Packaging and Deploying NGWS Applications

Tutorial

Abstract

This tutorial shows developers how to package and deploy NGWS applications and components using the PDC Tech Preview of the NGWS Software Development Kit (SDK) and Visual Studio 7.0. The development tools in this next major release of Visual Studio will utilize NGWS to allow developers to quickly build and deploy robust applications that take advantage of the new runtime environment to provide a fully managed, protected, and feature rich application execution environment. NGWS also provides:

- Improved isolation of application components
- Simplified application deployment
- Robust versioning

This tutorial walks you through packaging and deploying the classic “Hello World” program as well as a small, componentized application. These programs were written in C♯ - the new language designed for NGWS. The steps necessary to construct, compile, and run C♯ programs were detailed in a separate tutorial, *Developing with NGWS: An Introduction*. After reading this tutorial and working with the samples, you should be able to plan how to package and deploy traditional stand-alone executable NGWS applications. An appendix to this tutorial contains additional information on several useful deployment related utilities.

Note: This tutorial does not directly cover packaging and deploying Web server (ASP+) or browser-based applications. For information on Web server applications, please refer to the *Deploying ASP+ Applications* topic of the *ASP+ and Web Server Overview* section in the main NGWS Help (.chm) file

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

0200

CONTENTS

INTRODUCTION

This tutorial assumes you already have the PDC Tech Preview versions of both the NGWS SDK and the corresponding Visual Studio 7.0. If you are working with later versions of either the SDK or Visual Studio, you will need to obtain an updated version of this document since some details of the underlying technology – particularly the names of the underlying objects and their members – are likely to have changed.

We will walk you through the process of packaging and deploying two small programs that take advantage of the new NGWS environment. You will start with the simplest program: a command-line version of the traditional "Hello World" executable and then work with a small, componentized application that demonstrates several increasingly more complex deployment scenarios. These programs were written in C♯, though they could also have been written using Managed Extensions to C++ or Visual Basic

Prerequisites

Before reading this document and working with the accompanying samples, developers should first read the "NGWS Overview" (located in the complus2.chm file included with the SDK) to get a clear understanding of the NGWS architecture. Developers creating server applications should also review the "Developing with NGWS: An Introduction" tutorial.

To benefit most from this tutorial and accompanying samples, the reader should already be familiar with developing applications using COM+. Throughout this document we will be introducing new terms related to the NGWS technology: For additional material on these terms, you may wish to refer to the "NGWS Glossary" at the end of the complus2.chm file.

Finally, since the code samples are presented in C#, developers will find it helpful to be familiar with at least one of the major Microsoft programming languages.

Tools required

In general, the NGWS SDK includes everything necessary to package programs for deployment. All of the files needed to package the samples discussed in this tutorial are installed with the SDK (by default, into the C:\Program Files\NGWSSDK directory). The runtime files are located in a subdirectory below the \Windows\ComPlus subdirectory. For the PDC Tech Preview of the NGWS SDK, the important files and corresponding versions are:

- **mscorlib.dll** – 1999.13.????
- **csc.exe** – 7.0.????

Since source and project files C# are plain text, any text editor – even Notepad – is adequate for examining and modifying the accompanying sample files.

The PDC Tech Preview NGWS SDK includes a variety of tools, several of which are particularly useful to developers working with packaging and deployment: These are described in *Appendix B: Packaging And Deployment Tools*, in this document.

COMMON CONCEPTS

The sample programs that accompany this tutorial are located in the self-extracting "samples.exe" file.

Finally, to use the SDK tools you must have your **`path`** environment variable correctly configured to include (again, by default) the "C:\Program Files\NGWSSDK\Bin" subdirectory where they are stored.

A NGWS program can be deployed several ways depending on how complex the program is, the security / protection requirements, and how it is to be distributed. What doesn't matter is which NGWS runtime compatible language was used to develop the program: All programs written for use with NGWS are compiled to the same self-describing, managed intermediate language (IL) code and run against the same NGWS runtime.

Building Blocks

In addition to the runtime, NGWS also provides a common base class library organized into a hierarchical tree of *namespaces*. At the root is the System namespace, which contains objects for many other useful classes - including those for file IO, messaging, networking, and security – that can be used from any NGWS language.

NGWS class libraries that you and others create are also organized into hierarchical namespaces and are stored in portable executable (PE) files – typically DLLs and EXEs. You can have several namespaces – including nested namespaces – in one PE file, and you can split a namespace across multiple PE files. One or more PE files (and possibly non-PE files, such as resources) are combined together to create an *assembly*, which is a physical unit that can be deployed, versioned, and reused.

In NGWS, each class type is fully described through the type's *metadata*. Each assembly contains a *manifest* that includes the name of each type exported from the assembly along with information about which file contains its metadata. The manifest also includes information about the identity of the assembly (name, files make up the assembly, and version information) and full information about any of the assembly's dependencies on other assemblies. The NGWS runtime uses assembly manifests to locate and bind to the referenced types.

