Managing Server Resources in ArcGIS® Server .NET Applications

An ESRI® Technical Paper • October 2005

ESRI 380 New York St., Redlands, CA 92373-8100, USA • TEL 909-793-2853 • FAX 909-793-5953 • E-MAIL info@esri.com • WEB www.esri.com

J-9499

Managing Server Resources in ArcGIS Server .NET Applications

An ESRI Technical Paper

Contents **Page**

Introduction 1

The IDisposable Pattern 1

The Microsoft Garbage Collection Bug 2
Workarounds for the Microsoft Garbage Collection Bug 3
Workaround 1—Manually Calling Garbage Collection after *n* Requests 3
Workaround 2—Using ManageLifetime() 4
Workaround 3—A Combined Approach 7

Recommended Practices for ArcGIS Server Developers 7

Additional Information 8

Support 8

Managing Server Resources in ArcGIS Server .NET Applications

Introduction

Successfully managing the use and longevity of server resources within ArcGIS® Server .NET code is a primary factor in the overall success of any ArcGIS Server .NET application. It is therefore necessary for developers to be well informed about the various methods for cleaning up resources allocated by application code and to feel confident that they can generally keep their code efficient. The purpose of this paper is to explain why managing these resources is so important and provide the methods and recommended coding practices to ensure that your ArcGIS Server .NET code is as efficient as it can be. This efficiency is especially important as applications begin to experience increasing activity and traffic.

One of the main principles of good application development is to release resources as soon as you are finished using them. For example, if your application code accesses a file to perform a function, you want to close that file as soon as your code has finished using it. The same principles of good resource management will also apply to using ArcGIS Server resources in your application.

Some strategies that will be explored in this technical paper are directly available by design within the ArcGIS Server .NET Application Developer Framework (ADF) as convenient ways for developers to manage resources within their applications. Other strategies that will be given deal specifically with working around a bug in the Microsoft® .NET framework that directly affects ArcGIS Server developers. Some workarounds for this bug have been built into the ADF, but others have not. The following section will deal with the primary convenience mechanism of the ADF, and subsequent sections will deal with workarounds to the Microsoft bug. Code samples and references to additional information will be included.

The IDisposable Pattern

When working with ArcGIS Server, your application will make a connection to a server, obtain a server context, and use it to access resources on the server. The obtained server context must be released promptly so that it is available to other applications that might need to use it.

The ArcGIS Server .NET ADF provides the "IDisposable" pattern to facilitate the prompt release of server contexts. Implementing the IDisposable pattern within your own code is very easy. Simply encapsulate the functional ArcGIS Server code within using{} blocks in C#, or Try/Catch/Finally{} (TCF) blocks in VB .NET. Some examples follow.

C#

```
using(WebMap mymap = Map1.CreateWebMap())
```

```
{
mymap.NorthArrow(10,10,mymap.ImageDescriptor.ImageFormat,24,14);
Map1.Refresh();
}
```

VB .NET

```
Dim webMap As WebMap = Map1.CreateWebMap()
Try
        webMap.DrawFullExtent()
Catch ex As Exception
callErrorPage("Error Retrieving Map.", ex)
Finally
        webMap.Dispose()
End Try
```

The examples above illustrate the use of the IDisposable pattern. The WebMap ADF object is created for use explicitly within this block of code and is disposed of as soon as the code stops running. WebMap.Dispose() is called manually in VB .NET within the TCF{} block. Dispose() is called automatically on closing the using block in C#.

Q: How does this clean up behind the scenes?

A: The IDisposable pattern releases the server context after calling Marshal.ReleaseComObject() on the server context. Marshal.ReleaseComObject() ensures that not only the managed object but also the managed Runtime Callable Wrapper (RCW) is released, if no other processes are using it.

The IDisposable pattern is adequate for managing server resources in applications that only use the out-of-the-box functionality provided by the .NET ADF.

