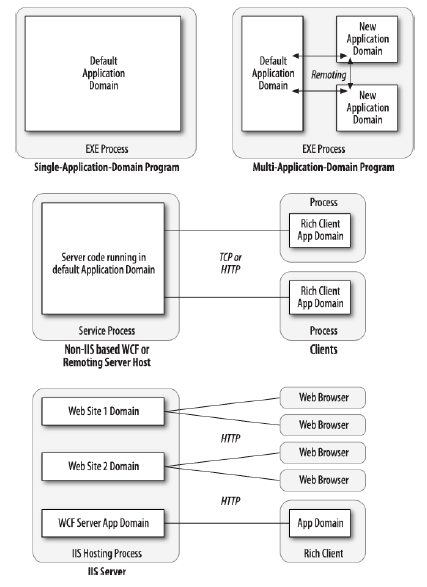
**Application Domain**

An *application domain* is the runtime unit of isolation in which a .NET program runs. It provides a managed memory boundary, a container for loaded assemblies and application configuration settings, as well as delineating a communication boundary for distributed applications. Each .NET process usually hosts just one application domain: the default domain, created automatically by the CLR when the process starts. It’s also possible—and sometimes useful—to create additional application domains within the same process. This provides isolation while avoiding the overhead and communication complications that arise with having separate processes. It’s useful in scenarios such as load testing and application patching, and in implementing robust error recovery

mechanisms.

Application Domain Architecture

Figure illustrates the application domain architectures for single-domain, multidomain, and typical distributed client/server applications. In most cases, the processes housing the application domains are created implicitly by the operating system—when the user double-clicks your .NET executable file or starts a Windows service. However, an application domain can also be hosted in other processes such as IIS or in SQL Server through CLR integration. In the case of a simple executable, the process ends when the default application domain finishes executing. With hosts such as IIS or SQL Server, however, the process controls the lifetime, creating and destroying .NET application domains as it sees fit.

****

Creating and Destroying Application Domains

You can create and destroy additional application domains in a process by calling the static methods AppDomain.CreateDomain and AppDomain.Unload. In the following example, *test.exe* is executed in an isolated application domain, which is then unloaded:

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

newDomain.ExecuteAssembly ("test.exe");

AppDomain.Unload (newDomain);

}

Note that when the default application domain (the one created by the CLR at startup) is unloaded, all other application domains automatically unload, and the application closes. A domain can “know” whether it’s the default domain via the AppDomain property IsDefaultDomain.

The AppDomainSetup class allows options to be specified for a new domain. The following properties are the most useful:

public string ApplicationName { get; set; } // "Friendly" name

public string ApplicationBase { get; set; } // Base folder

public string ConfigurationFile { get; set; }

public string LicenseFile { get; set; }

// To assist with automatic assembly resolution:

public string PrivateBinPath { get; set; }

public string PrivateBinPathProbe { get; set; }

The ApplicationBase property controls the application domain base directory, used

as the root for automatic assembly probing. In the default application domain, this

is the main executable’s folder. In a new domain that you create, it can be anywhere

you like:

AppDomainSetup setup = new AppDomainSetup();

setup.ApplicationBase = @"c:\MyBaseFolder";

AppDomain newDomain = AppDomain.CreateDomain ("New Domain", null, setup);

It’s also possible to subscribe a new domain to assembly resolution events defined

in the instigator’s domain:

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("test");

newDomain.AssemblyResolve += new ResolveEventHandler (FindAssem);

...

}

static Assembly FindAssem (object sender, ResolveEventArgs args)

{

...

}

This is acceptable providing the event handler is a static method defined in a type available to both domains. The CLR is then able to execute the event handler in the correct domain. In this example, FindAssem would execute from within newDomain, even though it was subscribed from the default domain.

The PrivateBinPath property is a semicolon-separated list of subdirectories below the base directory that the CLR should automatically search for assemblies. (As with the application base folder, this can only be set prior to the application domain starting.)

