Load balancing and failover
For kbmMW v. 2.50+
ProPlus and Enterprise Editions

Introduction

It’s highly encouraged to read this document in its entirety.

Load balancing and failover are both two important areas which are covered by kbmMW v. 1.00 and later. kbmMW v. 2.50+ contains some additional advanced features like learning and teaching capabilities between load balancers.

In many situations, it’s not possible or extremely expensive to upgrade a machine to something bigger to cope with increased load. Often it would be better if another relatively inexpensive machine could be added which would then participate in serving client requests, and thus share the load with the other machines. This is called load balancing.

There are several ways to do load balancing, some of them listed here.

Centralized load balancing

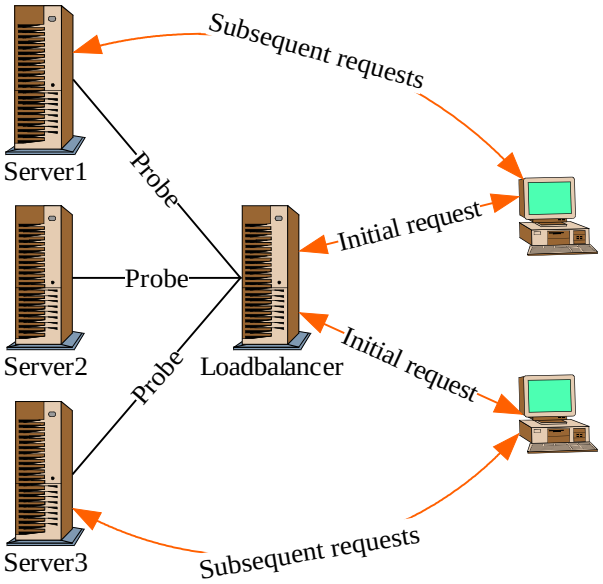


Fig1. Centralized load balancing

Centralized load balancing (Fig1) works from the concept that the clients initially ask a central server for information about which server actually to communicate with later on.
The load is balanced on a client basis, and not on a request basis since the client is being assigned to a specific application server.
The load balancer server keeps probing the application servers it knows about to check their load and availability and use this information to direct the clients to one of the application servers. Rules for the redirection of clients are explained a bit later in this document.
It's a good idea to have at least two load balancer servers running for fail over situations. Let the client's know only about these two, and enable load balancer teaching/learning functionality on the load balancers to let them learn from each other. There is a chapter about teaching/learning later on.

Distributed load balancing

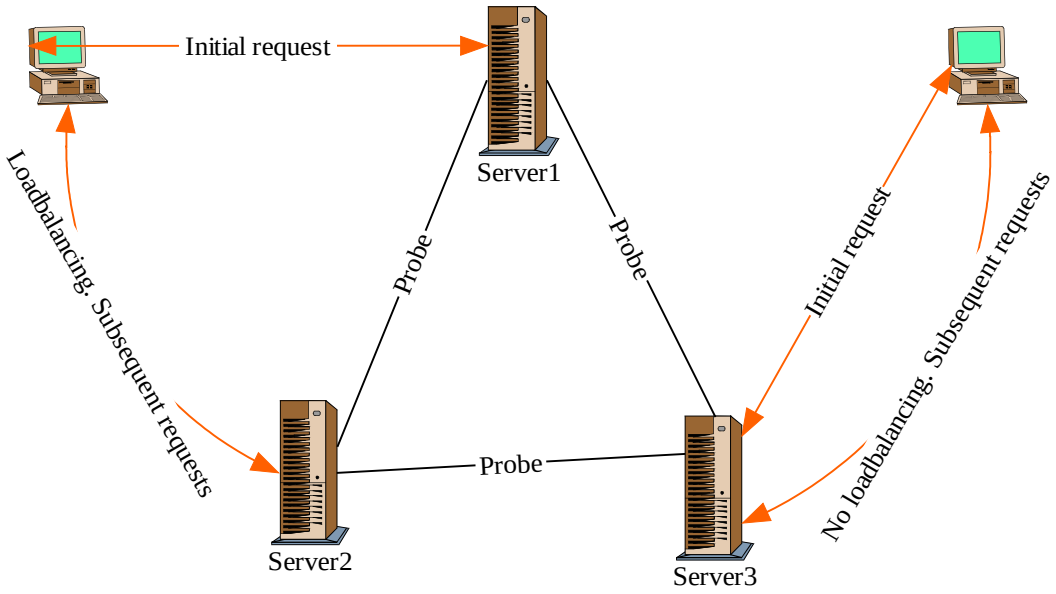


Fig2. Distributed load balancing

Distributed load balancing (Fig2) means that each server is able to redirect requests to other servers better suited for handling the request. This way the client just needs to connect to any of the servers, and only if the service requested is not available, or the server is too heavily loaded, the client will be redirected to another server. Every server keeps track of every other server including its load and availability.
Let the client's know about a couple of the server's, and enable load balancer teaching/learning functionality on all servers to let them learn from each other. There is a chapter about teaching/learning later on.

Fail over

Fail over means that a client automatically will try to reconnect to known alternative servers if a server do not respond on a request. Again there are several types of fail over techniques like client controlled fail over and server controlled fail over.

