

Prototype Load Balancing Infrastructure for COM/MTS

Tim Ewald, tjewald@develop.com

April 2nd, 1999

Table of Contents

1. [Overview](#)
2. [Architecture](#)
3. [Pieces](#)
 - o [The LoadBalancer Service](#)
 - o [Algorithms](#)
 - o [Envelopes](#)
 - o [Load Balanced Classes](#)
 - o [Clients](#)
4. [Method Timing](#)
5. [Custom Algorithms](#)
6. [What About Context Propagation?](#)
7. [What About JIT Activation?](#)
8. [The Code](#)

Overview

Scalability mandates spreading work across multiple hosts. Load balancing is necessary to spread work evenly across multiple hosts thereby optimizing both user response time and return on investment for additional hardware. The MTS programming model is designed to create scalable systems that can easily be spread across multiple hosts. Unfortunately, neither MTS nor COM provides a load balancing service. This infrastructure is intended to fill that gap.

COM+ does include a load balancing service and it's architecture inspired this work. Under COM+, a single machine (or a hardware cluster of machines -- 2 if you're using MS Cluster Server) is designated as a routing server. All client requests are directed to the routing server. The routing server maintains a list of available servers and redirects object creation requests to these server machines as dictated by a selection algorithm based on method timing.

The beauty of the COM+ load balancing service is that you don't have to change any thing in either the client or the server to make it work. You simply configure the client to send creations to a routing server, configure the routing server to know about the servers it can delegate creation to, and configure the classes for load balancing (so they know to collect performance data in their interceptors).

The problems with COM+ load balancing are: it isn't shipping yet, when it does ship it will work on Advanced Server only, and while Microsoft has talked about custom activators, they haven't yet shared the details of their implementation so it may not be possible to implement alternative algorithms with the first release of the service.

To address all these issues and because load balancing is really tremendously important, I designed and prototyped my own load balancing service that works with COM and MTS today.

WARNING: THIS INFRASTRUCTURE IS A PROTOTYPE AND ONLY A PROTOTYPE. IT IS NOT READY TO BE DEPLOYED IN A PRODUCTION ENVIRONMENT. ITS PURPOSE IS SIMPLY TO EXPLORE A POSSIBLE APPROACH TO CREATING A LOAD BALANCING SERVICE FOR COM AND TO DOCUMENT SOME INTERESTING OBSTACLES AND HOW TO GET AROUND THEM. LIKE THE PLUMB THAT IS ONLY GUARENTEED TO ANSWER ONE QUESTION CORRECTLY (Which way is down?), THIS PROTOTYPE IS GUARENTEED TO DO ONLY ONE THING: CONSUME SPACE ON YOUR HARDDRIVE. YOUR USE OF THIS INFRASTRUCTURE FOR ANY OTHER PURPOSE IS AT YOUR OWN RISK.

Architecture

The heart of the load balancing infrastructure is the LoadBalancer service which is installed on a single node which acts as the routing server (if you're worried about a single point of failure make the routing server part of a Wolfpack cluster). The LoadBalancer registers class objects for all the classes it is configured to balance. When creation calls arrive at the routing server, the SCM directs them to the LoadBalancer. The class objects the LoadBalancer registered in turn forward the creation calls off to one of a set of servers as dictated by the current algorithm.

It is important to note that after the object is created on the target server the load balancer is out of the picture. This mirrors the behavior of the SCM, which is there to introduce objects to clients but not to monitor how the relationship progresses.

The architecture of the COM+ load balancing service differs only in that the load balancing service can be viewed (at least logically) as a part of the SCM.

Pieces

The LoadBalancer Service

The LoadBalancer itself is implemented as an NT service. It does three things on startup:

1. Register class objects for all classes registered as implementing the category "Load Balanced Classes".
2. Load an algorithm based on the string stored in the "Default Algorithm" value of the HKEY_LOCAL_MACHINE\Software\DevelopMentor\LoadBalancing\RoutingServer registry key. This string may be either a CLSID or a ProgID.
3. Load a set of hosts based on the subkeys of the HKEY_LOCAL_MACHINE\Software\DevelopMentor\LoadBalancing\RoutingServer registry key and initialize the algorithm with that set of hosts.

