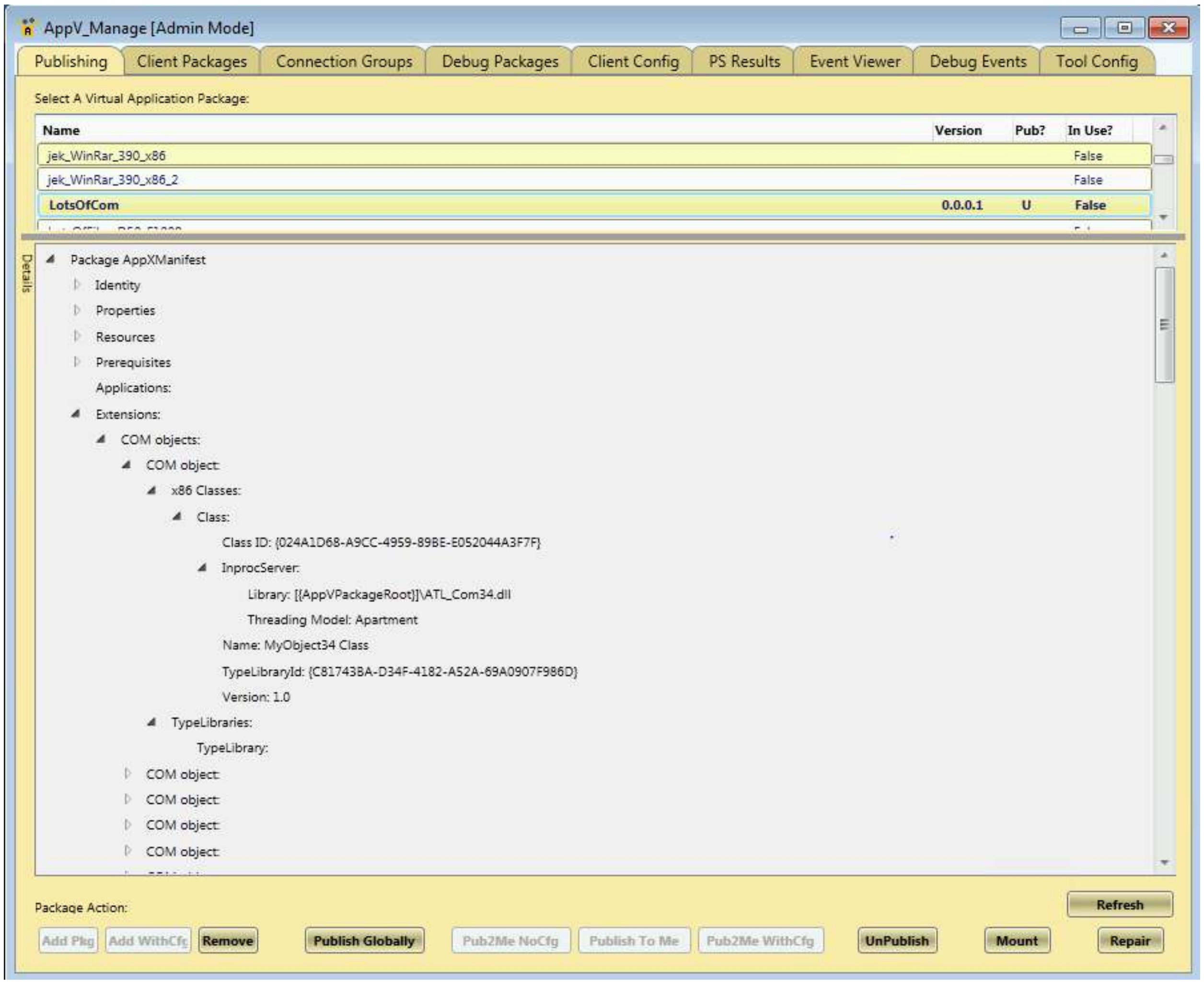
Effects of “Generic Com” in App-V 5 SP2 Deployment Performance

TMurgent Performance Research Series



Select A Virtual Application Package:

Name	Version	Pub?	In Use?
jek_WinRar_390_x86			False
jek_WinRar_390_x86_2			False
LotsOfCom	0.0.0.1	U	False

```
Package AppXManifest
  Identity
  Properties
  Resources
  Prerequisites
  Applications:
  Extensions:
    COM objects:
      COM object:
        x86 Classes:
          Class:
            Class ID: {024A1D68-A9CC-4959-89BE-E052044A3F7F}
            InprocServer:
              Library: [{AppVPackageRoot}]\ATL_Com34.dll
              Threading Model: Apartment
            Name: MyObject34 Class
            TypeLibraryId: {C81743BA-D34F-4182-A52A-69A0907F986D}
            Version: 1.0
        TypeLibraries:
          TypeLibrary:
      COM object:
      COM object:
      COM object:
      COM object:
```

June, 2014

Contents

1 Introduction

The purpose of this research paper is to document the effects that generic COM objects have in Microsoft App-V Virtual Application Packages.   The COM objects involved in this paper are generic run-of-the-mill in-process objects not used for integrations such as shell or browser extensions.

The effort is squarely aimed at answering questions on how such files in a package affect performance due to App-V's special handling of them. This work is part of a series of efforts to characterize the impact that different application elements have on the performance of virtual applications.

Most readers of this research will find themselves satisfied with reading the second and third section of this paper. The remaining sections detail the testing process, packages used, and provide further test details and additional findings.

2 Background on COM

To IT Professionals that have to deal with software, a COM[1] object is just a dll with a good PR agent. It really is just a fancy interface and GUID based registration into the windows Registry. There is nothing really special about COM files, except that the registration exists so that other programs can locate and use it.