Client controlled fail over

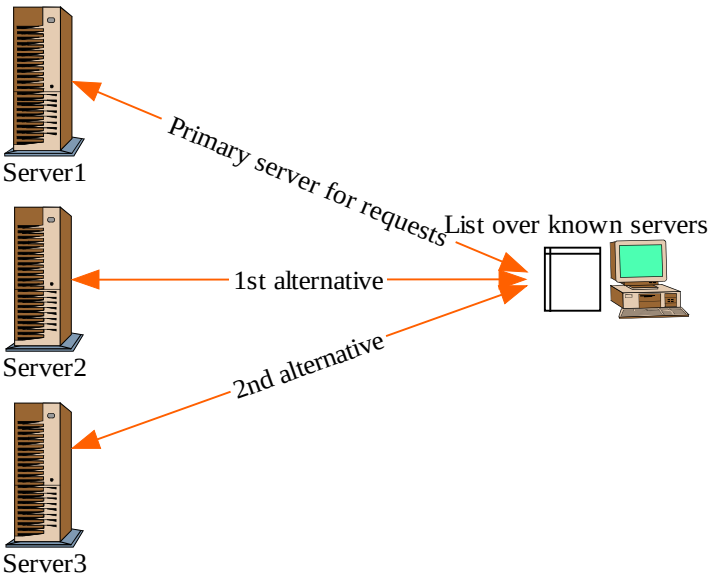


Fig3. Client controlled fail over

The client controlled failover works by letting the client automatically choose another server for its requests the moment connections are lost or servers do not respond on requests. That means that the client needs to know connection properties for a selection or all the servers. This type of fail over could also be used for simple load balancing if the client randomly selects a server from the list of servers.

Combined server and client controlled fail over

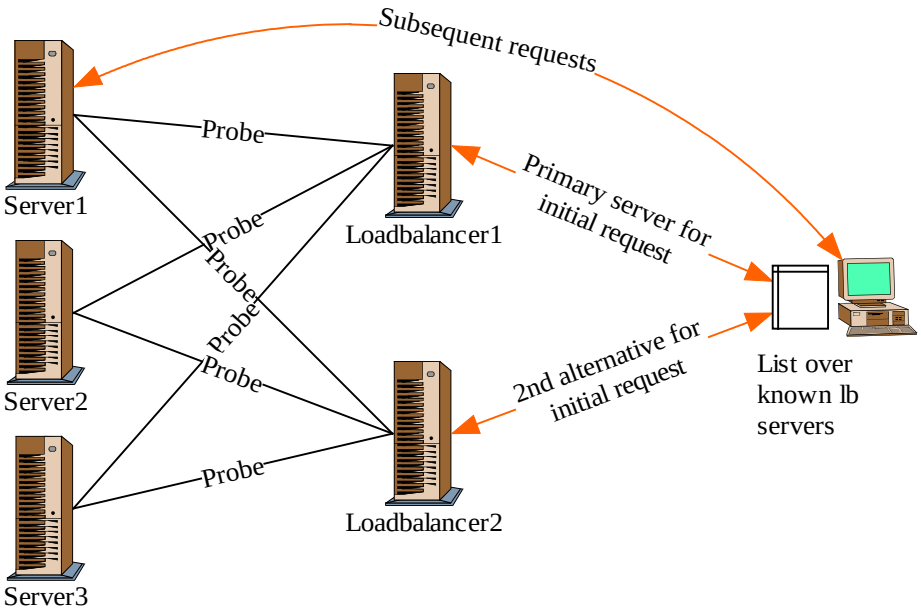


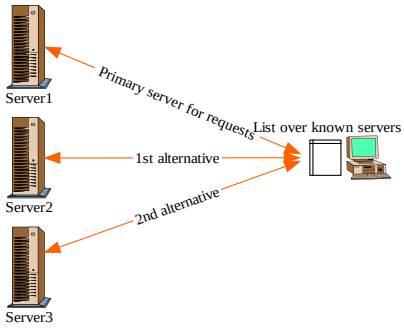
Fig4. Combined server and client controlled fail over

The combined server and client controlled fail over is actually a hybrid of load balancing and fail over. The client will only need to know connection properties for the load balancer servers. The load balancer servers will contain all knowledge about the application servers that have been assigned for use by this cluster.
The client controls fail over for the connections to the load balancer. If a connection fails, the client will automatically try a reconnect to one of the load balancers which will in turn redirect the client to an application server which is known to be running and have room for more work.