The LoadBalancer service must be configured to run as a user other than System which has the power to create objects on each of the target servers.

Algorithms

An algorithm is simply an MTA-based COM class that implements the interfaces `ILoadBalancingAlgorithm` and `IHosts`:

```
// Algorithm.idl

import "objidl.idl";

[
    uuid(741F3750-E3B1-11d2-8117-00E09801FDBE), object
]
interface ILoadBalancingAlgorithm : IUnknown
{
    HRESULT CreateInstance([in] REFCLSID rclsid,
                          [in] REFIID riid, [out, iid_is(riid)] void **ppv);
}

[
    uuid(6345D6A0-E6DC-11d2-B72D-00A0CC212296), object
]
interface IHosts : IUnknown
{
    HRESULT SetHosts([in] long nCount, [in, size_is(nCount)] LPOLESTR *rgwszHosts);
}
```

`ILoadBalancingAlgorithm` is called by the shim class object registered by the LoadBalancer service to delegate object creation to the algorithm.

The `IHosts` interface is used to seed the algorithm with a list of hosts.

Algorithms should be marked `ThreadingModel=Free` to ensure that they run in the MTA for maximum performance. However, the implementation of the LoadBalancer gaurentees that `SetHosts` will be called only once and therefore does not have to be thread-safe.

Four load balancing algorithms are provided:

- **Random** selects a server at random.
- **Round-robin** selects the next server in the collection.
- **CPU Load** selects the next server based on CPU load data gathered from the NT performance counters.
- **Method-timing** selects a server based on method timing data gathered by intercepting channel communications.

The method timing algorithm is discussed in more detail below.

Envelopes

One goal of this project is to implement load balancing without requiring client and server code to be changed. This is particularly true of client code that calls `CoCreateInstance`, especially code written in languages that hide this detail. Unfortunately the current implementation of `CoCreateInstance` doesn't support returning an object reference that refers to an object living on a server other than the one targeted by `CoCreateInstance`. In other words: if a client on machine A calls `CoCreateInstance` with a

remote server name B and the class object running on machine B (in the load balancing service) attempts to return an interface pointer to an object it created on machine C (as dictated by its algorithm), the client's call will fail with `RPC_E_INVALID_OXID`.

While there are undoubtedly good reasons why the implementation works this way, it feels like a bug to the COM philosopher because all object references should be equal. Since it is valid for a class object to return an object reference from another context, e.g., apartment, on its own machine, it should be able to return an object reference from a context, e.g., apartment, on another machine. But it can't.

This really got me down for a while because I was confronted with requiring clients to use `CoGetObject`. While one could argue this change is reasonable price of admission to the wonderful world of load balancing, it felt too high. Then my colleague Kevin Jones lit the fuse that burned quickly to an explosive insight. Kevin wondered if some form of client-side smart proxy might do the trick. Initially I discarded this notion because I didn't want to change the client code. Then it hit me: this problem can be solved with a COM variation on the Envelope/Letter idiom formalized by Coplien in his master work.

The current implementation of `CoCreateInstance` can't return a standard marshaled object reference from a machine other than the one targeted at the remote server by the call to `CoCreateInstance` (machine B in the paragraph above). But `CoCreateInstance` can return a custom marshaled object reference that happens to carry a standard marshaled object reference as its payload. When the custom marshaled object, the Envelope, is returned to the client it carries the marshaled image of the real interface pointer, the Letter, as its payload, shielding it from the prying eyes of `CoCreateInstance`. When the Envelope arrives on the client and is unmarshaled, it unmarshals its payload and simply returns the Letter: a proxy to the object running on the target server.

