

: .NET Remoting Overview

Presented by:

Kirk Fertitta
Chief Technical Officer
Pacific MindWorks

APPDOMAIN BASICS

In Windows, the operating system (OS) provides native objects known as *processes* to isolate executing code. Processes act as useful boundaries between code – they can be independently stopped and started, they are independently named, they can be independently debugged, they can be assigned independent security credentials, etc. The chief disadvantage of native OS processes is that they are heavy weight.

.NET introduces the AppDomain as a means to provide many of the same code isolation features as OS processes, but without the memory or CPU overhead. AppDomains provide an execution scope for all code that executes in the .NET Common Language Runtime (CLR). Every instance of a class in .NET belongs to exactly one AppDomain. As with processes, code executing in one AppDomain cannot (directly) access code executing in another AppDomain. AppDomains can be independently started, stopped, and debugged, and can operate with their own specific security credentials (as well as a host of other execution control settings that OS processes do not have).

Structurally, AppDomains exist within a physical OS process. A single OS process can host multiple AppDomains. When a .NET executable runs, an OS process is created and a default AppDomain is created. Additional AppDomains may be created explicitly by the application code (just as Win32 programmers could call `CreateProcess` in the “old” days), or they may be created by the CLR host (as with the ASP.NET runtime). Whether two .NET objects exist in two different AppDomains *within* a process or two different AppDomains *between* processes, .NET requires you to take the same explicit measures to allow those two classes to communicate. The infrastructure .NET provides for communicating between AppDomains is called *remoting*.

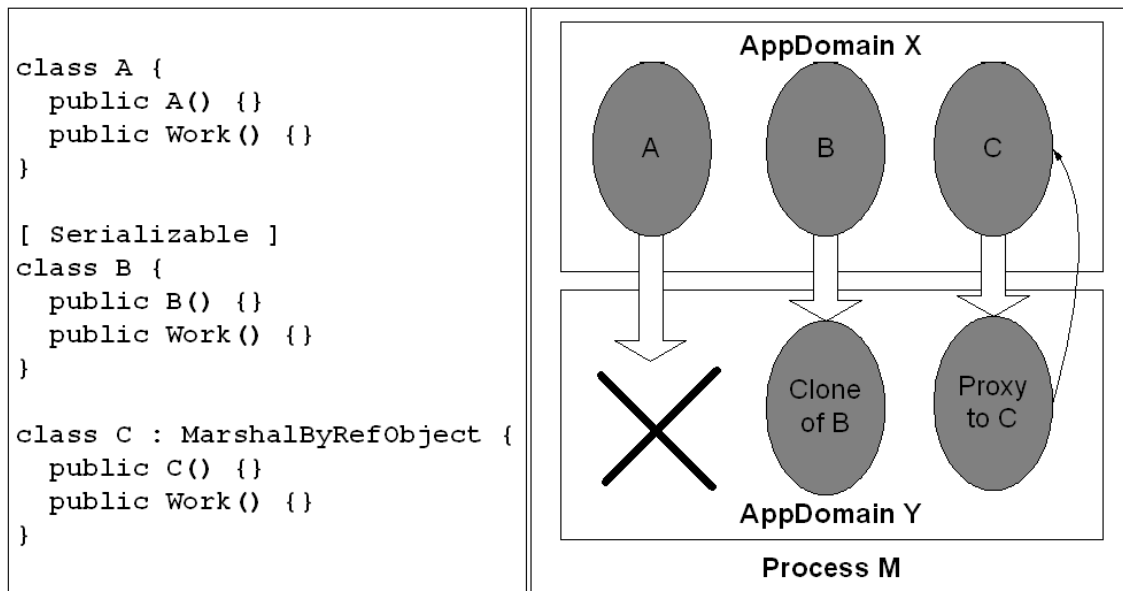
REMOVING BASICS

Remoting allows code in one AppDomain to access code in a remote AppDomain, whether the remote AppDomain exists within the same process, within different processes on the same machine, or within different processes on different networked machines. The most important thing to understand about remoting in .NET is that, by default, objects *are not* remotable. Rather, .NET requires the developer to take explicit action to instruct the CLR how instances of particular classes should be transported when performing cross-AppDomain operations.