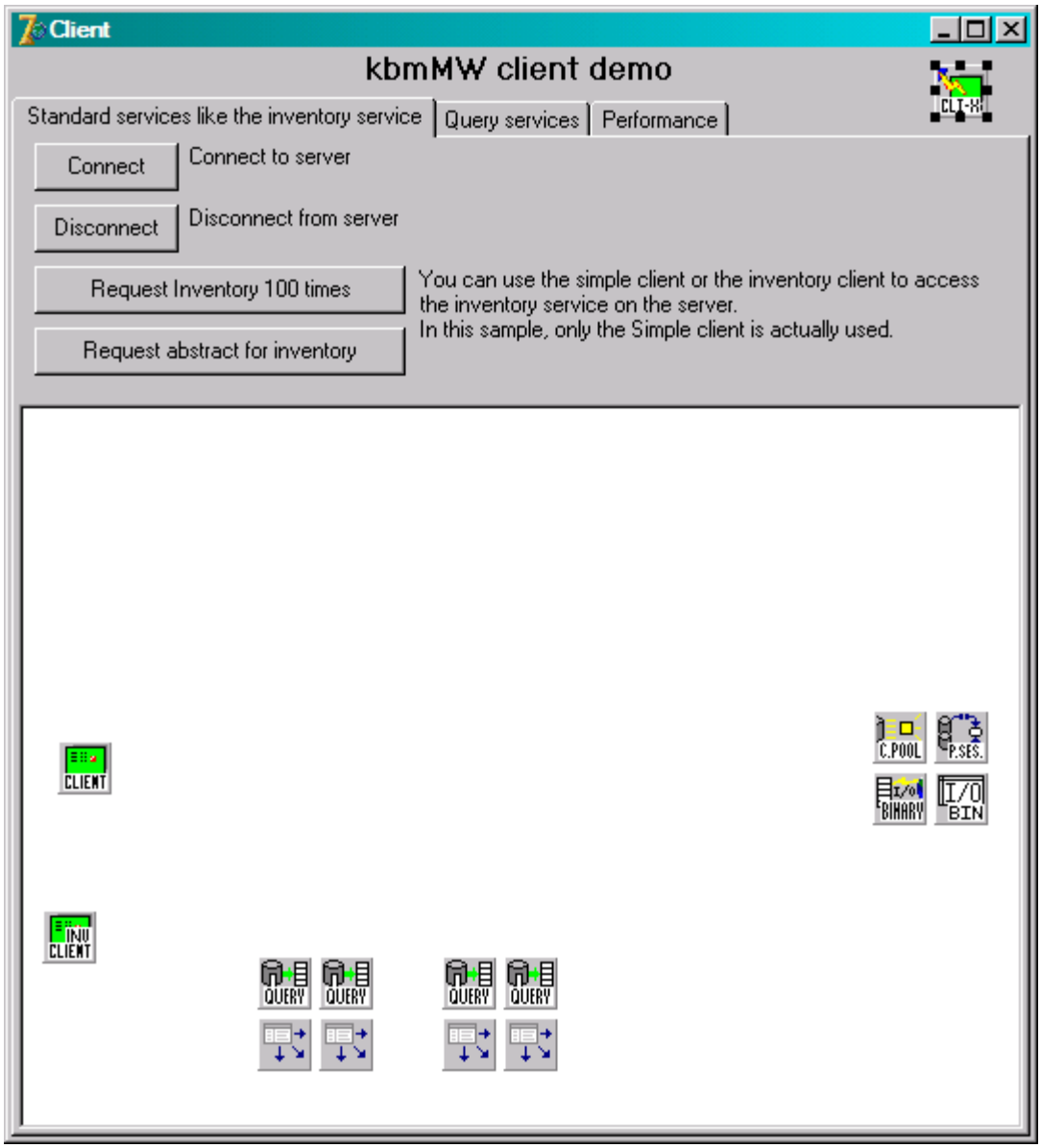
Putting it all to work

On the following pages you'll find a step by step description of what to do to implement the different techniques in client and server.
As a base for the client and server, the demo client and server projects are used. For basic information about how to create a client and a server please refer to other whitepapers.
Delphi 7 is used for all examples, but Delphi 5 and 6 should be able to follow same pattern directly. Borland C++ Builder users will have to adapt the description to the C++ environment.

Client controlled failover and simple loadbalancing

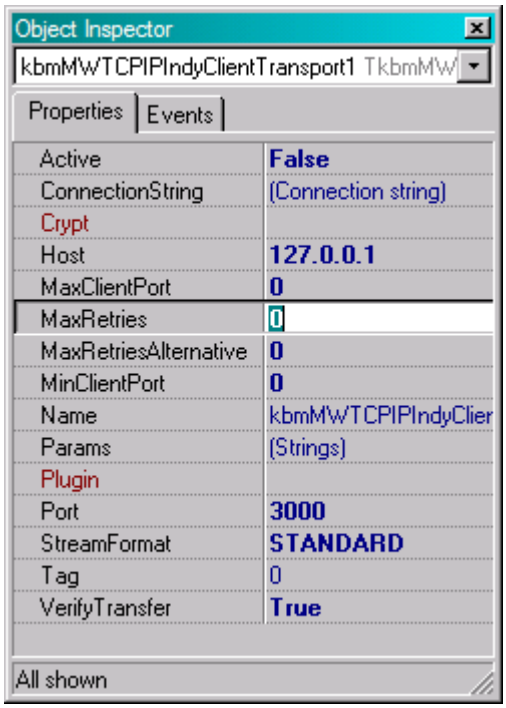


This only requires change to clients since they are doing all the work deciding which server to connect to.
This how modifies the demo client project while using the demo server project as is.



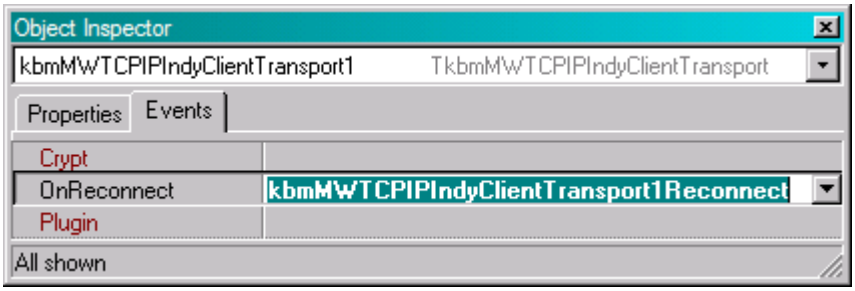
Open the main form and select the client transport.

In the object inspector find the **MaxRetries** and **MaxRetriesAlternative** properties.



MaxRetries determines how many times the client will try to reconnect to the same server when doing a request before giving up. **MaxRetriesAlternative** determines how many times the client will try to reconnect to alternative servers after giving up connecting to the primary server.

The actual control of which server to reconnect to is controlled via the **OnReconnect** event:



The following event code lets the client try reconnect to a random server out of the known servers:

```
procedure TForm1.kbmMWTCPIPIndyClientTransport1Reconnect(Sender: TObject;
  Alternative: Boolean; RetriesLeft: Integer);
var
   i:integer;
const
   AltHosts:array [0..9] of string = (
     'xyz.host1.com',
     'xyz.host2.com',
     '192.168.4.22',
     'xyz.host4.com',
     'xyz.host5.com',
     'xyz.host6.com',
     'xyz.host7.com',
     'xyz.host8.com',
     'xyz.host9.com',
     'xyz.host10.com' );
begin
     // Check if to reconnect to a random alternative host.
     if Alternative then
     begin
          i:=Random(High(AltHosts)+1);
          Host:=AltHosts[i];
     end;
end;
```

Alternative is true if the transport wants you to set an alternative server to connect to. **RetriesLeft** gives a count of how many retries is left until the client gives up.