The `IEnvelope` interface is shown below. Note the use of a write-only property (I always wondered if there was ever a use for such a thing!). There's no need for a get method because the Envelope opens itself when it unmarshals.

```
[
    object,
    uuid(5E7F74C0-E165-11D2-B72C-00A0CC212296),
    helpstring("IEnvelope Interface"),
    pointer_default(unique)
]
interface IEnvelope : IUnknown
{
    [propput, helpstring("A write-only property!")] HRESULT Letter([in] IUnknow
};
```

Here is the implementation of the the Envelope class. Note that the Envelope marshals by value using its own CLSID, but returns a pointer to the letter when it unmarshals.

```
STDMETHODIMP CEnvelope::put_Letter(
    /* [in] */ IUnknown *pUnk)
{
    m_pMshl = pUnk;
    return S_OK;
}

STDMETHODIMP CEnvelope::GetUnmarshalClass(
    /* [in] */ REFIID riid,
```

```

/* [unique][in] */ void __RPC_FAR *pv,
/* [in] */ DWORD dwDestContext,
/* [unique][in] */ void __RPC_FAR *pvDestContext,
/* [in] */ DWORD mshlflags,
/* [out] */ CLSID __RPC_FAR *pCid)
{
    *pCid = CLSID_Envelope; // return Envelope CLSID to shield raw reference fr
    return S_OK;
}

STDMETHODIMP CEnvelope::GetMarshalSizeMax(
/* [in] */ REFIID riid,
/* [unique][in] */ void __RPC_FAR *pv,
/* [in] */ DWORD dwDestContext,
/* [unique][in] */ void __RPC_FAR *pvDestContext,
/* [in] */ DWORD mshlflags,
/* [out] */ DWORD __RPC_FAR *pSize)
{
    return m_pMshl->GetMarshalSizeMax(riid, pv, dwDestContext, pvDestContext, m
}

STDMETHODIMP CEnvelope::MarshalInterface(
/* [unique][in] */ IStream __RPC_FAR *pStm,
/* [in] */ REFIID riid,
/* [unique][in] */ void __RPC_FAR *pv,
/* [in] */ DWORD dwDestContext,
/* [unique][in] */ void __RPC_FAR *pvDestContext,
/* [in] */ DWORD mshlflags)
{
    return m_pMshl->MarshalInterface(pStm, riid, pv, dwDestContext, pvDestConte
}

STDMETHODIMP CEnvelope::UnmarshalInterface(
/* [unique][in] */ IStream __RPC_FAR *pStm,
/* [in] */ REFIID riid,
/* [out] */ void __RPC_FAR *__RPC_FAR *ppv)
{
    return CoUnmarshalInterface(pStm, riid, ppv); // Open the envelope
}

STDMETHODIMP CEnvelope::ReleaseMarshalData(
/* [unique][in] */ IStream __RPC_FAR *pStm)
{
    return CoReleaseMarshalData(pStm);
}

STDMETHODIMP CEnvelope::DisconnectObject(
/* [in] */ DWORD dwReserved)
{
    return S_OK;
}

```

This envelope technique does require the Envelope class to be registered on the routing server and on each client. This is a little additional burden, but a *very* cheap price for making CoCreateInstance work with the load balancer.

Actually, there is one additional, slight price. The custom marshaling interface `IMarshal` includes the method `ReleaseMarshalData` which is meant to be called by the remoting architecture if the unmarshal fails to give the custom marshaling object a chance to clean up its resources. If, for instance, an object was custom marshaling using sockets for piping data, this would give the marshaling object a chance to close the socket it set up in `MarshalInterface`. Unfortunately with `Marshal-by-Value` objects like the `Envelope`, this method isn't called because the server-side copy of the object has already been destroyed. For most MBV objects this is no big deal because they don't carry references to objects. But the `Envelope` does carry an object reference, and if it fails to unmarshal it will leak references to the object. Luckily for us, the COM garbage collector will kick in and clean up these leaked references within 6 minutes, so this shouldn't be a problem.