Deployment

In the simplest case, a stand-alone NGWS executable can simply be executed – locally – from any machine on which the NGWS runtime is already installed. Nothing else is required: No Registry entries are made, nothing can break any other application or cause it to stop running, and simply deleting the file – if copied locally – is enough to clean up the application and leave "zero footprint" on the machine. Applications run from slow-access devices – like UNC paths, CDs, or floppies (essentially anything other than a local disk) – behave only slightly differently: assemblies will be installed in the *download* cache and are later automatically scavenged.

Componentized applications are only slightly more complicated, depending on

(1) HELLO WORLD

whether the components are private to the application, shared with other related applications, or shared with other potentially unknown applications. If all the components are private, then the componentized application can be treated in the same manner as the standalone application. Similarly, if several related applications use the same assembly, then it can be located in a common subdirectory. However, if the application uses assemblies that are shared with other – undetermined - applications, then these assemblies must be installed into the system assembly cache and must have certain properties – for instance a unique name and version information – that enables the NGWS runtime to ensure that the application binds to the appropriate component versions. An important feature of NGWS applications is the ability to maintain application configuration in plain-text files: This allows administrator to tailor an application's behavior on a particular machine without having to involve developers. The examples in the following sections will walk you through the common scenarios. While we will not be looking at ASP+ deployment in this tutorial, most of the same concepts apply.

Distribution

Of course, most client applications will be further packaged into a common distribution format – such as a .CAB file or .MSI file – and many will be installed using application distribution mechanisms such as Windows 2000 IntelliMirror or Microsoft's Systems Management Server (SMS) which both use the Microsoft Installer technology. For more on the Microsoft Installer, please refer to the corresponding section in the Win32 SDK.

First, we'll take a look at the simplest NGWS program, the traditional "Hello World" program written in C# and discussed in detail in the tutorial *Developing with NGWS: An Introduction*. Here's the C♯ source code (which can be found with the code accompanying this tutorial in the 1_HelloWorld subdirectory):

This sample stand-alone executable prints a single line to the System.Console, a type contained in the NGWS base class library. It does not reference any other libraries and does not itself produce a library. To provide convenient access to types in the System library requires the **`using`** statement:

```
using System;
```

Also, we define a **`class`** to enclose the application code:

```
class MainApp {…}
```

Finally, we define a method Main to provide the entry point to our code:

```
void Main () {…}
```

Compiling this small program (the file build.bat contains the single line necessary) using:

```
csc hello.cs
```

generates the stand-alone Hello.exe. Running the intermediate language disassembler (ILDasm.exe) against this executable yields a window similar to the following:

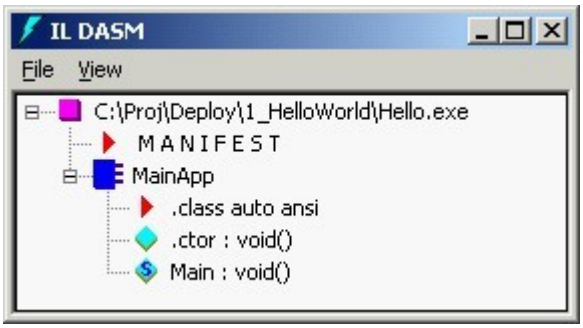


Figure ILDASM of Hello.exe

Even this simple program illustrates several important concepts behind programming for the NGWS environment. First, the program is clearly self-describing: the information necessary to understand the program is contained in the manifest. Double-clicking on the manifest line gives additional information:

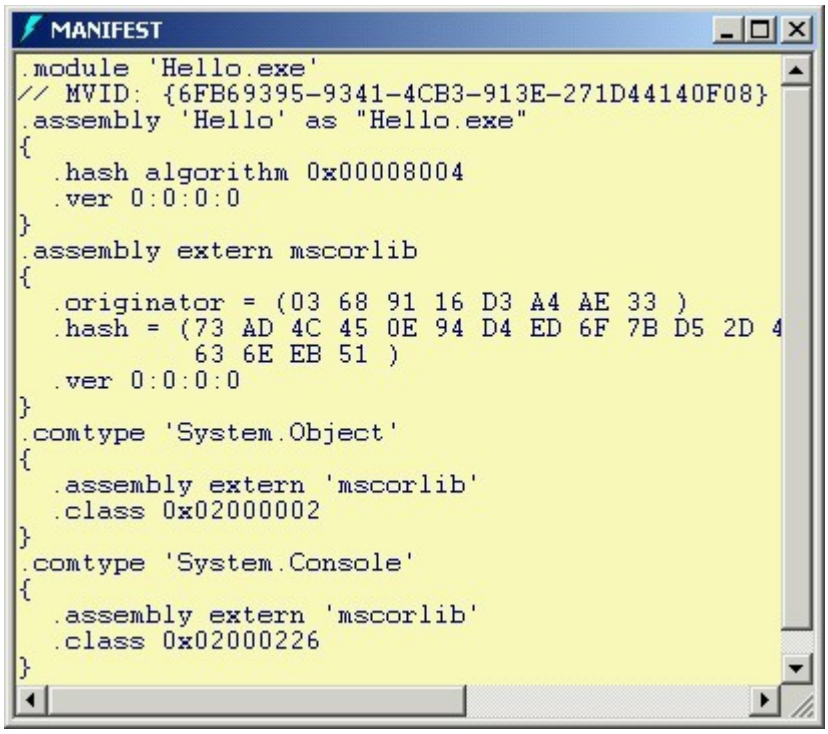


Figure ILDASM of Hello.exe Manifest

Here we can see information about the assembly including the version (not yet set) and which external libraries – and even which types within those libraries, in this case Object and Console – are used by the program.

ILDasm also shows the classes or types created within the program (in this case, the only class is **`MyClass`**) as well as the methods **`Main`** and a default constructor (indicated by .**`ctor`**). Our simple program doesn't have any other members. Information about the assembly can be saved to a file by using the **Dump** option on **File** menu.

(2) A SIMPLE COMPONENTIZED APPLICATION

Deployment

Deployment to machines with the NGWS runtime already installed couldn't be simpler. Our simple program can be run directly from a file server (more advanced programs might involve security issues), in which case no files are placed on the workstation, no entries are made into the system registry, and – in effect – there is no impact at all on the workstation. This also means that there is nothing to clean up (since there is nothing to uninstall). A related benefit is that running this program cannot "break" another program and neither can any other program cause this one to stuff functioning.

As you would expect, Hello.exe can also be copied to a local volume. In this situation, simply deleting the file is sufficient to "uninstall" the program and, again, nothing would remain on the workstation.

Of course, the "Hello World" discussed above is completely trivial and hardly representative of even the simplest real-world program. So, let's look at a version of the componentized program that was also described in detail in the *Developing with NGWS: An Introduction* tutorial. Our version, which can be found in the 2_Simple subdirectory, uses Client.exe to call only types contained in a single component (Stringer.dll). The code for the Stringer component (located in Stringer.cs) includes several important statements, the first of which specifies where in the global namespace the contained types can be found:

```
namespace org {…}
```

And again we must create a class:

```
public class Stringer {…}
```

Our class then has a single field (StringsSet), a constructor (Stringer, the same name as the class itself), a defined method (GetString), and a property (Count) with a corresponding property accessor (get_Count) automatically created by the compiler:

```
private string[] StringsSet;
public Stringer() {…}
public string GetString(int index) {…}
public int Count {
   get { return StringsSet.Length; }
}
```

Putting all of this together, then, a client program would fully qualify a reference to the **`GetString`** method (for instance) as **`org.Stringer.GetString`**.

Not surprisingly, an ILDASM display of the compiled component shows all of the members:

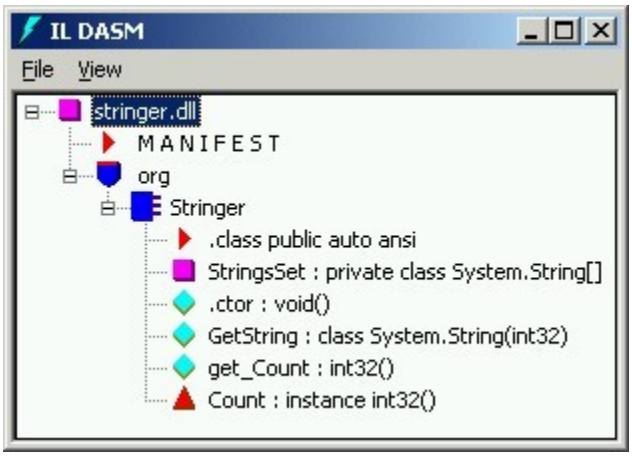


Figure ILDASM of Component Stringer.dll

Our client – the code is in Client.cs – includes a second **`using`** statement to allow easy access to the types in Stringer.dll by specifying the namespace:

```
using org;
```

Building the files in this project is straightforward. First we build the Stringer.dll component, then we build Client.exe and import the component using the name of the file containing the manifest rather than the namespace name (in this case, “org”):

```
csc /target:library Stringer.cs
csc /reference:Stringer.dll Client.cs
```

Just like Hello.exe, our new Client.exe contains manifest information about itself and the System library and types it uses. However, it now contains information about the Stringer component (we are using private assemblies where version information isn’t checked) as well as the referenced contained types (in this case org.Stringer):