All connection properties of the client transport can be altered to match the new server to connect to.

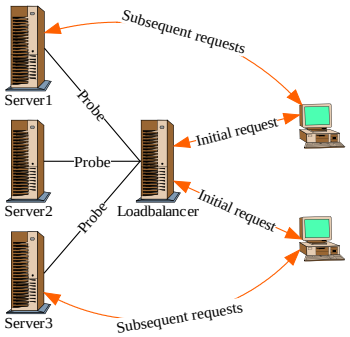
It's possible to provide a complete connection string, instead of only the host value. The advantage of that is that the client can connect on different ports, with different streamformats etc. as required by the application servers.

To do that, one would change the example code to:

```
procedure TForm1.kbmMWTCPIPIndyClientTransport1Reconnect(Sender: TObject;
  Alternative: Boolean; RetriesLeft: Integer);
var
   i:integer;
const
   AltHosts:array [0..9] of string = (
     'HOST=xyz.host1.com;PORT=3000',
     'HOST=xyz.host2.com;PORT=4000',
     'HOST=192.168.4.22;PORT=3000',
     'HOST=xyz.host4.com;PORT=3000',
     'HOST=xyz.host5.com;PORT=3000',
     'HOST=xyz.host6.com;PORT=3000',
     'HOST=xyz.host7.com;PORT=3000',
     'HOST=xyz.host8.com;PORT=3000',
     'HOST=xyz.host9.com;PORT=3000',
     'HOST=xyz.host10.com;PORT=4000' );
begin
     // Check if to reconnect to a random alternative host.
     if Alternative then
     begin
          i:=Random(High(AltHosts)+1);
          ConnectionString.Text:=AltHosts[i];
     end;
end;
```

There are a large number of possible settings for a connection string, depending on the type of transport. To deduce which connectionstring to use, put set the appropriate property values on the transport, and click the ConnectionString property to see the generated connection string. Then copy that to your application.

Centralized load balancing



This setup is based on two different application servers, one which contain load balancing components and nothing else, and one or more others which contain business code and a load balancing service.

First we create the load balancer application, and then we modify a standard demo server project to include the load balancing service to allow it to be probed.

Creation of an application server only working as a load balancer

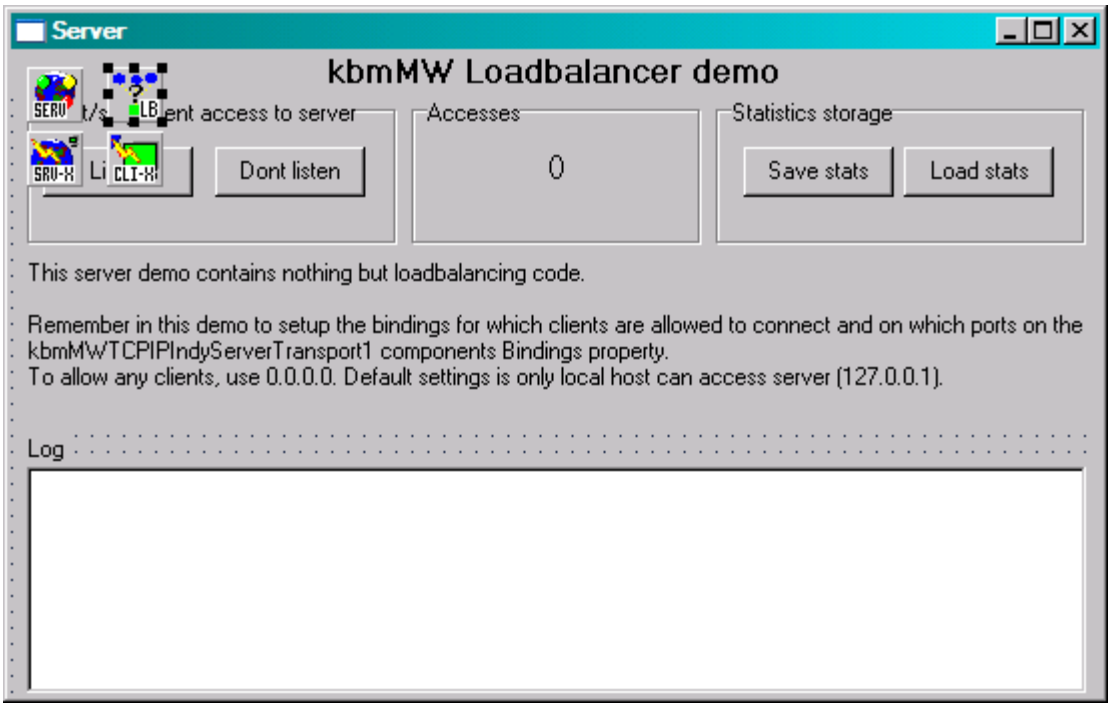
Make a new standard application. You can make it as a NT service application or other types applications too if you want to. But for the purpose of this paper, we create a standalone application.

Put a **TkbmMWServer**, some server transport like for example **TkbmMWTCPIPIndyServerTransport**, a load balancer component, as example **TkbmMWRoundRobinLoadBalancer** and finally a client transport like **TkbmMWTCPIPIndyClientTransport**.