Load Balanced Classes

For a class to be load balanced it must be registered on each of the target servers. It must also be registered on the routing server. The registration information need on the routing server is both minimal and special. Each CLSID must be registered as implementing the category `Load Balanced Classes` and with the `AppID` of the routing server. No other information is necessary, though its presence won't do any harm. For instance, the minimal registration necessary for the sample code's `Test Class` is:

```
[HKEY_CLASSES_ROOT\CLSID\{3C406C7B-E3B1-11D2-8117-00E09801FDBE}]
"AppID"="{5E7F74A4-E165-11D2-B72C-00A0CC212296}"
```

```
[HKEY_CLASSES_ROOT\CLSID\{3C406C7B-E3B1-11D2-8117-00E09801FDBE}\Implemented Categor
```

The category registration is necessary because the `LoadBalancer` service enumerates this category to decide what CLSIDs to register class objects for. The presence of the `AppID` is of special interest. While the service can register class objects whether or not the CLSID has the same `AppID` as the service, the class objects will not be exposed to the a client unless the `AppID` entry is present. Also, the `AppID` will enable the service to auto-start if a creation request for a load-balanced class arrives when it isn't running.

Clients

Nothing has to be done with clients other than registration of the `Envelope` class on each client machine.

Method Timing

The method timing algorithm bears special consideration because it is by far the most complex of the three and because it does not yet yield a stable load (in fact that's why I call this a `Load Balancing Infrastructure` and not a `Load Balancer` ;->). Here's how method timing works...

The current iteration of COM supports an undocumented (well semi-documented: the interface is in the headers and Don wrote a column about it) feature called `Channel Hooks`. A `Channel Hook` is registered into a process and given the chance to piggy-back data on every call through the channel. To support method timing, I built a channel hook that records the start time of a call into a server and, upon completion of the call, calculates the total time a method call took to complete. This data is collected in shared memory on each target server and collected periodically by a thread controlled by the `Method Timing` algorithm object running on the routing server.

To make method timing work you have to load the channel hook into both the client and server process. Since I didn't want developers to have to change either client or server code, I opted to load the channel hook as part of the proxy/stub DLL. This does require a minimal amount of work for developers: you have to link your PS code against some additional code and slightly change the build steps for your PS DLL. The code you have to link against is shown below:

```
#include
#include

DEFINE_GUID(CLSID_Loader,
0x233108A3, 0xE3CD, 0x11D2, 0x81, 0x17, 0x0, 0xe0, 0x98, 0x1, 0xfd, 0xbe);

extern "C" BOOL WINAPI OldDllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpRes

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    static IUnknown *pUnk = 0;
    switch(dwReason)
    {
        case DLL_PROCESS_ATTACH :
        {
            HRESULT hr = CoCreateInstance(CLSID_Loader, 0, CLSCTX_INPROC_SERVER, IID_IUnknown, (void**)&pUnk);
            if (SUCCEEDED(hr)) pUnk->Release();
            break;
        }
        case DLL_PROCESS_DETACH :
        {
            break;
        }
    }
    return OldDllMain(hInstance, dwReason, lpReserved); //TRUE;    // ok
}
```

Notice that this code implements DllMain and when it is attached to a process, creates an instance of CLSID_Loader, an object implemented in MethodTimeHook.dll, the DLL which sets up the channel hook. There is no need to hold this reference, it is needed just long enough to load the DLL. Don't worry about the hook DLL unloading, it's DllCanUnloadNow implementation always returns FALSE.

The thing this code does is to delegate to OldDllMain, which is really the original DllMain method provided by the PS infrastructure in dlldata.c. To achieve this name changing trick you have to modify your PS DLL makefile as shown below (this example is a .mk file generated by ATL):

```
TestServerps.dll: dlldata.obj TestServer_p.obj TestServer_i.obj methodtimehookps.obj
    link /dll /out:TestServerps.dll /def:TestServerps.def /entry:DllMain method
        mtxih.lib mtx.lib mtxguid.lib \
        kernel32.lib rpcndr.lib rpcns4.lib rpcrt4.lib oleaut32.lib uuid.lib
        ole32.lib advapi32.lib

.c.obj:
    cl /c /Ox /DWIN32 /D_WIN32_WINNT=0x0400 /DREGISTER_PROXY_DLL /DDllMain=OldD
        /MD \
        $<
```

```
.cpp.obj:
    cl /c /Ox /DWIN32 /D_WIN32_WINNT=0x0400 /DREGISTER_PROXY_DLL \
        /MD \
        $<

clean:
    @del TestServerps.dll
    @del TestServerps.lib
    @del TestServerps.exp
    @del dlldata.obj
    @del TestServer_p.obj
    @del TestServer_i.obj
```

Notice that TestServerps.dll is not linked against methodtimehookps.obj, that for the compilation of all .c files DllMain has been remapped to OldDllMain and compilation rules for .cpp files have been added.