Take, for example, a directory structure where a program has, in its base folder, a single executable (and perhaps a configuration file) and all the referenced assemblies in subfolders as follows:

c:\MyBaseFolder\ -- Startup executable

\bin

\bin\v1.23 -- Latest assembly DLLs

\bin\plugins -- More DLLs

Here’s how an application domain would be set up to use this folder structure:

AppDomainSetup setup = new AppDomainSetup();

setup.ApplicationBase = @"c:\MyBaseFolder";

setup.PrivateBinPath = @"bin\v1.23;bin\plugins";

AppDomain d = AppDomain.CreateDomain ("New Domain", null, setup);

d.ExecuteAssembly (@"c:\MyBaseFolder\Startup.exe");

Note that PrivateBinPath is always relative to, and below, the application base folder. Specifying absolute paths is illegal. AppDomain also provides a PrivateBin PathProbe property, which, if set to anything other than a blank string, excludes the base directory itself from being part of the assembly search path. (The reason PrivateBinPathProbe is a string rather than a bool type relates to COM compatibility.) Just before any nondefault application domain unloads, the DomainUnload event fires. You can use this event for tear-down logic: the unloading of the domain (and the application as a whole, if necessary) is delayed until the execution of all DomainUn load event handlers completes. Just before the application itself closes, the ProcessExit event fires on all loaded application domains (including the default domain). Unlike with the DomainUnload event, ProcessExit event handlers are timed: the default CLR host gives event handlers two seconds per domain, and three seconds in total, before terminating their

threads.

Using Multiple Application Domains

Multiple application domains have the following key uses:

• Providing process-like isolation with minimum overhead

• Allowing assembly files to be unloaded without restarting the process

When additional application domains are created within the same process, the CLR provides each with a level of isolation akin to that of running in separate processes. This means that each domain has separate memory, and objects in one domain cannot interfere with those in another. Furthermore, static members of the same class have independent values in each domain. ASP.NET uses exactly this approach to allow many sites to run in a shared process without affecting each other. With ASP.NET, the application domains are created by the infrastructure—without

your intervention. There are times, however, when you can benefit from explicitly creating multiple domains inside a single process. Suppose you’ve written a custom authentication system, and as part of unit testing, you want to stress-test the server code by simulating 20 clients logging in at once. You have three options in simulating 20 concurrent logins:

• Start 20 separate processes by calling Process.Start 20 times.

• Start 20 threads in the same process and domain.

• Start 20 threads in the same process—each in its own application domain.

The first option is clumsy and resource-intensive. It’s also hard to communicate with each of the separate processes, should you want to give them more specific instructions on what to do.

The second option relies on the client-side code being thread-safe, which is unlikely especially if static variables are used to store the current authentication state. And adding a lock around the client-side code would prevent the parallel execution that we need to stress-test the server.

The third option is ideal. It keeps each thread isolated with independent state and yet within easy reach of the hosting program. Another reason to create a separate application domain is to allow assemblies to be unloaded without ending the process. This stems from the fact that there’s no way

to unload an assembly other than closing the application domain that loaded it. This is a problem if it was loaded in the default domain, because closing this domain means closing the application. An assembly’s file is locked while loaded and so cannot be patched or replaced. Loading assemblies in a separate application domain that can be torn down gets around this problem—as well as helping to reduce the memory footprint of an application that occasionally needs to load large assemblies.

The LoaderOptimization Attribute

By default, assemblies that load into an explicitly created application domain are reprocessed by the JIT compiler. This includes:

• Assemblies that have already been JIT-compiled in the caller’s domain

• Assemblies for which a native image has been generated with the *ngen.exe* tool

• All of the .NET Framework assemblies (except for *mscorlib*)

This can be a major performance hit, particularly if you repeatedly create and unload application domains that reference large .NET Framework assemblies. A workaround is to attach the following attribute to your program’s main entry method:

[LoaderOptimization (LoaderOptimization.MultiDomainHost)]

This instructs the CLR to load GAC assemblies *domain-neutral*, so native images are honored and JIT images shared across application domains. This is usually ideal, because the GAC includes all .NET Framework assemblies (and possibly some invariant parts of your application).

