course when able to.

Setting up a FlashFiler Comms Data Module

In the Delphi IDE you will need to create a new project. Now follow these steps to create the data module shown in the figure below:

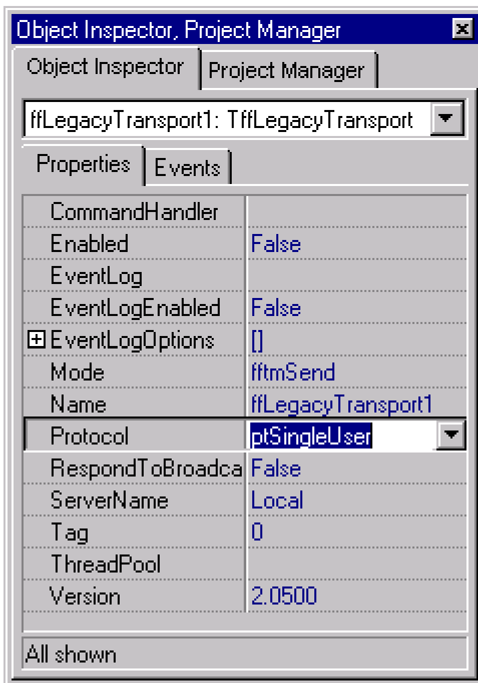


Creating the FlashFiler Embedded Server Data Module

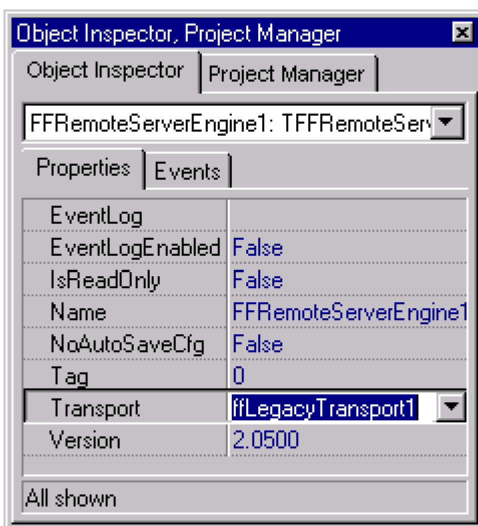
1. Boot up the FF server engine on your workstation (located in, depending on your install path chosen, XXX\FF2\BIN) and ensure that the SUP transport is started and an appropriate alias has been created.

2. Now create a new Data Module in a project in the IDE.

3. Place a legacyTransport component from the FlashFiler Server tab of the VCL onto the form. Set the transport protocol to ptSingleUser from the pull down list. The properties should look as follows:

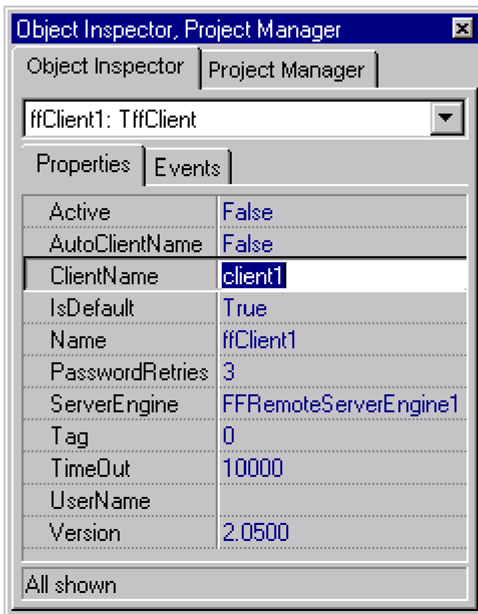


4. Grab a remoteServerEngine component, also on the FlashFiler Server tab of the VCL, and add it to the data module. Connect it to the legacyTransport component by selecting from the pull down list of the Transport property. The properties of your component should now look as follows:

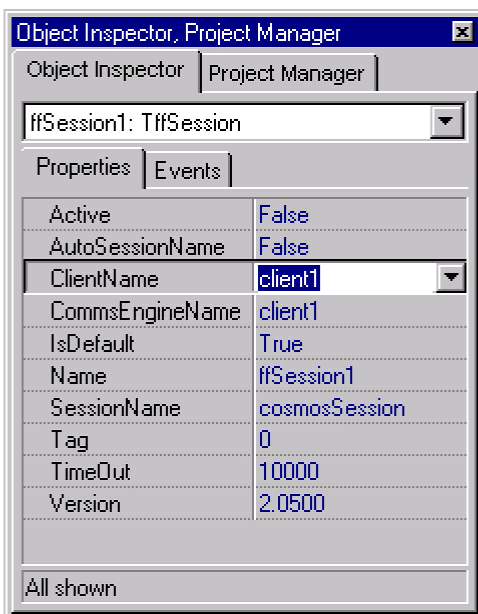


5. Now add in turn client, session and database components (located on the FlashFiler Client tab of the VCL) to the data module as shown in the first figure in this section above.

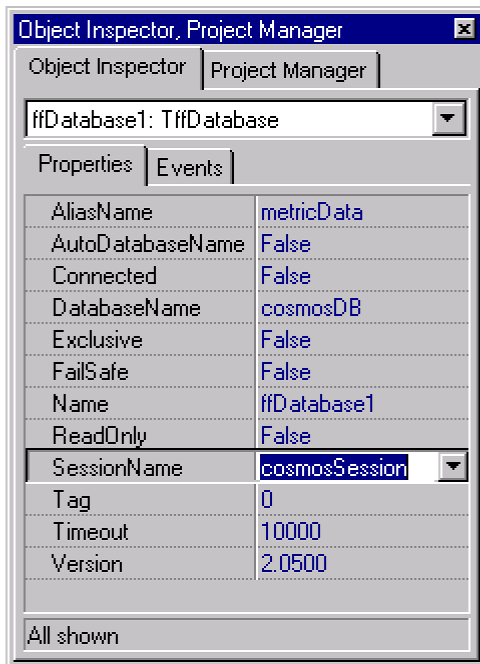
6. Connect the ffClient1 component to your remote server engine (from the pull down list of the ServerEngine property) and name the client, eg., client1. You should have set 2 properties only. The properties of the client should now look as follows:



7. Connect the ffSession1 component to your client component (pull down list of the ClientName property) and give it a name, eg., cosmosSession (to stand out). Whatever you do, do **not** leave the ClientName property blank or set to [Automatic]. The properties of your session should now look as follows:



8. Now comes the connections that seem to cause the most bother for programmers new to FlashFiler. Connect the ffDatabase1 component to the cosmosSession component (from the pull down list) and give it a name, eg., cosmosDB (it is important that you do **not** accidentally use an alias name here). Finally, you will need to select a name from the pull down list of the AliasName property which will connect you to an alias that you have already established in ffServer. (To the curious – for my performance benchmark tests, I used an alias called “metricData”.) Your database component should now look as follows:



You are now ready to begin! Although more involved then setting up the comms for single exe applications, its not really difficult. The server engine can now be configured external to your client application (in the control front screen of ffServer).

Making the client executable independent of protocol setting

As with the single exe application, let's add a run-time enhancement to our data module so that the resulting client app is not restricted the protocol set when compiled. That is, let's make the application select which server to connect to on bootup.

Setting the FF server on execution of the application

1. Double-click into the onCreate event handler of the data module itself.
2. You will need to now read the preferred protocol. My first go at this resulted in the following (inefficient!) code. I am in the process of changing this to more "standard" Windows INI calls (and have suggestions for using exception calls), but for the moment, just enter the following code (or your own once you have looked at this!)

```
var  f:textFile;
    st,pc:string;
    connectedOK:boolean;
begin
    randomize;
    connectedOK:=false; {assume NOT  able to connect to the flashfiler server}
    if fileExists('cosmos.ini') then
    begin
        assignFile(f,'cosmos.ini');
        reset(f);
        try
          if tag=1 then
          begin
              st:='cmaFFremote';
```

```
          pc:='ptSingleUser'
        end else
        begin
          readln(f,st);
          readln(f,pc)
        end;
        with ffLegacyTransport1 do
        begin
          if enabled then enabled:=false ;
          connectedOK:=true;
          if pc='ptTCPIP' then protocol:=ptTCPIP else
          if pc='ptSingleUser' then protocol:=ptSingleUser else
          begin
            showMessage('Invalid protocol settin g.  Value in ini file='+pc);
            connectedOK:=false
          end;
          serverName:=st;
          enabled:=true
        end;
        if connectedOK then
        try
          connectedOK: =false;
          ffClient1.active:=true;
          try
            ffSession1.active:=true;
            try
              ffDatabase1.connected:=true;
              connectedOK:=true
            except
              showMessage('Could not connect the Database to the remote FF server.')
            end;
          except
            showMessage('Could not activate the Session component to the Client.')
          end;
        except
          showMessage ('Could not activate the Client to the remote server engine.')
        end;
      except
        showMessage ('Could not enable the Legacy Transport.  Server name='+st);
      end;
      closeFil e(f)
  end else
  begin
      try
        with ffLegacyTransport1 do
        begin
          if enabled then enabled:=false;
          protocol:=ptRegistry;
          enabled:=true
        end;
        connected OK:=true
      except
        showMessage('Could not find cosmos.ini nor a registry entry  - PC needs fixing!');
        connectedOK:=false
      end
  end;
  if not connectedOK then
  begin
      application.terminate
  end
```