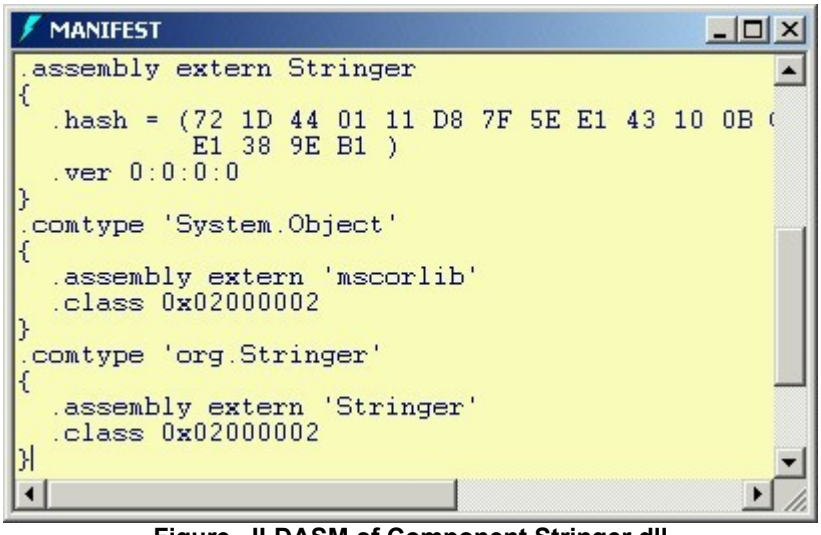


Figure ILDASM of Component Stringer.dll

Note that, in this particular example, the .DLL comprises the entire assembly: This is not always true however. For instance, in some development scenarios it may be

(3) PATH FOR PRIVATE COMPONENTS

necessary to combine .DLLs authored in several different languages into a single assembly. It may also proved advantageous to combine several .DLLs together into a single assembly to take advantage of special scoping rules that allows access to methods between components but internal to the assembly itself. In these situations, developers can use the Assembly Linker (AL) utility described in Appendix B to custom tailor their assemblies.

Deployment

As before, Client.exe can be run directly off a file server from any workstation with the NGWS runtime installed. Client.exe and Stringer.dll can also be copied to a local volume and deleting the two files would be sufficient to "uninstall" the program.

The Client example just discussed above has one important weakness: both Client.exe and Stringer.dll must reside in the same directory. In the real world, however, it may be advantageous to use a directory structure to manage components. NGWS provides a configuration mechanism that allows administrators to specify a directory from which to load private components.

Building on the previous Client example, the 3_SimplePath subdirectory contains a version of the program that works with a private directory. All of the source code is the same, but for illustration purposes the build process has been modified to have the Stringer dll build in the Stringer subdirectory:

```
csc /target:library /out:Stringer\Stringer.dll  ↵
   Stringer\Stringer.cs
csc /reference:Stringer\Stringer.dll Client.cs
```

While the **`/reference:`** compile option works to locate a component in a subdirectory when *compiling* the program, a separate XML-based application configuration file is required at runtime to support components located in other directories. For client executables like the ones we are covering in this tutorial, the configuration file resides in the same directory as the executable and has the name of the exe but with a file extension of .cfg. The sample file Client.cfg file specifies a **`PrivatePath`** tag:

Listing 1 Configuration file for Client.exe (Client.cfg)

```
<?xml version ="1.0"?>
<Configuration>
   <AppDomain
       PrivatePath="Stringer"
    />
</Configuration>
```

When this configuration file is placed in the same directory as the executable, at runtime the NGWS environment uses the PrivatePath to determine where to look for components in addition to the app directory.

Deployment

As with the previous example, this revised Client.exe can be run directly off a file

(4) A SHARED COMPONENT

server from any workstation with the NGWS runtime installed. Client.exe and Stringer.dll (and the application .cfg file) can also be copied to a local volume – using the same relative directory structure – and deleting the files (and directory) would be sufficient to "uninstall" the program.

While they are not used in this particular example, it is important to know that – in additional to application configuration files – NGWS also supports separate machine and user configuration files for many common configuration settings.

While the Client programs discussed above show the basics for constructing a complex program, they only illustrate the use of components that are private to the Client executable. On the other hand, many applications make use of components that are shared by many applications: These components – which are typically provided with by 3rd party developers – are installed in a common place on the system. By default, the system looks for each program's components in that place, known as the global assembly cache (GAC). In classic COM+ applications, this mechanism is heavily dependent on the Windows System registry where information about each component – including its version and physical file location – is stored. Unfortunately, while this method allows multiple applications to share a single component, it has also allows situations where installing a component with one application can overwrite the existing installed component, possibly causing other applications to break. This is often difficult to detect since the offending application appears to work fine and by the time the broken application is run, it might not be possible to recover the common files to a stable configuration.

Shared Names

The solution to the problems described above is to more strongly associate a distinct build (indicated by a combination of a version number and a special value called the originator) of a component assembly with the client application. The system can then take care to isolate these component assemblies so that different versions might be running at the same time for different client applications, something that wasn't possible in the past. This system of protection is sometimes called "side-by-side" execution (in contrast to "backwards compatible"), since applications can run along-side other versions of the same applications without affecting their respective execution environments.