You can go a stage further by specifying LoaderOptimization.MultiDomain: this instructs *all* assemblies to be loaded domain-neutral (excluding those loaded outside the normal assembly resolution mechanism). This is undesirable, however, if you want assemblies to unload with their domain. A domain-neutral assembly is shared between all domains and so does not unload until the parent process ends.

Using DoCallBack

Let’s revisit the most basic multidomain scenario:

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

newDomain.ExecuteAssembly ("test.exe");

AppDomain.Unload (newDomain);

}

Calling ExecuteAssembly on a separate domain is convenient but offers little opportunity to interact with the domain. It also requires that the target assembly is an executable, and it commits the caller to a single entry point. The only way to incorporate flexibility is to resort to an approach such as passing a string of arguments to the executable.

A more powerful approach is to use AppDomain’s DoCallBack method. This executes on another application domain, a method on a given type. The type’s assembly is automatically loaded into the domain (the CLR will know where it lives if the current domain can reference it). In the following example, a method in the currently executing class is run in a new domain:

class Program

{

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

newDomain.DoCallBack (new CrossAppDomainDelegate (SayHello));

AppDomain.Unload (newDomain);

}

static void SayHello()

{

Console.WriteLine ("Hi from " + AppDomain.CurrentDomain.FriendlyName);

}

}

The example works because the delegate is referencing a static method, meaning it points to a type rather than an instance. This makes the delegate “domain-agnostic” or agile. It can run in any domain, and in the same way, as there’s nothing tying it to the original domain. It’s also possible to use DoCallBack with a delegate referencing an instance method. However, the CLR will attempt to apply Remoting semantics (described later), which in this case happens to be the opposite of what we want.

Monitoring Application Domains

From Framework 4.0, you can monitor the memory and CPU consumption of a

specific application domain. For this to work, you must first enable application domain

monitoring as follows:

AppDomain.MonitoringIsEnabled = true;

This enables monitoring for the current domain. Once enabled, you can’t subsequently

disable it—setting this property to false throws an exception.

You can then query an AppDomain’s CPU and memory usage via the following three

instance properties:

MonitoringTotalProcessorTime

MonitoringTotalAllocatedMemorySize

MonitoringSurvivedMemorySize

The first two properties return the *total* CPU consumption and managed memory allocated by that domain since it was started. (These figures can only grow and never shrink). The third property returns the actual managed memory consumption of the domain at the time of the last garbage collection. You can access these properties from the same or another domain.

Domains and Threads

When you call a method in another application domain, execution blocks until the method finishes executing—just as though you called a method in your own domain. Although this behavior is usually desirable, there are times when you need to run a method concurrently. You can do that with multithreading. We talked previously about using multiple application domains to simulate 20 concurrent client logins in order to test an authentication system. By having each client log in on a separate application domain, each would be isolated and unable to interfere

with another client via static class members. To implement this example, we need to call a “Login” method on 20 concurrent threads, each in its own application domain:

class Program

{ static void Main()

{

// Create 20 domains and 20 threads.

AppDomain[] domains = new AppDomain [20];

Thread[] threads = new Thread [20];

for (int i = 0; i < 20; i++)

{

domains [i] = AppDomain.CreateDomain ("Client Login " + i);

threads [i] = new Thread (LoginOtherDomain);

}

// Start all the threads, passing to each thread its app domain.

for (int i = 0; i < 20; i++) threads [i].Start (domains [i]);

// Wait for the threads to finish

for (int i = 0; i < 20; i++) threads [i].Join();

// Unload the app domains

for (int i = 0; i < 20; i++) AppDomain.Unload (domains [i]);

Console.ReadLine();

}

// Parameterized thread start - taking the domain on which to run.

static void LoginOtherDomain (object domain)

{

((AppDomain) domain).DoCallBack (Login);

}

static void Login()

{

Client.Login ("Joe", "");

Console.WriteLine ("Logged in as: " + Client.CurrentUser + " on " +

AppDomain.CurrentDomain.FriendlyName);

}

}

class Client

{

// Here's a static field that would interfere with other client logins

// if running in the same app domain.

public static string CurrentUser = "";

public static void Login (string name, string password)

{

if (CurrentUser.Length == 0) // If we're not already logged in...