That's it. Oh, you will also need to add FFLLPROT to your uses clause. A typical ini file is setup as follows:

To connect to a remote server using TCP/IP it will look like

```
myServer@192.168.0.1
ptTCPIP
```

or, to connect to a server on the same workstation as ffServer using SUP:

```
cmaFFremote
ptSingleUser
```

Caution and final notes

Pretty much the same set of notes applies here as with the embedded server.

- Always boot up ffServer and bring up the server (usually set to automatic) before booting up Delphi.
- Make sure you set the client component active property to false before compiling, testing or distributing to end users. Why? Because otherwise the database component will try to connect to the aliasName directory and it may not be the final one.
- Always open the data module as the first form in the IDE before doing anything else. Otherwise your table and querry components will have nothing to connect to and you will be wondering why the IDE has gone AWOL.
- Never, ever, EVER close the data module off whilst you are still working in the IDE – even if you want to test-run the app. Leave it open at all times.
- Ditto, make sure you close off all your forms, and the comms data module last, before exiting the IDE. If you are curious, just leave all your forms open and close down the IDE – then start up again and see what happens….
- Make sure in your project options that the comms data module is the first form auto-created. It (obviously) is not going to be the mainform, but it must be created before any other form.

FlashFiler Terminology

It helps to become familiar with the FF terminology to not only save confusion but also to help for when you are posting messages to the news group for the support you will undoubtedly need. This is meant to supplement, not replace, the terminology given in chapter 2 of the FF manual.

Term	Explanation
Protocol	This refers to the communication mechanism used to transmit data between ffServer and the client applications. There are three such protocols available and are each described separately below.
SUP	Single User Protocol. This is not to be confused with a single exe (common mistake). This simply means that the server and the client app are both on the same physical machine. There can be as many clients apps booted on this machine making it a true client server setup. It is also a neat way to run for example a specialized client app on the server perhaps doing different tasks from the other network clients.
TCP/IP	This is the standard internet protocol which allows any client app to talk to the server provided they are on the same network. This is the transport of first choice when connecting to remote servers. Broadcasting only works on the same subnet. Connecting to known IPs should always work
Embedded Server	This is where the ffServer engine has been compiled directly into the client application. The advantage of this is that you not only get a true single exe app with no DLL distribution headaches but there is no need to use any form of windows messaging to talk to the server. Consequently embedded server applications are incredibly fast. They also have the advantage of the true client server explicate transaction wrappers. These features mean that these embedded server applications are actually a better solution than Delphi connected to any other backend for single exe application development.
ffServer	This is the ffServer executable that you must have running for all your applications except when you are developing an embedded server app. During development you must remember to always boot ffServer prior opening the Delphi IDE. When you distribute your application this is the server engine you should distribute and copy to the appropriate file server.
ffE	This is the FF explorer utility. It is the equivalent of Delphi’s database desktop. It allows you to maintain aliases and their associated database tables. Table repair is also usually performed with this utility.
ffEXP	This is a third party utility developed by Otto and can be

	downloaded from the third party NG from the TurboPower site. It is intended to be a cutdown version of ffE (described above) which you can freely distribute to your end users so they can perform some limited database maintenance functions. I highly recommend it as the tool to give to your end users – consider it the database desktop you would normally distribute with your Paradox application.
Connection Manager	This is one heck of a neat component written by Eivind as a replacement for the legacy transport component. It automates the connection to an ffServer across up to four protocols. In principle you need only have a unique name for your server and this component will do the rest for you. I have used it on DHCP NT LAN's and it works like a charm. It is available from the FF foundry reachable from Yahoo. A must for when you have a serious application with a multitude of user connections. When used correctly (that is, as intended by Eivind!) you will not need the onCreate event handler logic I put into the multi user section above. This should handle it all for you! (Ben from TP is currently evaluating it for use in a TP product: ProActivate!)