The server transport is used to receive requests from the clients just like any other normal application server.
The client transport is used by the load balancer component to probe servers specified on the **KnownServers** list to see if they are alive and how they perform.

The Round Robin load balancer component takes selects between available servers using the Round Robin algorithm (next server in list). kbmMW contains more load balancing components like **TkbmMWRandomLoadBalancer** and **TkbmMWBestFitLoadBalancer**. The random load balancer selects between the available servers in a random manner. The best fit load balancer takes server load into account and tries to find one that currently has least load.
All load balancers takes into account which service and service version there is asked for.
Thus some servers may contain some services and others some other. The load balancer will through its probe automatically figure out which servers have which services with which versions and use that knowledge during the load balancing.

If you add a few more visual components you could end up with a load balancing server form like this:



The components need to be connected together.

kbmMWServer1.LoadBalancer -> kbmMWRoundRobinLoadBalancer1
kbmMWRoundRobinLoadBalancer1.Transport -> kbmMWTCPIPIndyClientTransport1
kbmMWTCPIPIndyServerTransport1.Server -> kbmMWServer1

Just like with all other application servers, you need to decide which StreamFormat clients should use to connect to this server. Usually choose **STANDARD**, and set **VerifyTransfer:=true** on the server transport.

The properties on the client transport don't matter at this time. They will automatically be set by the load balancer component according to the **KnownServers** string list contents.

The **KnownServers** property of the load balancer should contain one or more connection strings which define the complete access path to other application servers.

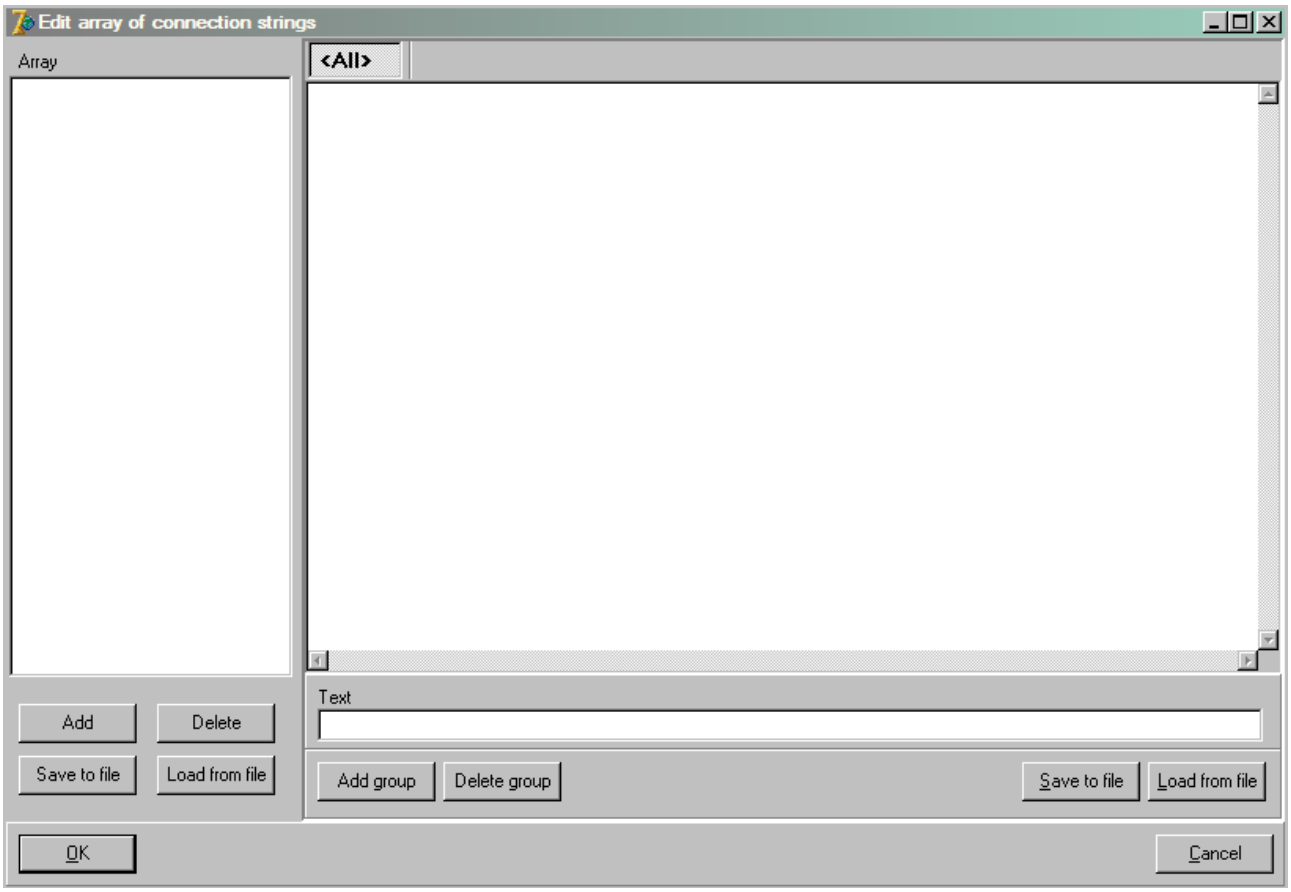
The contents of the connection string depend on the type of transport. For the Indy TCPIP transport, a connection string could look like this:

```
STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host1.com;PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0
```

Only relevant elements are required. Thus as an example if all application servers are using **VerifyTransfer:=true** then you can set that on the client transport and then remove the VERIFYTRANSFER section from the connection string to make it shorter.

Either fill out the **KnownServers** during design time or do it in runtime before setting the load balancers Active property to true.

At designtime, clicking on the KnownServers property, the following editor will show.