{

// Sleep to simulate authentication...

Thread.Sleep (500);

CurrentUser = name; // Record that we're authenticated.

}

}

}

// Output:

Logged in as: Joe on Client Login 0

Logged in as: Joe on Client Login 1

Logged in as: Joe on Client Login 4

Logged in as: Joe on Client Login 2

Logged in as: Joe on Client Login 3

Logged in as: Joe on Client Login 5

Logged in as: Joe on Client Login 6

Sharing Data Between Domains

Sharing Data via Slots

Application domains can use named slots to share data, as in the following example:

class Program

{

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

// Write to a named slot called "Message" - any string key will do.

newDomain.SetData ("Message", "guess what...");

newDomain.DoCallBack (SayMessage);

AppDomain.Unload (newDomain);

}

static void SayMessage()

{

// Read from the "Message" data slot

Console.WriteLine (AppDomain.CurrentDomain.GetData ("Message"));

}

}

// Output:

guess what...

A slot is created automatically the first time it’s used. The data being communicated

(in this example, "guess what ...") must either be *serializable*, or be based on MarshalByRefObject. If the data is serializable (such as the string in our example), it’s copied to the other application domain. If it implements MarshalByRefObject, Remoting semantics are applied.

Intra-Process Remoting

The most flexible way to communicate with another application domain is to instantiate objects *in the other domain* via a proxy. This is called *Remoting*. The class being “Remoted” must inherit from MarshalByRefObject. The client then calls a CreateInstance*XXX* method on the remote domain’s AppDomain class to remotely instantiate the object.

The following instantiates the type Foo in another application domain, and then calls

its SayHello method:

class Program

{

static void Main()

{

AppDomain newDomain = AppDomain.CreateDomain ("New Domain");

**Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (**

**typeof (Foo).Assembly.FullName,**

**typeof (Foo).FullName);**

Console.WriteLine (foo.SayHello());

AppDomain.Unload (newDomain);

Console.ReadLine();

}

}

public class Foo : **MarshalByRefObject**

{

public string SayHello()

{

return "Hello from " + AppDomain.CurrentDomain.FriendlyName;

}

public override object InitializeLifetimeService()

{

// This ensures the object lasts for as long as the client wants it

return null;

}

}

When the foo object is created on the other application domain (called the “remote” domain), we don’t get back a direct reference to the object, because the application domains are isolated. Instead, we get back a transparent proxy; transparent because it *appears* as though it was a direct reference to the remote object. When we subsequently call the SayHello method on foo, a message is constructed behind the scenes, which is forwarded to the “remote” application domain where it is then executed on the real foo. Rather like saying “hello” on a telephone: you’re talking not to a real person but to a piece of plastic that acts as a transparent proxy for a person. Any return value is turned into a message and sent back to the caller. The methods on Foo can return more MarshalByRefObject instances, in which case more transparent proxies are generated when those methods are called. Methods on Foo can also accept MarshalByRefObject instances as arguments—in which Remoting happens in reverse. The caller will hold the “remote” object, while the callee will have a proxy.

As well as marshaling objects by reference, application domains can exchange scalar values, or any *serializable* object. A type is serializable if it either has the Serializable attribute or implements ISerializable. Then, when crossing the application domain boundary, a complete copy of the object is returned, rather than a proxy. In other words, the object is marshaled by *value* rather than reference. Remoting within the same process is client-activated, meaning that the CLR doesn’t attempt to share or reuse remotely created objects with the same or other clients. In other words, if the client creates two Foo objects, two objects will be created in the

remote domain, and two proxies in the client domain. This provides the most natural object semantics; however, it means that the remote domain is dependent on the client’s garbage collector: the foo object in the remote domain is released from memory only when the client’s garbage collector decides that the foo (proxy) is no longer in use. If the client domain crashes, it may never get released. To protect against this scenario, the CLR provides a lease-based mechanism for managing the lifetime of remotely created objects. The default behavior is for remotely created objects to self-destruct after five minutes of nonuse. Because in this example the client runs in the default application domain, the client doesn’t have the luxury of crashing. Once it ends, so does the whole process! Hence, it makes sense to disable the five-minute lifetime lease. This is the purpose of overriding InitializeLifetimeService—by returning a null lease, remotely created objects are destroyed only when garbage-collected by the client.