However, developers can also leverage more advanced ArcGIS Server functionality in their applications by directly using ArcObjects™. For instance, developers can call server-side ArcObjects through properties exposed on .NET ADF convenience classes (e.g., WebMap.Map). As we will see, this more fine-grained method of coding with ArcGIS Server exposes a bug within the Microsoft .NET framework that ArcGIS Server .NET developers are forced to work around.

The Microsoft Garbage Collection Bug

One of the main features of the Microsoft .NET framework is the inclusion of Garbage Collection (GC). Garbage Collection can be a powerful tool for keeping server resources clear and generally helping to clean up after .NET applications. Though GC has many advantages, it currently has a bug that affects DCOM objects in ASP .NET applications. Since ArcGIS Server is highly dependent on DCOM communication, this bug affects all aspects of using fine-grained ArcObjects in ArcGIS Server with .NET.

The bug causes COM objects that are not explicitly released with calls to Marshal.ReleaseComObject() or Garbage Collection to accumulate on the DCOM server. The server periodically tries to ping clients to see whether the COM objects should be

kept alive. As the number of unreleased objects increases, the pings take longer and longer and eventually incapacitate the server.

See the Additional Information section at the end of this technical brief for additional information on this bug. Microsoft has released a hot fix (#888000) that solves this bug on Windows® 2000 Server. In the documentation for the hot fix, this issue is listed as "DCOM servers may start to respond slowly and eventually become unusable as unreleased COM objects accumulate on the server."

In internal ESRI test results, the problem does appear to be fixed on Windows 2000 Server when the hot fixes are applied. However, testing has also revealed that this issue has *not* been addressed on Windows 2003 Server and Windows XP. For development on Windows XP or final production on Windows 2003 Server, ESRI provides the following workarounds to this bug.

Workarounds for the Microsoft Garbage Collection Bug

Workaround 1—Manually Calling Garbage Collection after *n* Requests

ArcGIS Server developers have the option of setting up manual Garbage Collection in the Global.asax file. To do this, you will want to track the number of requests for an application, then trigger GC.Collect() followed by GC.WaitForPendingFinalizers() after every *n* request. Following is an example that makes the call every five requests (C#).

```
public class Global : System.Web.HttpApplication
{
static private int requests = 0;

//Standard code from the Global.asax file here…

protected void Application_EndRequest(Object sender, EventArgs e)

//This method is provided by default in the Global.asax file.

{
if (++requests == 5)
{
GC.Collect();
GC.WaitForPendingFinalizers();
requests = 0;
}
}

//Standard code from the Global.asax file here…

}
```

As a result of implementing this code, Garbage Collection will be manually invoked immediately at the conclusion of every fifth request to this application. This provides added insurance for developers by effectively adding a safety net that releases all unreferenced managed and COM objects periodically.

Workaround 2—Using ManageLifetime()

The Global.asax implementation for Garbage Collection is a general way to address this bug. However, there are cases in which code may create a large number of COM objects within a code loop or by iterating through a set of returned information (using feature cursors, for example). In this case, a large number of COM objects are created by the code within a single request, and the above solution will not invoke Garbage Collection in a manner that fits within this scenario. In this case, it is up to the developer to manually manage the COM objects in his/her code. One way to do this is to manually release COM objects by calling Marshal.ReleaseComObject(). ESRI has provided a convenience mechanism—in the form of the ManageLifetime() method—as a more convenient way for developers to invoke Marshal.ReleaseComObject() on COM objects once they are no longer in use.

See appendix D of the *ArcGIS Server Administrator and Developer Guide* for basic information about interoperating with COM objects in the .NET framework. Information presented in this section is designed to complement the information in the guide.

General Rule: Call ManageLifetime() for every COM object instance that is created either explicitly or implicitly within your application code.

