Under Construction: Pooling And Brokering

by Bob Swart

We'll complete our MIDAS 3 coverage by considering Object Pooling (of remote data modules) and Object Brokering (of connections), two very interesting techniques in MIDAS 3. We'll see what these concepts mean, how we can use them, and we'll implement a custom Object Broker.

Object Pooling

To avoid possible disappointments later, I should say that Object Pooling is a feature which is only available when using HTTP connections; that is, using the `TWebConnection` component and not using the normal `TDCOMConnection` component, for example, for which MTS will support just-in-time activation and deactivation, database handle pooling and more.

Whenever we design a new Remote Data Module using the *New Remote Data Module* wizard from the Object Repository, we have the choice of creating it as a single or multiple instance object. If we pick single instance, the remote data module will be used for a single client request. Additional client requests will be served in new instances of the MIDAS server executable. This means that, for every client request, a new instance of the MIDAS server executable is started, where the remote data module is exclusive for the client request. If we pick multiple instance, we only get one executable but a new instance of the remote data module for every client request. Both cases can result in severe server strain when hundreds of concurrent connections are being made (or at least when the attempt is made to make hundreds of concurrent connections). Apart from the memory usage, you also duplicate database connections for every instance, which will slowly bring your server to its knees. The alternative, having a single remote data module and hundreds of client requests all having to wait in line to be serviced, might be even worse: your server will remain up, but your clients may get the feeling that you're letting them down!

Object Pooling gives you the ability to set a maximum for the number of instances of the remote data module inside the MIDAS server application. Whenever a client request is received, the MIDAS server checks to see if a free remote data module exists in the pool. If not, it creates a remote data module instance (but never more than the specified maximum number of instances), or raises an exception with message 'Server too busy'. The remote data module, in turn, services the client requests and duly waits for the next one. After a certain period of time without client requests, the remote data module is freed automatically by the object pooling mechanism.

In previous versions of MIDAS this feature would not have been possible, since we now have instances of a remote data module that service more than one client. As a result, the server cannot rely on state information, this has to be maintained by the client. And, as you've seen in Issue 57, MIDAS 3 is indeed stateless. As a consequence, you have to use the `OnBeforeGetRecords` events as explained in Issue 57 to send state information back and forth between the client and the MIDAS server.

The big question should now be: how do we enable Object Pooling for HTTP connections? Well, for the answer we must get inside the `UpdateRegistry` method. Using Delphi 5 (ie MIDAS 3) this method is automatically created for every new remote data module, so you only have to get inside this routine and add a line to call `RegisterPooled` when registering the server and call `UnregisterPooled` when unregistering the server. `RegisterPooled` takes three arguments. The first one is easy, that's the `ClassID` which is already passed as an argument to the `UpdateRegistry` method. The second argument specifies the maximum number of instances. Obviously, you should pass a positive value here. If a client request is received by the MIDAS server and no remote data modules are available, then an exception with message 'Server too busy' is raised. The third argument specifies the number of minutes the remote data module can wait idle in the pool of remote data modules. After spending the specified amount of time without any client requests, the remote data module will be freed automatically by the MIDAS server. According to the documentation, the MIDAS server checks every 6 minutes to see if any remote data module should be freed. Specifying a timeout value of 0 means the remote data module will never timeout, so in that case the only useful feature is the limit on the amount of remote data module instances.

In practice, there's a fourth argument you can pass to the `RegisterPooled` method: a default method that specifies whether or not the remote data module should be a singleton. The default is `False` and results in the situation I've just described. If you set it to `True`, the number of instances and timeout arguments will be ignored and only a single remote data module (which must be free-threaded) will be created to handle all client requests.

The modified `UpdateRegistry` for an example `TPoolingMidasServer` remote data module, with up to 10

instances that timeout after 42 minutes of inactivity, is shown in Listing 1.

Note that I've hardcoded the numbers 10 and 42 here. This is not a good idea in real life, especially since it means that you need to recompile the MIDAS server whenever you want to make some changes (for example, if you added new memory to the server so it can handle more than 10 instances). And that's not even considering the fact that the same MIDAS server could be placed on multiple machines, each with a different configuration. I always recommend using an external configuration file where you can specify, for each machine and for every time you first start the MIDAS server, the number of instances and time-out minutes. This adds flexibility to the power already present in Object Pooling.