The code to demonstrate adding these additional build attributes – and creating a protected shareable component – can be found in the 4_Shared subdirectory. Building on the code in 3_SimplePath, this step adds a second, shareable component that reverses an array of strings.

First let's build this new component assembly without specifying any options to make it shareable. If we simply compile the new Reverser.dll component: using, for instance NoShare.bat (located in the \4_Shared\Reverser subdirectory) :

```
csc /target:library /out:Reverser.dll Reverser.cs
```

we can then examine the metadata using ILDasm, which shows that the assembly is lacking an originator and does not have an established version number:

```
.module 'Reverser.dll'
.assembly 'Reverser' as "Reverser"
{  …
   .hash algorithm 0x00008004
   .ver 0:0:0:0
}
```

To mark an assembly as shareable, we have to compile it using a private key (public keys are used for verification). So, before compiling, we first have to generate a public/private key pair: We use the Shared Name (sn.exe) utility to generate a new key pair and place them in a file (found in the \4_Shared subdirectory):

```
sn –k orgKey.snk
```

Now that we have a private key, we are ready to compile the component, specifying the key file and the version number to be assigned:

```
csc /target:library /out:Reverser\Reverser.dll ↵
   Reverser\Reverser.cs /a.keyfile:orgKey.snk /a.version:1.0.0.0
```

If we run ILDasm again on Reverser.dll, we can verify that the assembly is now shareable as indicated by the presence of an **`.originator`** property and a non-default version (**`.ver property`**) of 1.0.0.0:

```
.module 'Reverser.dll'
.assembly 'Reverser' as "Reverser"
{  …
   .originator = (00 … FC 4A DC 9B 9C)
   .hash algorithm 0x00008004
   .ver 1:0:0:0
}
```

For more information on the SN, see the section on that utility in *Appendix B: Packaging and Deployment Tools* in this document.

Deployment

Deployment with shared components is more complicated than in the previous examples. While components can easily be shared by related applications simply by putting them in a common component directory, shared components that are used by many applications on the system are often stored in the system assembly cache.

As with the previous examples, this revised Client.exe can be run directly off a file server from any workstation with the NGWS runtime installed and Client.exe and Stringer.dll can also be copied to a local volume.

To install the shareable component into the system assembly cache, however, requires an additional step on the machine that will be running the corresponding Client.exe program:

```
al /install:Reverser\Reverser.dll
```

After installing the Reverser assembly, we can then examine the system assembly cache by navigating to the \Winnt\Assembly subdirectory and using the cache shell extension (see the Shell Cache Viewer section in the Appendix for more information):

(5) COMPONENT VERSIONING

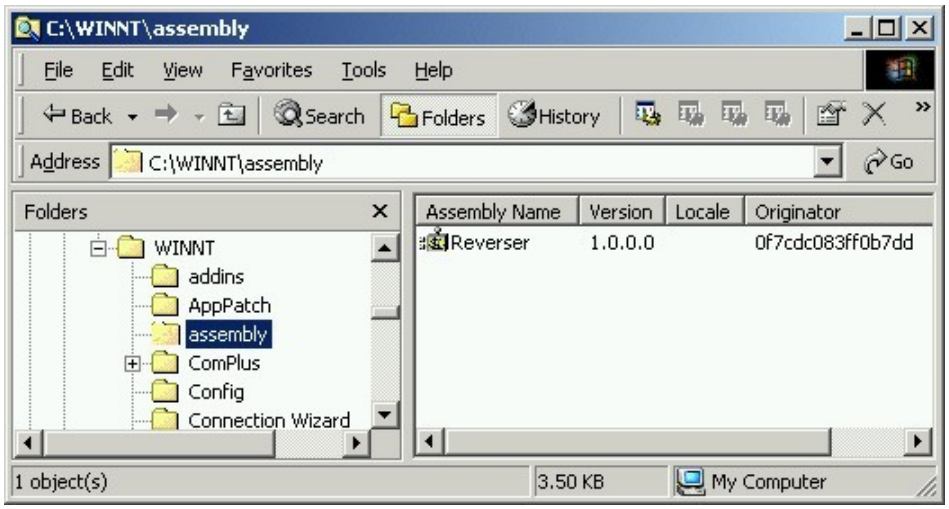


Figure System Assembly Cache

When it comes time to clean up the application, we need to do a little more work than with the previous examples. In addition to deleting the files, it's a good practice to remove the shared component file from the system cache (the system cache, unlike the download cache, is not automatically scavenged). In the PDC Tech Preview of the NGWS SDK, the easiest way to do this is to use the cache shell extension, select the appropriate component(s), and delete them.