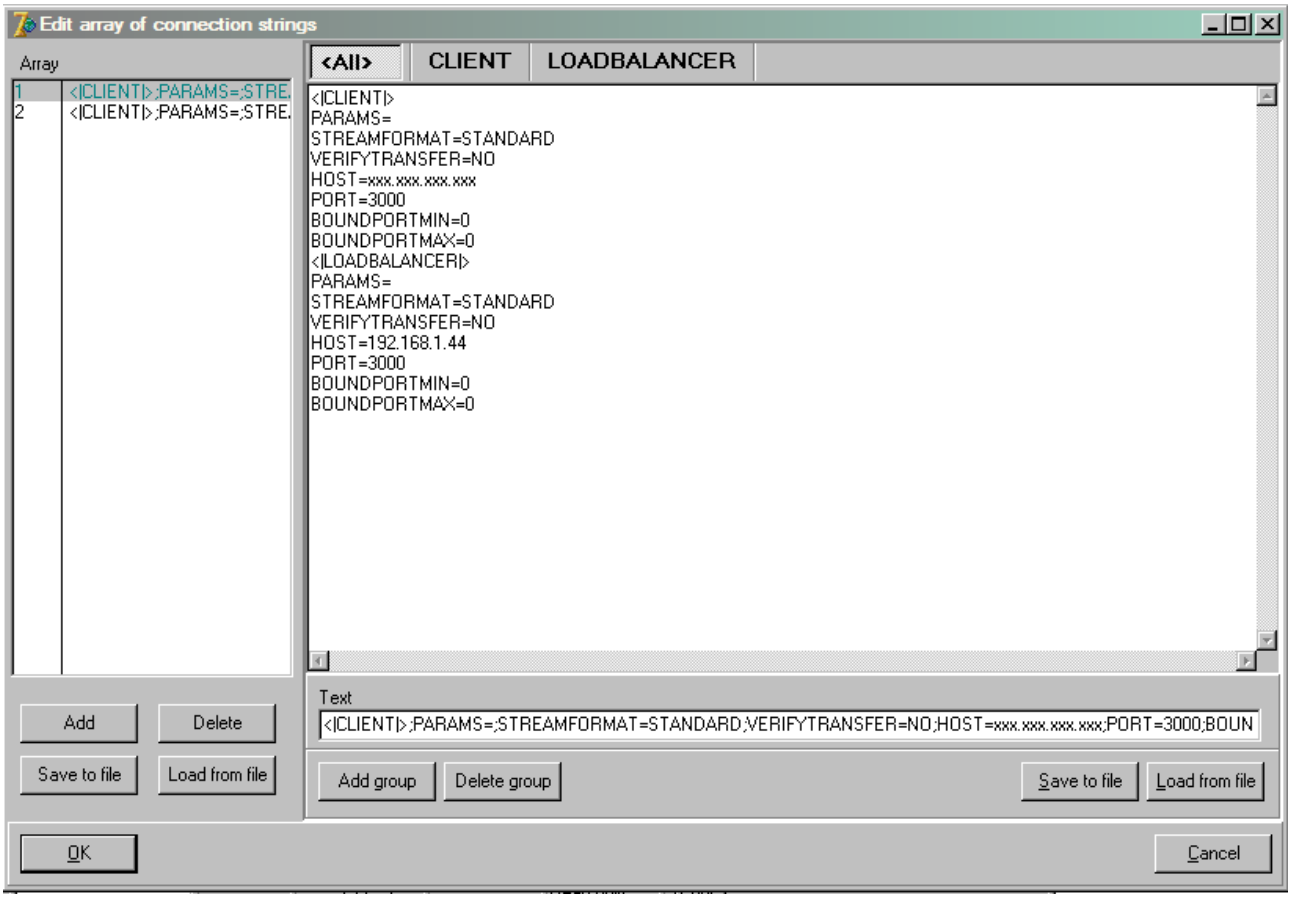
As KnownServers contains an array of connection strings, the editor have the array listed in the left pane, and the connection strings for that entry in the array, in the right pane.

To add an entry, click Add in the left pane. When that has been done, the right pane is now editable.

kbmMW v. 2.50 supports differentiating between what connection string other load balancers must use to connect to an application server, and what connection string clients must be handed to connect to the application server. That can be very practical because the clients may be required to use encryption or SOAP or some other special setting, while all the load balancers want to use the most efficient method because they (in our example) all are placed within a safe network in the company.
If the load balancers and the clients should use the same connection string, all you have to do is enter the connection string in the ALL tab. You can copy the connection string from a client transports connection string editor and paste it into this editor.

If the load balancers should use a separate connection string to connect to the specified server, you need to add a group. Name it **LOADBALANCER**
Similarly you can add a group called **CLIENT** which contains the connection string the clients should use to connect to the application server you are specifying.

The ALL tab will then contain both groups as seen in this screenshot:



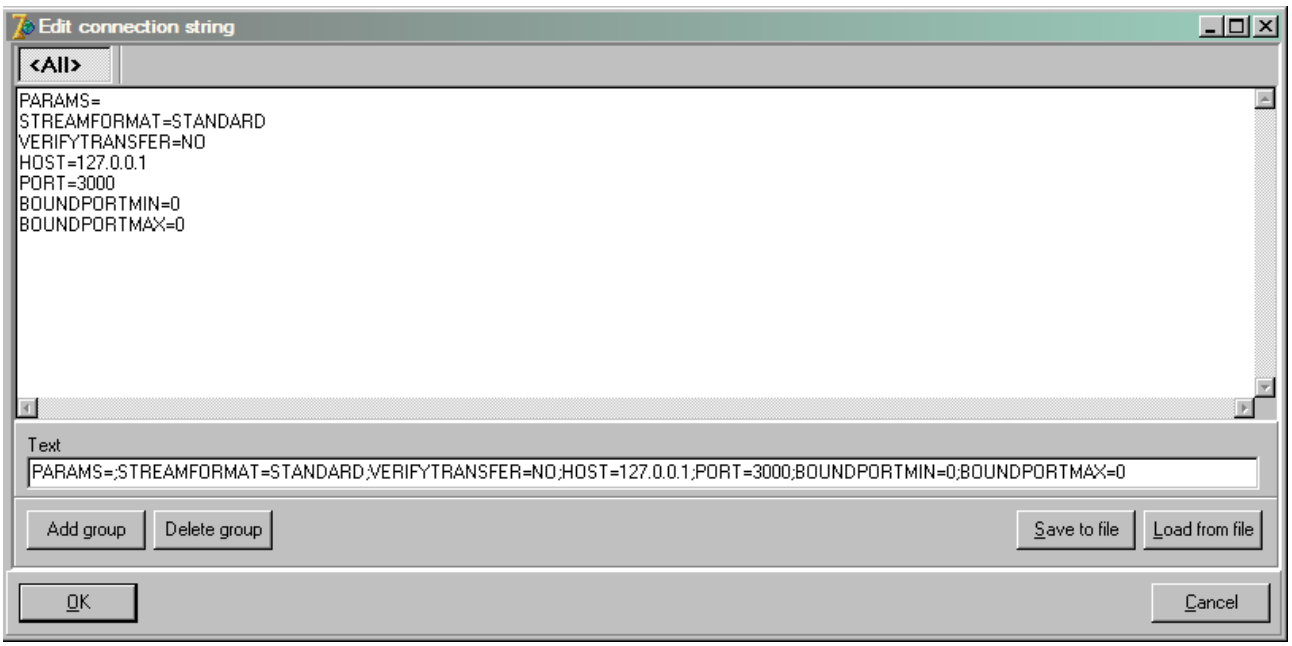
Clicking on the CLIENT tab will show only the parts related to the CLIENT group etc.
It's allowed to add additional custom groups for your own use having whatever name you choose. Just remember that for the KnownServers property, kbmMW only recognizes the special groups named CLIENT and LOADBALANCER.

As an example KnownServers can be setup at runtime in the forms OnCreate event:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   // Add a list of known servers to the known servers property of the loadbalancer.
   // This can be done at designtime too if so desired.
   with kbmMWRoundRobinLoadBalancer1 do
   begin
                // Notice that connection strings are added to the KnownServers list.
                // Its also possible to use connection strings containing groups like shown in the
                // design time editor. Just copy the contents of the Text field of the design time editor
                // and paste in as the connection string for the Add method.
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host1.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host2.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host3.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host4.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host5.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host6.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host7.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host8.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                KnownServers.Add('STREAMFORMAT=STANDARD;VERIFYTRANSFER=YES;HOST=xyz.host9.com;'+
                                 'PORT=3000;BOUNDPORTMIN=0;BOUNDPORTMAX=0');
                Active:=true;
   end;
end;
```

This example defines 10 known application servers which all should participate in this server cluster. The same connection string is used for load balancers and for clients to connect to the application server.

The connection strings can actually be easily obtained from all client transport components via their **ConnectionString** property. Put a dummy client transport component on the form and set its properties to match a server, and then double click the **ConnectionString** property. This will bring up a dialog which allows you to copy the complete connection string which you can then paste where needed.



Further the load balancers role has to be defined. When is it going to redirect clients to another server?
Since this application server do not contain any services and thus only operates as a load balancer, we want the load balancer to redirect all requests to another server.
Set the **kbmMWRoundRobinLoadBalancer1.When** property to [**mwlbwAlways**].

If **mwlbwNoService** is specified, it will redirect to other servers when the current server do not contain the service with the specified version requested by the client.
If **mwlbwFull** is specified, it will redirect to other servers when the requested service is found on the current server, but its currently serving max number of concurrent requests (**MaxInstances** of the servicedefinition).
If **mwlbwAlways** is specified, then all requests will be redirected.

This actually is all what is needed to make a central load balancing server. If to run a high availability setup, minimum two such servers should run with the clients configured for failover to one of those two servers. Then there will be no single point of failure on the server side.

An additional little feature (which the demo load balancer also shows) is the OnProbeConnectFailed event of the load balancer. Its called whenever a connection cannot be obtained to a server in the cluster. This can be tracked in a log for example.

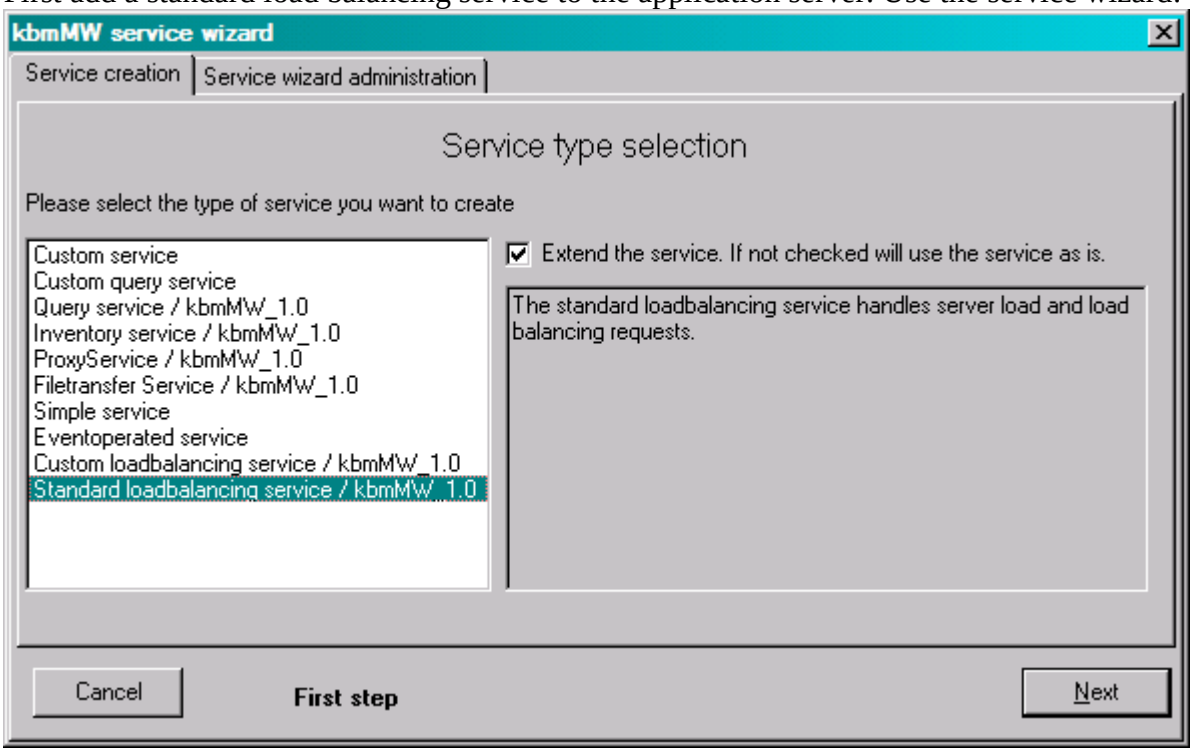
```
procedure TForm1.kbmMWRoundRobinLoadBalancer1ProbeConnectFailed(
  AConnectionString: TkbmMWConnectionString; AException: Exception);
begin
     mLog.Lines.Add(AException.Message+' : '+AConnectionString.Text);
end;
```

Preparing an app. server for participation in a centralized load balancing server cluster

All application servers which participate in a load balancing cluster should contain a load balancing service which is used by the load balancers to probe the server's status etc.

As an example we take the standard BDE demo server and add load balancing support to it, so it can participate in a load balancing cluster.

First add a standard load balancing service to the application server. Use the service wizard:



Go through all the pages without changing anything even though some entries may be specified as required.

This will give you a standard load balancing service which should be registered on your application server. Eg:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
     kbmMWServer1.RegisterService(TkbmMWInventoryService,false);
     kbmMWServer1.RegisterServiceByName('KBMMW_QUERY',TTestQuery,false);
     kbmMWServer1.RegisterService(TkbmMWLoadBalancingService3,false);   // << The LB service!
end;
```