Brokering Connections

Apart from Object Pooling, a technique to share and reuse remote data modules among multiple client requests, there's also a way to actually share multiple MIDAS servers (the applications holding the remote data module instances). This is one level higher. Why is this beneficial? Well, for example, if you do not have a single server machine that is able to handle the strain of all the concurrent client connections and you are forced to balance the load over more than one machine, each running a similar MIDAS server. Another reason is fail-over: if one server machine goes down, others will remain, so you won't go out of business just because of one failure. Note that this is different to letting the end-user know that a mirror exists, because in that case the end-user has to move to the mirror website if the original is down, whilst connection brokering should automatically connect new requests to a secondary server if the primary server goes down. Of course, if someone is already connected to a server and the machine goes down, then that's the end of that session and the end-user should start again.

OK, so let's assume that having multiple physical server machines (all running the same MIDAS server) is a good thing. How would the client application determine which MIDAS server to connect to? The client would need to know exactly which servers are available (or he might miss one, maybe the last one that's available at that time). Fortunately, MIDAS 3 offers a helpful hand in this case with, you guessed it, *Object Brokering*.

When using Object Brokering, the client can make a connection to a MIDAS server without knowing which MIDAS server it will end up with. Apart from the `TCorbaConnection` component, which has its own CORBA-based brokering mechanism, each of the other three connection components in Delphi 5 Enterprise (`TDCOMConnection`, `TSocketConnection` and `TWebConnection`) has a property called `ObjectBroker`. This property is used to connect to a component derived from `TCustomObjectBroker`. This `ObjectBroker` component will be responsible for telling the connection component which MIDAS server to connect to, by specifying the `ServerName` or `ServerGUID` information. Note that when you assign an `ObjectBroker` component to the `ObjectBroker` property of a connection component, you will override the values of `ServerName` and `ServerGUID` that may have been assigned to this connection component previously (because now the `ObjectBroker` will dynamically provide a `ServerName` and `ServerGUID` for you).