ffServer

The ffServer has several statistics on its main form that are useful for understanding FF performance:

Configuring and testing ffServer

The first step is to configure the FlashFiler Server. Boot up ffServer from C:\FF2\BIN, or wherever you installed FF.

1. Select the Config-Network menu item from the ffServer main menu. The Network Configuration window displays.
2. Verify that the TCP/IP and IPX/SPX transported are enabled and that the "Listen for Broadcasts" option is checked (if that is what you want). Don't change any of the port numbers. Click the OK button when done
3. Select the Config-Aliases menu item from the ffServer main menu. The Alias Configuration window displays.
4. Verify that your aliases are defined correctly. If you specified a directly that does not exist then its field in the grid will display with a red background colour. Click OK when done.
5. Verify that the ffServer is started. In the transport list in the lower half of the window, the transport status should be "Started". A "Failed" or "Driver not installed" status indicates some other kind of problem.

The second step is to verify you can connect to the ffServer using FlashFiler Explorer on the same machine. Start FlashFiler Explorer and switch between the TCP/IP and IPX/SPX transports using Options – Transport menu items in FlashFiler Explorer. This will verify you can at least see the ffServer from the same machine.

The third step is to verify you can connect to the ffServer from a different workstation via FlashFiler Explorer. When FlashFiler Explorer starts, it sends out a broadcast for available servers. Both the TCP/IP or IPX/SPX transports should show the ffServer on the other machine. Be sure to try out both transports in FlashFiler Explorer.

Messages

This counter lets you see how many messages have been sent to the server, and messages are usually the best place to optimize for performance. By resetting the counters and performing a specific operation on your client you can see the exact number of messages being created. If you want to see exactly what is being messaged back and forth you can turn on debug login and review what the server is being asked to do (warning: debug login really shows things down, it should only be used for debugging or profiling). Watching message counts is useful because it builds awareness of what your design is actually doing in a C/S perspective. Message counts also give you a quantitative measure of your process efficiency.

Messages/Second

This is useful for looking at long-winded processes where an exact message count isn't as useful as a "throughput" measure. Essentially you want to maximize the messages/second. Network latency can have a huge impact on this value. Consider a ping time of 50msecs between the client and the server, this means that no matter what you do you can't push more than 50 messages per second throughout your network. When you can predict these barriers you can make wise choices, such as batch operations to retrieve or insert a large number of records in a single message.

AV. Time Message

This is a great indicator of how hard the server is working. By reporting the average number of milliseconds each client request takes it gives an excellent measure of "work" the server is doing. Difficult and complex operations will cause this number to rise. An interesting relationship also exists between this number and the Messages/Second above. The product of theses two values is the number of milliseconds in each second that the ffServer is busy, giving a great tool to determine the average load on the server. If your server load is low (ie less than 100 or 10%)

Garbage Collection

When garbage collection runs, it does the following (see TffServerEngine.seCollectGarbage in unit FFSRENG):

- Removes unused server-side objects (eg, clients, sessions). These are server-side objects that couldn't be closed at the time they were told to close or they are tables that no cursors are currently using.
- Tell the SQL engine to perform garbage collection. Removes server-side SQL engine objects that are no longer needed.
- Flushes the lock container pool, semaphore pool, and buffer manager memory pools every 5 minutes regardless of how often garbage collection runs.

Running garbage collection every 30 seconds shouldn't be a big deal. If you have a system with a very large number of users than anyone opening a table or database or trying to establish a connection may notice a slight pause when garbage collection runs, if the two events occur at the same time. The reason for this is that when removing the unused server-side objects. It only locks one list at a time. If the server is sent a request to create a new server-side object (ie, open a table) or free a server-side object (ie, close a database) then the request must wait until the relevant data structure/s are available.