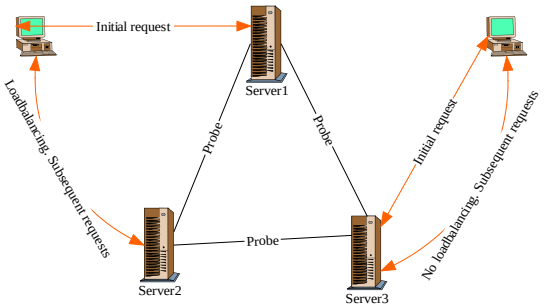
Make sure to add the newly created unit to the uses clause of the form that contains the TkbmMWServer.

Then compile the application server and that's all what is needed.

Since the load balancing service is added to the kbmMWServer just like any other services, everybody can in access it, even clients unless requests are blocked by specific checks in OnAuthenticate or similar events.
Its possible to bind a specific service to a specific server transport. Thus only requests coming through that transport will be allowed to access the service. That way you can simply add another server transport component to the form, set its connection properties so only clients withing the cluster can access it typically via the bindings property of for example the Indy server transport, and set the BoundTransport property of the service to point on that specific transport.
Then the application server will listen to two server transports one of them serving normal clients, the other serving load balancing clients.

Or the load balancing transport could be using another media than the one the clients are using. For example a RS232 or USB transport which is leads to a set of closely connected servers.

Distributed loadbalancing



This setup is based on standard application servers, containing real business logic, in which load balancing capabilities have been enabled.

We modify a standard demo server project to include the load balancing service and load balancing probes to both allow it to be probed and to probe others.

Preparing an application server to be part of a distributed load balancing server cluster

This is a more advanced version of an application server which participates in a centralized load balancing cluster why you can add to that server to make it a distributed load balanced server.
In essence it combines a load balancer server with a load balanced application server (an application server that contains the load balancing service).

Add, as with a standalone load balancer, a load balancing component and a client transport used by the load balancer to the same form/datamodule hosting the TkbmMWServer component.
Point kbmMWServer.LoadBalancer to the load balancer component etc. just like for a standalone load balancer.

Where there is a difference is in the setting of the load balancer component's When property. Choose to set it to [mwNoService,mwFull] which means that if a client requests a service which is not found on this server, it will automatically try to redirect the client to another server publishing that service. mwFull means that if the maximum number of instances has been reached on the requested service on this server, the client will automatically be redirected to another server publishing the same service.
For this to work, its important that the service has been registered with the DontQuery service definition property set to true (default false). If not, clients will be queued waiting for a service instance. Eg.:

```
with kbmMWServer1.RegisterService(Tyourservice,false) do
begin
  MaxInstances:=10;  // Maximum 10 concurrent client requests for this service.
  DontQueue:=true;
end;
```

Now the cluster will be self load balancing. What is important in such a distributed cluster is to determine how many instances should be allowed for each service before it severely will impair performance and client requests should be redirected.

This concludes the load balancing and failover whitepaper.

best regards

Kim Madsen
Components4Developers