If you are using the Universal Marshaler to marshal dual and/or oleautomation interfaces via typelibs, you can't use the method timing channel hook. This doesn't mean you can't use dual and/or oleautomation interfaces, it just means that you have to build PS DLLs for them. This is easy if you're using ATL because the wizards emit interface definitions outside the library statement, which means proxy/stub code is already emitted for them (the Test Class sample code demonstrates this). If you're using VB-generated interfaces you have to reverse engineer the IDL via OleView or an equivalent tool and then build a proxy/stub DLL. Finally, you have to remember to register the PS DLL **after** registering the server which includes your typelib so that typelib registration doesn't overwrite the registration of your DLL.

You always need to register the MethodTimeHook DLL on clients and servers that are going to use method timing.

The previous paragraphs briefly explain how the method timing data is collected. The more interesting question is what to do with the data. The method timing channel hook will count the total number of calls and the total processing time for the calls on a given node. Periodically, a thread managed by the method timing algorithm object on the routing server reaches polls each target server to get this information. It does this by instantiating the Loader object implemented in the MethodTimeHook DLL, which is configured to run in a the standard surrogate process, DLLHOST.EXE. The loader object returns a cooked version of the data from each server which is used to select the server to send work to until the next time the thread does its polling.

I wasn't able to find a documented algorithm for load balancing based on method timing so I cooked up my own. I didn't have a lot of time to spend on it so it isn't as tuned up as I'd like, but it is better than what I started with. In essence, the loader cooks the timing data to return "task-seconds" per interval where "task-seconds" is the total time of all measured COM calls ending in the interval. The interval is controlled by the algorithm, and the current version is 1/2 a second. The load also amplifies the signal slightly as the data is being handled as longs and the application of floor implicit in integer division mean values under 1 (the original data range) are dropped. The method timing algorithm object softens the impact of dramatic method timing changes by only applying 1/2 the delta from the reading at the previous interval. All these decisions are based entirely in empirical study on my set of hosts for this particular test client and server. Your mileage may vary (greatly). If you have experience and/or time to spend on improving this algorithm to make it more stable, please do. Send me the results and I'll

incorporate them in the kit.

The CPU Load algorithm is similar to the method timing algorithm in that it gathers data from the target servers. The implementation uses the PDH library supplied with the Platform SDK to access the NT performance counters on the remote machines. For this to work you have to have PDH.DLL somewhere on your path (it is available as part of the Platform SDK in \mssdk\bin\winnt). The LoadBalancer service must be running as an account that is allowed access to this information.

The performance counter being monitored is "System\% Total Processor Time" and it is consumed raw.

To help debug these pieces, both the method timing and CPU load algorithms log data points to the debug window. To collect this data, simply attach a debugger to the running service.

Custom Algorithms

If you like you can write custom algorithms for this infrastructure. One interesting idea would be to send timing data from clients to the load balancer so that a variation on the method timing algorithm could include network time. This could be done fairly easily by having the algorithm wrap the interface pointer it is returning in a Universal Delegator (visit <http://www.develop.com/kbrown> for more info) to the client via the envelope. The UD could track method times and a service on the client could periodically send them up to the routing server...

If you want to write a custom algorithm simply:

1. Implement `ILoadBalancingAlgorithm` and `IHosts`.
2. Mark your class as `ThreadingModel=Free`.
3. Implement `CreateInstance` in a thread-safe manner.
4. Don't worry about making `SetHosts` thread-safe, it won't be called more than once.