KeepAlives control this:

LASTMSGINTVAL=x
ALIVEINTERVAL=y
ALIVERETRIES=z

The time before the client is killed is:

x+y*z (milli-sec)

Transactions in FlashFiler

There are two very different types of database updates. An "explicit" transaction is initiated with a call to StartTransaction followed by whatever Edits, Inserts and Posts are required and ends with a call to Commit or Rollback. An "implicit" transaction is automatically performed by the server when Post is called after an Edit or Insert which has not been preceded by a call to StartTransaction. All transactions should be kept as short as possible and explicit transactions should never be wrapped around user input.

Implicit Transactions

Implicit transactions use record level locking. An individual record is locked when Edit is called and this lock is released in the Post. During the Post the server will start a transaction, post the record to the table, then commit the transaction. The whole table will be locked for the duration of this implicit transaction but this will be very brief because it only lasts for the amount of time it takes to post a single record.

FF uses pessimistic locking. Edit obtains an exclusive-write lock on the record. If an Edit is attempted on the locked record error 10241 occurs.

10241 = $2801 = ERRCAT_LOCKCONFLICT: ERRCODE_LOCKED

The exception does not tell you which table is involved.

The server retires the second lock request for up to TffTable.Timeout milliseconds. If the lock is not granted within that time the exception is raised.

The TDataSet class imposes a limit of one record lock per cursor. If you need multiple record locks you can use direct calls to the (Remote) ServerEngine methods. Only write locks are supported at the record level.

Explicit Transactions

Explicit transactions use table level locking. Explicit transactions are used when several records holding related information are being updated together and will ensure they cannot be left in an inconsistent state. An explicit transaction is initiated by a call to StartTransaction. Subsequent Edits and Inserts will obtain exclusive-write locks on the relevant tables until the transaction ends by a call to Commit or Rollback.

Commit has to obtain totally-exclusive locks on the tables it wants to update to prevent any other access to them. A transaction can only obtain a totally-exclusive lock on a table when all share-read and exclusive-write locks have been released by other transactions using the table.

Only the transaction performing the updates will see its alterations before Commit has been called.

A transaction can obtain a share-read lock on an exclusive-write locked table but only one transaction can have the exclusive-write lock required to perform Edit, Insert or Delete. Concurrent transaction are therefore only possible if the transactions are writing to different tables.

A transaction can stop another transaction committing by acquiring a share-read or exclusive-write lock on a table it wants to update. In this deadlock one or both of the transactions will time out.

Undocumented methods TffTable.LockTable and TffTable.UnlockTable add and remove a table level lock.

There is no server request that returns locking information to the client.

If repeatable reads are required (ie after a dataset reads record A it will always read the same value) perform the reads inside an explicit transaction because this will prevent any other transactions updating the table.

A transaction is confined to a directory and managed on a directory basis (an alias points to a directory and a TffDataBase, which is the owner of a transaction, uses a single alias).

Explicit FailSafe Transactions

An explicit transaction has its own "server cache" in memory on the server. This cache is only visible to this transaction. All updated records are written to the server cache. Pages of the cache which have been changed are flagged "dirty" and when the transaction is committed will be written to the tables on disk. If the power fails when some, but not all, of the dirty pages have been written to disk the database will be left in an inconsistent state.

To prevent this possibility set TffDataBase.FailSafe to true. As the transaction proceeds clean "before images" of the cache pages which are about to be updated are written to a Transaction Journal File (TJF). When a transaction is committed all the dirty pages are written as "after images" to the TJF and the completed file is closed and flushed to disk. The dirty pages are then written to the database tables on disk and when complete the TJF is deleted. If the power is lost the TJF can be used later to automatically complete the transaction.

For a Rollback the dirty cache pages are discarded and the TJF is deleted.

The NewsGroup

Most of the above comes from postings to the FF Newsgroup (NG) by the developers of FF and expert users, the so-called TPX'ers. You are well advised to read and participate in this NG. It is the forum for asking questions about FF as well as (usually) the first place where TurboPower (TP) announce new upgrades and bug fixes. The users of this NG have developed a unique camaraderie. You will come to appreciate the personalities of most of the users and may even look forward to adding your own threads.