Developers and administrators who wish to automate the process, however, will want to use the command-line interface to the system cache manager:

```
rundll32 fusion.dll, RemoveAssemblyFromCache Reverser
```

See "Appendix B: Packaging and Deployment Tools" for more information on this approach.

The final packaging and deployment step we will look at involves updating both the client and shared component to a new version. We will then deliberately update the shared component to break compatibility and demonstrate how NGWS allows us to configure the client application to continue to use the original version of the shared component. The key to making applications run safely is to understand how assembly compatibility versioning works.

Versioning

Each assembly has a specific compatibility version number as part of its identity. As such, two assemblies that differ by compatibility version are completely different assemblies as far as the NGWS runtime class loader is concerned.

This compatibility version number is physically represented as a 4-part number with the following format:

```
<major version>.<minor version>.<revision>.<build number>
```

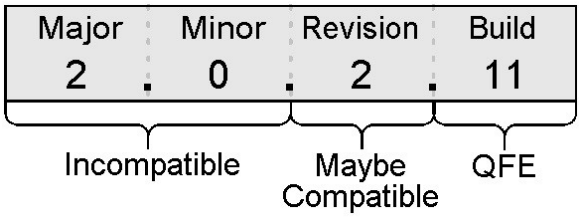
Each portion of this number has a specific meaning to the NGWS runtime as it decides which version of an assembly to load. Logically, the compatibility version number has three parts, with the following meanings:

Incompatible: A change has been made to the assembly that is known to be incompatible with previous versions. **Example:** Major new release of the product.

Maybe Compatible: A change has been made to the assembly that is thought to be compatible and carries less risk than an incompatible change. However, backwards compatibility is not guaranteed. **Example:** Service Pack or release of a new daily build.

QFE: An engineering fix that customers must upgrade to. **Example:** an emergency security fix.

These three logical parts map to the physical 4-part version number as follows:



For example, an assembly with compatibility version number 2.0.0.0 is considered incompatibly with an assembly whose compatibility number is 1.0.0.0. Also, compatibility number 2.0.2.11 is considered a QFE to compatibility number 2.0.2.1.

Shared Names

The solution to the problems described above is to more strongly associate a distinct build (indicated by a combination of a version number and a special value called the originator) of a component assembly with the client application. The system can then take care to isolate these component assemblies so that different versions might be running at the same time for different client applications, something that wasn't possible in the past.

The code to demonstrate adding these attributes – and creating a shareable component – can be found in the 4_Versioned subdirectory. Building on the code in 4_Shared, this step creates two separate versions of the shareable component and uses additional application configuration options to demonstrate how an application can made to run. A method in version 2.0.1.0 of Reverser.dll was deliberately made incompatible with the same method in 2.0.0.0 so that a Client that successfully called that method using version 2.0.0.0 would fail with the later revision.

To illustrate how versioning keys can change from one version of an assembly to the next, we generate a new key pair using the Shared Name (sn.exe) utility and place them in a file:

```
sn –k orgVerKey.snk
```

Now that we have a new private key, we are ready to compile both version 2.0 components, specifying the key file and the version number to be assigned:

```
csc /target:library /out:Reverser_v2.0.0.0\Reverser.dll ↵
   Reverser_v2.0.0.0\Reverser.cs /a.keyfile:orgVerKey.snk ↵
   /a.version:2.0.0.0
csc /target:library /out:Reverser_v2.0.1.0\Reverser.dll ↵
   Reverser_v2.0.1.0\Reverser.cs /a.keyfile:orgVerKey.snk ↵
   /a.version:2.0.1.0
```

If we run ILDasm again on the two updated Reverser.dlls, we can verify that the assemblies are shareable as indicated by the presence of a different (since we used a different key pair) `.originator` property and updated version (2.0.0.0 or 2.0.1.0, depending on which one we look at):

```
.module 'Reverser.dll'
.assembly 'Reverser' as "Reverser"
{
  .originator = (00 … 5D 85 7D 05 B3)
  .hash algorithm 0x00008004
  .ver 2:0:0:0
}
```

Deployment

To illustrate how applications can be configured to use shared components that are either the latest or are known to be compatible, we must install both 2.0 versions of Reverser.dll into the system assembly cache:

```
al /install:Reverser_v2.0.0.0\Reverser.dll
al /install:Reverser_v2.0.1.0\Reverser.dll
```

After installing these Reverser assemblies, we can then examine the system assembly cache by navigating to the \Winnt\Assembly subdirectory and using the cache viewer shell extension (see the Shell Cache Viewer section in the Appendix for more information):