To clarify the general rule, look at an example in which two objects are *explicitly* created from the ESRI Display library for use in an ASP .NET Web application (C#).

```
using (WebMap webMap = Map1.CreateWebMap())
{

IServerContext m_ctx = webMap.ServerContext;

ESRI.ArcGIS.Display.IRgbColor pRGB =
m_ctx.CreateObject("esriDisplay.RgbColor") as ESRI.ArcGIS.Display.IRgbColor;

ESRI.ArcGIS.Display.IAlgorithmicColorRamp pRamp =
m_ctx.CreateObject("esriDisplay.AlgorithmicColorRamp") as
ESRI.ArcGIS.Display.IAlgorithmicColorRamp;

webMap.ManageLifetime(pRGB);
webMap.ManageLifetime(pRamp);

}
```

In this case, we have created two new COM objects from the server context m_ctx (pRGB and pRamp, respectively) using the CreateObject() method. Since we have called ManageLifetime() on these COM objects, they will be released when WebMap.Dispose() is called at the end of the using{} block.

Q: Why do I still need to use ManageLifetime() when my code implements the IDisposable pattern?

A: The IDisposable pattern is used for explicitly managing the object you create within the using() declaration (C#) or dim statement in a Dim/Try/Catch/Finally block (VB .NET). By using ManageLifetime() on all COM objects you create within the block, you add those objects to the list of things that are cleaned up when the using{} or TCF block closes. As such, when Marshal.ReleaseComObject() is called at the end of the block, both the object in the using() declaration (or dim statement) and the objects passed to ManageLifetime() will be disposed of. If you do not use ManageLifetime() within the block, the IDisposable pattern will only call Marshal.ReleaseComObject() for the server context used by the declaration object and for the COM objects used internally by the ADF, not for the COM objects your code created.

Q: What if I use several COM objects through a single interface? Do I need to call ManageLifetime() for each COM object?

A: Yes. You need to call ManageLifetime() once for each COM object that your code creates.

As mentioned previously, using ManageLifetime() in this way is especially important if you want to use ArcObjects code to iterate through a set of values or features (such as using IFeatureCursor). Using cursors can implicitly create a large number of COM objects and can lock access to features if they are not properly released. The use of insert cursors is a good example of this. If an insert cursor is created but not released, the cursor will block any other users from getting an insert cursor until it is released. See appendix D of the *ArcGIS Server Administrator and Developer Guide* for additional information about using ManageLifetime() with geodatabase feature cursors.

An example follows that performs a simple use of a cursor to iterate through the features of a map layer, filtered by the current map extent. Once the features are found, the cursor is used to get the NAME field for all returned features. The names are then delivered to a list box in the application (C#). See the code comments for details.

```
using (WebMap wm = Map1.CreateWebMap())
{
//Clear the list box and get information about the map.
ListBox1.Items.Clear();
IMapServer ms = wm.MapServer;
IServerContext sc = wm.ServerContext;
IMapServerObjects mo = ms as IMapServerObjects;

//Get the map and the first layer in the collection.
IMap map = mo.get_Map(wm.DataFrame);
ILayer lyr = map.get_Layer(0);
IFeatureLayer fl = lyr as IFeatureLayer;
IFeatureClass fc = fl.FeatureClass;
```

```
//To be completely safe and guarantee the release of resources, use
// ManageLifetime() on these COM objects created implicitly from fine-grained
//calls to ArcObjects.

wm.ManageLifetime(map);
wm.ManageLifetime(lyr);
wm.ManageLifetime(fl);
wm.ManageLifetime(fc);

//Create a spatial filter in the server context.
ISpatialFilter sf = sc.CreateObject("esriGeoDatabase.SpatialFilter") as
ISpatialFilter;

//Use ManageLifetime() on the object explicitly created in the context.
wm.ManageLifetime(sf);

//Set the geometry of the spatial filter to be the current map extent.
sf.GeometryField = fc.ShapeFieldName;
sf.SpatialRel = esriSpatialRelEnum.esriSpatialRelIntersects;
sf.Geometry = wm.MapDescription.MapArea.Extent;

//Use the cursor to store all features returned by a search on the current extent.
IFeatureCursor fcur = fc.Search(sf,true);

//Use ManageLifetime() on the feature cursor.
//This is necessary because the feature cursor is created implicitly
//as the result of the fc.Search() method.
wm.ManageLifetime(fcur);

int fidx = fc.FindField("NAME");
IFeature f = null;

//Iterate through all the features, and add the NAME attribute to the list box.
while ((f = fcur.NextFeature()) != null)
  {

     //Use ManageLifetime() on the feature for every iteration.
     //This is necessary because the feature COM object is created separately
     // from the feature cursor, and a new COM object is created for each
     //returned feature.  Calling ManageLifetime() within the loop ensures
     //that all individual feature COM objects will be cleaned up.
     wm.ManageLifetime(f);

     ListBox1.Items.Add(f.get_Value(fidx).ToString());
  }

}
```