Isolating Types and Assemblies

In the preceding example, we remotely instantiated an object of type Foo as follows:

Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (

typeof (Foo).Assembly.FullName,

typeof (Foo).FullName);

Here’s the method’s signature:

public object CreateInstanceAndUnwrap (string assemblyName,

string typeName)

Because this method accepts an assembly and type *name* rather than a Type object, you can remotely instantiate an object without loading its type locally. This is useful when you want to avoid loading the type’s assembly into the caller’s application domain. To illustrate, suppose we were writing a text editor that allows the user to load and unload third-party plug-ins. When it came to actually executing the plug-in, however, all we did was call ExecuteAssembly. With Remoting, we can interact with plug-ins in a richer fashion. The first step is to write a common library that both the host and the plug-ins will reference. This library will define an interface describing what plug-ins can do. Here’s a simple example:

namespace Plugin.Common

{

public interface ITextPlugin

{

string TransformText (string input);

}

}

Next, we need to write a simple plug-in. We’ll assume the following is compiled to

*AllCapitals.dll*:

namespace Plugin.Extensions

{

public class AllCapitals : MarshalByRefObject, **Plugin.Common.ITextPlugin**

{

public string TransformText (string input) { return input.ToUpper(); }

}

}

Here’s how to write a host that loads *AllCapitals.dll* into a separate application

domain, calls TransformText using Remoting, and then unloads the application

domain:

using System;

using System.Reflection;

using Plugin.Common;

class Program

{

static void Main()

{

AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

**ITextPlugin plugin = (ITextPlugin) domain.CreateInstanceFromAndUnwrap**

(**"AllCapitals.dll", "Plugin.Extensions.AllCapitals");**

// Call the TransformText method using Remoting:

Console.WriteLine (**plugin.TransformText ("hello")**); // "HELLO"

AppDomain.Unload (domain);

// The AllCapitals.dll file is now completely unloaded and could

// be moved or deleted.

}

}

Because this program interacts with the plug-in solely through the common interface, ITextPlugin, the types in AllCapitals are never loaded into the caller’s application domain. This maintains the integrity of the caller’s domain and ensures that no locks are held on the plug-in assembly files after their domain is unloaded.

Type discovery

In our preceding example, a real application would need some means of discovering

plug-in type names, such as Plugin.Extensions.AllCapitals.

You can achieve this by writing a discovery class in the *common* assembly that uses

reflection as follows:

public class Discoverer : MarshalByRefObject

{

public string[] GetPluginTypeNames (string assemblyPath)

{

List<string> typeNames = new List<string>();

Assembly a = Assembly.LoadFrom (assemblyPath);

foreach (Type t in a.GetTypes())

if (**t.IsPublic**

**&& t.IsMarshalByRef**

**&& typeof (ITextPlugin).IsAssignableFrom (t)**)

{

typeNames.Add (t.FullName);

}

return typeNames.ToArray();

}

}

The catch is that Assembly.LoadFrom loads the assembly into the current application domain. Therefore, you must call this method *in the plug-in domain*:

class Program

{

static void Main()

{

AppDomain domain = AppDomain.CreateDomain ("Plugin Domain");

**Discoverer d = (Discoverer) domain.CreateInstanceAndUnwrap (**

**typeof (Discoverer).Assembly.FullName,**

**typeof (Discoverer).FullName);**

**string[] plugInTypeNames = d.GetPluginTypeNames ("AllCapitals.dll");**

foreach (string s in plugInTypeNames)

Console.WriteLine (s); // Plugin.Extensions.AllCapitals

**Questions**

1. What is application domain?
2. How can application domains share the process resources?
3. How can application domains create interprocess communications?