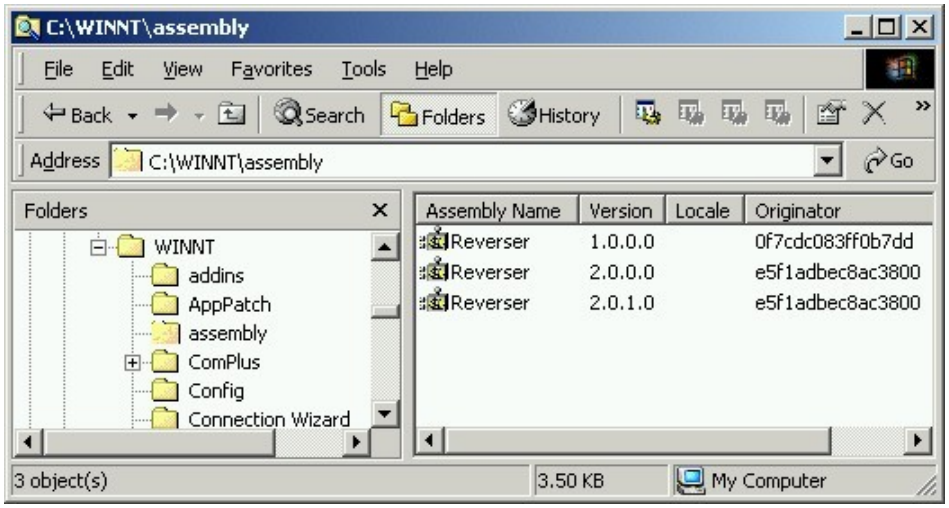


Figure System Assembly Cache with 2.0 Versions

We are now ready to compiled the VerClient executable, for which we specify the version 2.0.0.0 of the Reverser component:

```
csc /reference:Stringer\Stringer.dll; ↵
   Reverser_v2.0.0.0\Reverser.dll VerClient.cs
```

As mentioned in the section above in the section *Path for Private Components*, locating assemblies at runtime can be controlled using an application configuration file. In particular, the <BindingMode> and <BindingPolicy> tags can be used to redirect the reference to a different version of a shared assembly (private assemblies are not version checked).

The first runtime configuration option is the <BindingMode>, which can be configured to be “safe” or “normal”. In safe mode, the only version (meaning major and minor version as well as revision, but not QFE builds) of the shared assembly that can be used is the one that was also used at compile time.

```
<AppBindingMode Mode="safe"/>
```

Changing “safe” to “normal” and rerunning the VerClient program will cause the program to fail since the runtime is now loading version 2.0.1.0 of the Reverser component, which includes an incompatible interface.

The second runtime option is specified using <BindingPolicy>, which essentially means to override the version in the original reference with this version. Binding policies provide for several distinct configuration options. First, an application may be configured to use a specific newer version of a shared component than was used for compilation. The following option says that, regardless of which version (meaning, combination of major and minor version numbers) was used to compile the application (indicated by the “*”), the version that should be used at runtime is 2.0.0.0:

```
Version="*" VersionNew="2.0.0.0"
```

This allows an administrator to reconfigure an application without having to have it recompiled. An application may also be configured to automatically pick up the latest revision (meaning, the third field in the version number):

```
UseLatestBuildRevision="yes"/>
```

Note: UseLatestBuildRevision affects only a particular assembly, unlike AppBindingMode which affects all assemblies referenced in the application.

The sample VerClient.cfg file in the 5_Shared subdirectory contains all of these options:

Listing 2 Configuration file for VerClient.exe (VerClient.cfg)

APPENDIX A: FOR ADDITIONAL INFORMATION

```
<?xml version ="1.0"?>
<Configuration>
   <AppDomain
       PrivatePath="Stringer"
    />
   <BindingMode>
      <AppBindingMode Mode="safe"/>
      <!-- normal | safe -->
   </BindingMode>
   <BindingPolicy>
      <BindingRedir Name="Reverser"
           Originator="e5f1adbec8ac3800"
           Version="*" VersionNew="2.0.0.0"
           UseLatestBuildRevision="yes"/>
           <!-- no | yes -->
   </BindingPolicy>
</Configuration>
```

Since a method of a type in version 2.0.1.0 of Reverser.dll was deliberately made incompatible with the same method in 2.0.0.0, a 2.0.0.0-compatible client that attempts to call this later revision will fail. By changing from:

```
<AppBindingMode Mode="normal"/>
```

to:

```
<AppBindingMode Mode="safe"/>
```

it is possible for you (or, more typically, an administrator) to “repair” an application so that it will continue to run successfully if it happens to be broken by a subsequent install of another application that used a different version of the same shared component.

Finally, when it comes time to clean up the application we should remove the shared component files from the system cache:

```
rundll32 fusion.dll, RemoveAssemblyFromCache Reverser
rundll32 fusion.dll, RemoveAssemblyFromCache Reverser
```

This tutorial leads you through the process of packaging and deploying NGWS applications. You learned how to work with standalone, componentized, and versioned application types. You also learned how administrator could configure, or “repair” – an application by using a configuration file rather than having to obtain a newly compiled version of the application.