➤ *Listing 1*

```
class procedure TPoolingMidasServer.UpdateRegistry(Register: Boolean;
  const ClassID, ProgID: string);
begin
  if Register then begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
    RegisterPooled(ClassID,10,42); // max. 10 instances, time-out = 42 minutes
  end else begin
    UnregisterPooled(ClassID);
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

TSimpleObjectBroker

As an example of how to implement your own Object Brokering techniques, Delphi 5 Enterprise comes with a `TSimpleObjectBroker` component (already installed and available on the MIDAS tab of the Component Palette). This `TSimpleObjectBroker` component has two interesting properties. The `Servers` property should be filled (usually at design-time) with a list of available MIDAS servers. For each of these servers, you need to specify the `ComputerName`, the `Port` (211 by default) and whether or not the server is initially enabled. As the developer, you must make sure yourself that this list is filled with initial values and also maintained properly. When a new server becomes available you should put it in the list of available servers. Note that you do not have to disable servers manually, as the Object Broker will do that automatically (more on that shortly).

The second important property of the `TSimpleObjectBroker` component is `LoadBalanced`. As the name indicates, when set to `True` this will ensure that the servers are load balanced, or at least that the requests (for initial connections) are balanced among the servers. The technique used here (don't laugh) is based on a random number generator. When `LoadBalanced` is set to `True`, each connection component will be connected to a random server from the list available in the `Servers` property. When `LoadBalanced` is set to `False` each connection component on the client application will be connected to the first server that's still available on the servers list. Note that if you decide to use the `TSimpleObjectBroker`, the `LoadBalanced` property is set to `False` by default (ie the first MIDAS server for the list is selected for all connections), perhaps not such a good value to work with.

Note that if you only use one connection component on your remote data module, then you may want to make sure that each client application will indeed select a different MIDAS server from the list of servers (and not simply the first one that's available). And in that case, using a random MIDAS server is only effective if the `Randomize` function is called, in order to initialise the random sequence with a unique starting point. Without a call to `Randomize`, the `TSimpleObjectBroker` would have returned the same 'random' number for the (only) connection component.

TCustomObjectBroker

If you ever need to write your own Object Broker algorithm, then `TSimpleObjectBroker` might be a good place to start. However, although the `TSimpleObjectBroker` example provided by Borland should be the first thing to look at, you also may want to derive your own Object Broker implementation from the `TCustomObjectBroker` abstract base class (available in unit `mconnect`). This class has four virtual abstract methods that we need to override, so let's start a new component (`File | New`, select the `Component` icon) and create a new component called `TDMObjectBroker` derived from `TCustomObjectBroker`. Inside this component, we must override and implement the four methods shown in Listing 2 (that will be called by the connection component).

The `GetComputerForGUID` method will be called by the connection component (passing the `ServerGUID` value) in order to get the name of a server (machine) that implements the given `GUID`, an interface that should be implemented by the MIDAS server that runs on that machine. The `GetComputerForProgID` does the same but, instead of using a `GUID`, it will be passed the actual class name of the server, ie the `ServerName` property of the connection component. These two methods, `GetComputerForGUID` and `GetComputerForProgID` are the methods that have to implement a semi-intelligent solution in order to determine which MIDAS server (machine) to select.

`GetPortForComputer` is given the `ComputerName` (either the actual `ComputerName` or an IP address) as an argument and expects the `Port` as the return value. This can be considered additional information: the selection for a MIDAS server machine has already been made.

Finally, the procedure `SetConnectStatus` is called by the connection component to indicate whether or not a connection to that particular `ComputerName` has been a success. Thus, if the MIDAS server machine happens to be down, or temporarily unavailable, it can be removed from the list of available servers for a certain time.

Each of these four methods needs access to a list of MIDAS servers, and each list item should at least store the `ComputerName`, the `Port` and whether or not the server is actually available. You can either implement this yourself, or take a look at the implementation of the `TSimpleObjectBroker`, which has a `Servers` property of type `TServerCollection` defined and implemented in the ObjBrkr.pas unit. In fact, when you compare the `TCustomObjectBroker` abstract base class, with four virtual abstract methods, and the `TSimpleObjectBroker` example implementations, the latter has implemented all the support you need for these four virtual abstract methods (that will be called by the connection component). So, maybe starting all the way from scratch by taking `TCustomObjectBroker` is only a good idea if you really want to reinvent the wheel by yourself. If you want to reuse as much as possible (the `ServerCollection`), then it's good to know that the actual server selection algorithm, which is based upon a random generator for the `TSimpleObjectBroker`, is in fact implemented by a single function `GetNextComputer` from the `TSimpleObjectBroker`. The implementation of this function is shown in Listing 3.

If you want to implement your own Object Brokering algorithm and not reinvent the wheel with regard to the list of available servers (the `Servers` property of the component), then you quickly get into some problems. The `GetNextComputer` isn't a virtual method, so you cannot override it. And neither can the `GetBalancedName` or `GetNextName` methods from the `Servers` property. This is a bit of a disappointment. Although the `TSimpleObjectBroker` was provided as a good example of how to implement a (simple) object brokering example, it is not built to be extended. Rather, I guess, it is made available to have parts (the `Servers` property and `TServerCollection` for example) copied and modified. And after you've copied and pasted that code inside your own `TDMObjectBroker`, then the method that actually does the 'smart' work of selecting a new MIDAS server machine is implemented in the `GetBalancedName` method of the `ServerCollection`.

GetBalancedName

As an alternative algorithm, you could cycle through the list of available MIDAS server machines. This means that the `Servers` collection should somehow maintain state (which of the servers was being used to connect to last time?). One place to store this state and to include some server specific information is inside the class `TServerItem`, derived from `TCollectionItem`, that defines the MIDAS server information in the