What About Context Propagation

Several people have asked how I intend to solve the MTS context propagation problem. To propagate context in MTS, objects within MTS must create sub-objects using `IObjectContext::CreateInstance`. The implementation of this method makes sure context information (activity, transaction stream, security bits, etc.) gets propagated to the new object. It does this through an undocumented interface on the MTS class factory wrapper called `IClassFactoryWithContext`. Without documentation or even a specification of this interface, adapting the load balancer to support it is exceedingly difficult if not impossible.

But that's okay, because, frankly I argue that one should avoid load-balancing sub-objects anyway. It makes sense to load balance once so that client's objects fan out across servers. From there sub-objects should, if possible, be created on the same host or on a parallel host with a 1 to 1 mapping to the first tier of servers covered by the load balancer. That being said, I'm sure someone out there will want to load balance their sub-objects and they'll have some good reason why they need to do it. What can I say, it isn't going to work this time around. I haven't had a chance to test to see if this will work with the COM+ load balancing service, but my guess is it should.

What About JIT Activation

Many people want load balancing to work across JIT activations in MTS. Needless to say this

infrastructure doesn't work that way -- and neither does the load balancing service in COM+. The reason for this limitation is that migrating the server side of the proxy/stub connection to another context while keeping the plumbing aware of what's going on (e.g., the OXID resolvers on the client and both servers which would need to update their ping sets so GC keeps working without a hitch, and who know what else) is a *very* complicated problem. In fact, this is so difficult that, if it were me, I'd likely add client-side layer that would release the proxy transparently and rebind to another server. Developing such a smart-proxy to encapsulate re-balancing details should be fairly straight forward (although I leave it to you to decide how often you should rebind - experiment with the test client and the method timing algorithm to see how much fun this could be).

One colleague told me recently he thought COM+ load balancing was broken because it violated the MTS "grab a reference and hold it, JIT Activation will take care you" programming model. He's right about the violation of that rule, but that's not the MTS programming model (at least not now that there's load balancing).

And Finally: The Code

REPEAT WARNING: THIS INFRASTRUCTURE IS A PROTOTYPE AND ONLY A PROTOTYPE. IT IS NOT READY TO BE DEPLOYED IN A PRODUCTION ENVIRONMENT. ITS PURPOSE IS SIMPLY TO EXPLORE A POSSIBLE APPROACH TO CREATING A LOAD BALANCING SERVICE FOR COM AND TO DOCUMENT SOME INTERESTING OBSTACLES AND HOW TO GET AROUND THEM. LIKE THE PLUMB THAT IS ONLY GUARENTEED TO ANSWER ONE QUESTION CORRECTLY (Which way is down?), THIS PROTOTYPE IS GUARENTEED TO DO ONLY ONE THING: CONSUME SPACE ON YOUR HARDDRIVE. YOUR USE OF THIS INFRASTRUCTURE FOR ANY OTHER PURPOSE IS AT YOUR OWN RISK.

This code has been tested on 3 of my NTS4 SP4 boxes and a could of machines Martin setup over in the UK, so I make no promises.

There are lots of outstanding issues with this code. For instance:

- There is little error checking, especially with configuration. In the COM security and MSMQ traditions, get it right or else it just won't work. Sorry, I was more interested in writing algorithms. The kit's bin directory does include some scripts to help with installation and configuration and the docs directory includes a configuration script, [config.htm](#). Be wary if you start working on the code, automatic reregistration of the LoadBalancer in particular wacks the service-related registry entries.
- There is no support for detecting a dead target server and recovering.
- The service does not shutdown gracefully under a load.
- The method timing and CPU load algorithms are not guarenteed to be stable.
- I'm not confident that horrible race conditions don't exist in service startup and shutdown (although it seems pretty stable).
- No support for per-class algorithms or host lists.
- No support for dynamically updating host, class, or algorithm settings.
- No support for pausing and continuing.
- There are bound to be a lot of other things I haven't even thought of.
- And there are essentially no comments.

If you have questions or comments, send me [mail](#). If you made it this far, you must want the code. Please

note that it is provided for your use but that all rights are reserved. [Enjoy](#).