This is a good example of the general rule for ManageLifetime() in action. The code uses ManageLifetime() both on new objects created explicitly within the server context as well as on objects that are created implicitly by fine-grained use of ArcObjects. In the above example, ManageLifetime() is used on the spatial filter that is created with the CreateObject() method; on the ArcObjects calls for the map, layer, and feature class; on the feature cursor that is returned by the FeatureClass.Search() method; and finally on the feature object that was used to contain the currently selected record.

Note that there is no need to use ManageLifetime() on the objects that are specific to the connection to ArcGIS Server, the server context, and the server object (ms, sc, mo). The life cycle of this connection information is handled automatically by ArcGIS Server. In this sample, the code is accessing a pooled server object, so the context will be released as a result of using the IDisposable pattern. If the code was instead accessing a nonpooled server object, it would be necessary to call ReleaseContext on the server context when the code has finished with it. In addition, Marshal.ReleaseComObject() would also have to be called for the server context. See chapter 5 of the *ArcGIS Server Administrator and Developer Guide* for more information about managing the necessary connections for nonpooled server objects.

Workaround 3—A Combined Approach

Note that all ArcGIS Server developers can use a combination of the IDisposable pattern, ManageLifetime(), and the Global.asax solutions within a single application. This is considered by ESRI to be a complete ArcGIS Server .NET programming model with respect to managing server resources in Web applications and services.

Recommended Practices for ArcGIS Server Developers

ESRI officially recommends the third workaround—a combined approach—to all developers of .NET Web applications and services using ArcGIS Server. To fully understand this coding pattern, it is useful to review the recommended coding practices that have been covered thus far.

1. IDisposable pattern should be used to manage access to server contexts through the .NET ADF controls and convenience classes. To work around a bug in the Microsoft .NET framework, additional coding strategies must also be used.

2. Use the Global.asax code solution to periodically clean up any other remote objects created by your application after a set number of requests.

3. ManageLifetime() should be used on all explicitly or implicitly created COM objects (fine-grained calls to ArcObjects, iterating through collections, etc.) that were created outside the IDisposable declaration.

This complete solution offers a high degree of assurance that your applications will properly release all server resources, and will run efficiently and trouble-free. For applications running exclusively on Windows 2000 Server, steps 2 and 3 are not required. For applications that will have to run on Windows 2003 Server and Windows XP Professional, either immediately or in the future, all three recommendations are required until Microsoft makes a fix available for the DCOM bug on these platforms.

Additional Information

Microsoft Knowledge Base
http://support.microsoft.com/?kbid=888000
http://support.microsoft.com/?kbid=890340

Support

Information about ArcGIS Server and the .NET ADF can be found online at http://www.esri.com/arcgisserver. For answers to GIS capacity planning and solution questions, contact ESRI Systems Integration at sihelp@esri.com. For technical support, contact ESRI Technical Support at http://support.esri.com.