```
function GetComputerForGUID(GUID: TGUID): string;
function GetComputerForProgID(const ProgID): string;
function GetPortForComputer(const ComputerName: string): Integer;
procedure SetConnectStatus(ComputerName: string; Success: Boolean);
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
function TSimpleObjectBroker.GetNextComputer: string;
begin
  if LoadBalanced then Result := Servers.GetBalancedName
  else Result := Servers.GetNextName;
end;
```

list of servers. You could add a property `LastConnectionTimeStamp` to it (of type `TDateTime`), that contains the `Now` value at the time of connection, and hence automatically increases for every new connection is made, or at least everytime the `GetBalancedName` returns with the particular MIDAS server.

Using this additional information, the `GetBalancedName` routine should first filter only the (still) available MIDAS server machines from the list of servers, and then sort them by the `LastConnectionTimeStamp` property. This should ensure the distribution is a bit more fair and balanced, although perhaps a bit less random.

Note that right before we return the selected `ComputerName`, we must not forget to set the new value of the `LastConnectionTimeStamp`. And also note that if a server hasn't had a connection made yet, then its connection timestamp will be 0, and hence it will be right up in the list of potential servers to select.

Even more complex Object Brokering algorithms could contain more information about each individual MIDAS server (machine), such as the number of connections (load) this machine is capable of handling, so a fair distribution (load balance) is attempted by the client application. Of course, like I said before, this is most effective only when using multiple connection components inside each MIDAS client, otherwise we must use other means to ensure that each individual client application will connect to a different server (and in that case the random approach, or a combination of the random approach and the time stamp approach, will work better).

Reconnection

One final word on Object Brokering: once a connection component is connected to a MIDAS server, it will remain connected to that particular server until the `Connected` property is set to `False` again. When the connection component has made a connection with the MIDAS server that has been selected by the Object Broker component, it saves the relevant values of the connecting properties like `ComputerName`, `Address` and `URL`. If the connection component closes and later reopens the connection, it first tries to use these property values to reconnect and only requests a new server from the Object Broker if the connection fails (ie when the first MIDAS server seems to be unavailable).

This also opens up a great way for connection error recovery: if the client ever gets an error (RPC server unavailable for example), then simply toggling the `Connected` property from the connection component will disconnect from the remote MIDAS server (which might be down or unavailable) and connect to another MIDAS server (provided at least one is still up). This is a great way to provide recovery of client activities! Of course, all session information might be lost, but at least the client can restart and continue to work even if one or more servers are going down.

```
function TServerCollection.GetBalancedName: string;
var
  OldestTimeStamp: TDateTime;
  OldestServer, GoodCount, i: Integer;
  GoodServers: array of TServerItem;
begin
  GoodCount := 0;
  OldestTimeStamp := Now;
  OldestServer := 0;
  SetLength(GoodServers, Count);
  for i:=0 to Pred(Count) do begin
    if (not Items[i].HasFailed) and (Items[i].Enabled) then begin
      GoodServers[GoodCount] := Items[i];
      if GoodServers[GoodCount].LastConnectionTimeStamp < OldestTimeStamp
        then begin
        OldestServer := GoodCount;
        OldestTimeStamp := GoodServers[GoodCount].LastConnectionTimeStamp
      end;
      Inc(GoodCount)
    end
  end;
  if GoodCount = 0 then
    raise EBrokerException.CreateRes(@SNoServers);
  GoodServers[OldestServer].LastConnectionTimeStamp := Now;
  Result := GoodServers[OldestServer].ComputerName;
end;
```

➤ *Listing 4*

Summary

We've seen that Object Pooling can be used to share and re-use remote data modules within MIDAS servers, whereas Object Brokering can be used to distribute MIDAS servers among multiple machines. Both techniques will help you to increase the potential of your MIDAS server application. Object Pooling is a matter of configuration, where you might want to store the max number of instances and timeout values in an external configuration file (so the MIDAS server application can be the same for many machines and only the configuration file contains the specific details for that machine). Object Brokering, on the other hand, can be much more involved, and may require you to implement a specific brokering algorithm for the application at hand. Combined, they extend the reach of MIDAS servers tremendously.

Next Time

Next month we'll examine strings. From the old short strings to long strings, AnsiStrings, WideStrings, WideChars, PChars, conversion routines, efficiency and more. And believe me, even if you *think* you know strings, you'll really get to know strings *the hard way*. Especially with Delphi 5. *So stay tuned...*

Bob Swart (aka Dr.Bob, www.drbob42.com) is an @-Consultant for TAS Advanced Technologies and a freelance technical author.