Developers have three choices for enabling remoting in .NET – 1) no remoting allowed, 2) serializable (marshal-by-value), or 3) marshal-by-reference. To disallow remoting of a class, the developer literally does nothing. By default, all .NET classes are not remotable. If a developer wishes to have marshal-by-value semantics for the class, the class must be marked with the `[Serializable]` attribute. Finally, mar-

shal-by-reference remoting behavior is enabled by deriving the class (directly or indirectly) from the standard .NET class MarshalByRefObject. Note that classes that are marked with the [Serializable] attribute and that derive from MarshalByRefObject obey marshal-by-reference semantics – (e.g. MarshalByRefObject takes precedence).

The three different options for remoting are depicted in the figure below:



The figure also indicates how each remoting option differs in behavior at runtime. Class A is not remotable, so an exception will be thrown at runtime if code in AppDomain Y attempts to access instances of class A. Class B is marked as being serializable, so instances will be marshaled by value to remote AppDomains. AppDomain Y will receive a disconnected clone of any instance of class B. Class C derives from MarshalByRefObject, so code in the remote AppDomain will access instances of C through a proxy object. More details of [Serializable] and MarshalByRefObject types are provided in the following sections.

UNDERSTANDING [SERIALIZABLE] TYPES

The CLR provides a very flexible and powerful serialization engine that is enabled for a type whenever that type is marked with the [Serializable] attribute. When a serializable type is remoted, the CLR takes the instance state (member variables), serializes them into a special remoting packet, transports them to the remote AppDomain, and “rehydrates” a disconnected clone of the original object in the remote AppDomain. One could accurately think of a remoted [Serializable] object as a “snapshot” of the state of the original object.

Methods executed against the clone in the remote AppDomain always execute within that AppDomain and are never sent back across the wire to the original AppDomain. This makes these method calls very, very efficient. A classic example of where [Serializable] types would make sense is in classes that represent large, read-only database results. If, for example, a method on a database returned a result set that was only intended to be displayed on a web page, it is much more efficient to make that result set object [Serializable], so that each operation to retrieve elements in the result set wouldn't incur a round-trip across AppDomain boundaries. Indeed, this is how the venerable disconnected ADO recordset operates (albeit with COM instead of .NET). Any changes made to the result set would have no effect on the original database.

Without going into the gory details, it should be noted that the CLR does provide a few different ways of customizing the serialization process that occurs whenever a [Serializable] type is remotized. Developers can optimize the serialization process by taking either partial or complete control of the CLR serialization engine for a specific type.

UNDERSTANDING MARSHALBYREFOBJECT TYPES

Types that derive from MarshalByRefObject will use *proxies* to communicate with remote AppDomains. Whenever an object in a remote AppDomain attempts to access a MarshalByRefObject instance, the remote object will actually be talking to a proxy object that serves as the "local representative" of the original MarshalByRefObject instance. Method calls from the remote AppDomain are packaged into special *message objects* that contain all of the information about the method call, such as the name of the method to execute and the value of the method parameters. This message object is sent across the wire to the remote AppDomain, unpacked, and executed on the original object. Any return values from the method call are packaged up into a message object and sent to the caller in the remote AppDomain.

Unlike the case with [Serializable] objects, method calls on MarshalByRefObject instances execute in the AppDomain of the original object, *not* in the AppDomain of the caller. This means that method calls on MarshalByRefObject instances are considerably slower than method calls on [Serializable] objects. However, it also means that the original object always "feels" changes that remote AppDomains make to remote instances of the object. Following with the database example from above, consider the situation where a database method returned a result set to a remote AppDomain and the remote AppDomain needed to perform both read and update operations to the database. If the result set was maintained as a MarshalByRefObject, then each method call that updated the result set would be marshaled back to the original AppDomain. Other AppDomains reading from the database would always be looking at the most up-to-date information in the database.

The remoting infrastructure for .NET is fabulously complex when MarshalByRefObjects are involved, and a thorough discussion of the topic is way, way beyond the scope of this paper. Even more than is the case with [Serializable] types, MarshalByRefObjects have numerous customization and extensibility options that allow developers to take control of as much of the remoting process as they can possibly stand.