For the latest information on NGWS, visit the Technology Preview Web site at http://dapdweb.microsoft.com/ngws/sdk, which includes a pointer to all the NGWS newsgroups on Microsoft news server. Feedback – including reporting bugs – on the SDK can be submitted through http://dapdweb.microsoft.com/ngws/sdk/feedback.

ASP+

This tutorial focused on packaging and deploying traditional client applications. For information deploying ASP+ applications to Web server, please refer to the *Deploying ASP+ Applications* topic of the *ASP+ and Web Server Overview* section

APPENDIX B: PACKAGING AND DEPLOYMENT TOOLS

in the main NGWS Help (.chm) file.

Assemblies

For additional information on how NGWS locates assemblies when they are referenced at runtime, refer to the topic *How the NGWS Runtime Locates Assemblies* in the main NGWS Help (.chm) file. This topic covers the Assembly Resolver, shared and private assemblies, application and administrator version policies, codebase locations, and QFE updates.

The NGWS SDK includes several useful tools for examining assemblies and working with the system assembly cache. The CPTools.chm file contains additional documentation on these tools.

Assembly Linker (AL)

The Assembly Linker is used to both create assembly manifests and to install assemblies into the global assembly cache. Compilers and development environments may already provide either or both of these capabilities, so it is often not necessary to use this tool directly. The Assembly Linker will be most useful to developers needing to create a single assembly from multiple components files, such as might be produced with mixed-language development.

Shell Cache Manager (FUSION.DLL)

The NGWS utility library managing the system cache also provides a command-line interface. Since the extension is a .dll, it must be accessed using the rundll32.exe utility.

```
rundll32 fusion.dll, RemoveAssemblyFromCache Reverser
rundll32 fusion.dll, AddAssemblyToCacheA -m Reverser.dll –p
```

Note: the exported functions are case sensitive and must be typed exactly as shown. To obtain a complete list of command-line functions, you can list the exports from fusion.dll using the dumpbin.exe utility that comes with Visual C++:

```
C:>dumpbin /exports \WINNT\ComPlus\v2000.14.xxxx\fusion.dll
```

Where xxxx is the build version of NGWS you are using.

Shell Cache Viewer (SHFUSION.DLL)

To view the global assembly cache using the shell cache viewer, the shfusion.dll must be registered on the local system. In the PDC Tech Preview, this file is installed in the c:\Winnt\Assembly subdirectory but is not automatically registered by Setup. To register the viewer, it is necessary to run the following command line:

```
Regsvr32 c:\winnt\assembly\shfusion.dll
```

Note: If you decide to delete files from the global assembly cache using other methods, do not delete shfusion.dll.

Intermediate Language Disassembler (ILDASM)

You can also explore the namespaces in files– either those that come with the

runtime or those created by yourself or others - using the command-line IL disassembler tool to create a Windows output, for example:

```
C:\WINNT\ComPlus\v2000.14.xxxx>ildasm system.net.dll
```

which produces the following display:

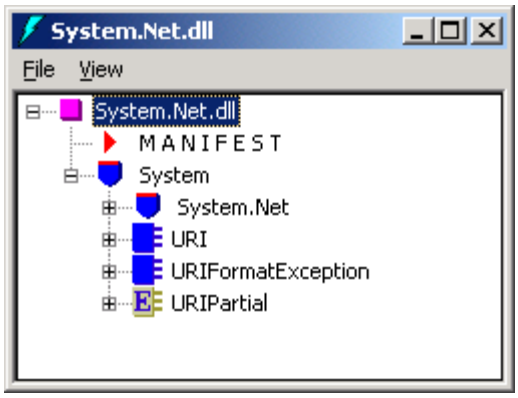


Figure IL Disassembler in NGWS SDK

Each of the namespace nodes represents a separate namespace, which can be expanded to explore the class objects and their methods and properties.

ILDasm also features a number of command-line options, which are particularly useful when redirecting output to the console or to a text file for subsequent analysis. One handy tip for developers is to put a shortcut to ILDasm.exe in their SendTo folder.

Shared Name (SN)

The Shared Name (SN) command-line utility can be used for several purposes when working with shared components. First, SN can be used to generate a new public/private key pair and write that pair to an output file:

```
-k <outfile>
```

SN can also be used to extract public key from key pair from a file and export it to a separate output file:

```
-p <infile> <outfile>
```

Two other options (-t and –T) can be used to extract key *tokens* from files. Rather than storing complete, for efficiency only the last 8 bytes (or 64 bits) are stored in key *tokens*. Finally, SN can also be used to verify an assembly for shared name signature self consistency:

```
-v[f] <assembly>
```

SN can thus be used to verify that a particular assembly was signed using a particular key file. Using the files in the above section *A Shared Componen*t, we first need to extract the public key out of orgKey.snk:

```
sn -p orKey.snk pub.snk
```

Then we can verify that that both components were signed by the same key pair by obtaining the same key token values for both of the following SN commands:

```
sn -t pub.snk
sn -T reverser.dll
```