While users of software can be indifferent to COM, developers have good reasons (or at least had good reasons in the past) to be interested in COM. When initially introduced, COM was the successor to DDE as a mechanism to achieve inter-process communication. To developers, the exciting thing about COM was that it allowed a clean multi-language interface. A COM object could be written in one language[2] and used by an application written in a different language. When it comes to programming languages, developers tend to be like Americans – they only know one language (and they probably butcher that one too). So being able to use a module written by someone else in a different language was a huge deal. Today, this isn't seen as such a big deal because Microsoft has in recent years made the different programming languages use a common base and interfaces – so language issues largely disappear.

In most cases, we want to isolate this COM registration to use within the package (or package group). On occasion, we want the COM object exposed externally for others to use. Shell extensions, and Browser Helper Objects are two examples of this that are specially detected and handled by App-V. But occasionally we find other unexpected integrations to COM objects needed to get large packages like SAS to work with Office.

2.1 In-Process, versus Out-of-Process COM

A Com dll may be declared to be "in-process" or "out-of-process", which is recorded in the registry registration of the object. This designation has everything to do with the mechanism for using the object; it does not signify whether the object is designed to be used by third party applications.

An in-process COM object is handled much like any other dll that you are used to dealing with. If a process needs to call the COM object, it asks for it using a standard windows API and passing in the GUID. This API looks up the GUID, determines it to be an in-process object, locates the dll file from the registration and loads it into the process space of the calling application.

[1] COM is short for "Common Object Model", which is programmer speak for something I can use without knowing what it is really made of.

[2] "Unmanaged" programming languages like Visual Basic, C, and C++, require translation stubs to call each other because the standard low level calling interfaces about registers and memory stacks were different.

An out-of-process COM object, when detected by the API, instead launches a TaskHost.exe process to load the object into its memory space and facilitates the communication between the calling process and the COM object running in TaskHost.

Developers tend to use the phrase “COM Server”, which can be confusing. When using the term they just mean the functionality exposed by the Com object. The Server term really comes from the concept that the software is provided as a service to other software.

2.2 32-bit and 64-bit COM objects

COM objects, being dlls are either 32-bit or 64-bit. To my knowledge, there are no “any cpu” COM objects as they are not managed dlls (.Net).

In-process COM objects, because they are loaded in the memory space of the calling application, must match the bitness of the calling application process. Because of this you may see both 32 and 64 bit implementations of the same COM object included in a package when you install on a 32-bit OS.

Newer “managed code” (.Net) can be delivered as “any-cpu”, which means it will run in the bitness of the OS. This code can call COM objects, so having both 32 and 64 bit versions makes sense if the COM object is “in-process”.

Out-of-process COM objects need not match the bitness of the calling application.

2.3 COM+ and DCOM

COM+ is an extension to COM to make the object act as an on-demand service. DCOM is an extension to make things available from another machine. These have traditionally been considered non-virtualizable by the older application virtualization product, and both are detected by the sequencer which will produce warnings to indicate that functionality of the virtual application may not work.

There are some hints that some kinds of COM+ might be able to work with App-V 5, except that it is not supported via sequencing. Fortunately, there is little COM+ out there so the demand isn’t high enough to worry about. Microsoft PFE Steve Thomas has a nice writeup on COM+ and DCOM here: http://blogs.technet.com/b/gladiatormsft/archive/2013/11/08/app-v-on-com.aspx.

2.4 App-V Isolation of COM objects

App-V, in general, wants to isolate addressable components. Often, this is accomplished via file or registry virtualization (redirection), so that things outside the bubble won’t see them. But sometimes privatization must be performed.

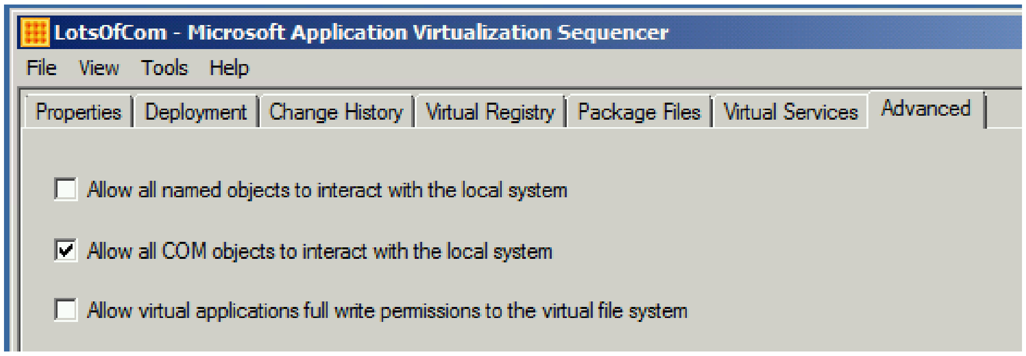
For many “named” kernel objects, isolation is achieved by privatizing the name used. For example, if an app used a memory mapped segment (a kind of named kernel object) to communicate between two processes in an application, the application processes would coordinate the memory segment by using a unique name that only those two processes would know. It is necessary to privatize the name as part of a virtual environment. This allows the two processes from one instance to see each other, but not the memory segment of a different virtual environment. This is important on RDS when two users run the same application, or even on a single user desktop OS when two versions of the same app run at the same time. You may have noticed a list of named objects in the registry that App-V does not privatize. This list was generated through years of experience and you *almost* never have to modify that list.

Another form of privatization occurs for .net component names. .Net components that would normally end up in the Global Assembly Cache (GAC for short) need privatization to not be seen by things outside the bubble. In App-V 4.6, we even had a special policy to disable this privatization, although I don’t remember running into an app that needed this disabled.

For a COM object, I think that isolation could be achieved by making the registry based registration visible or not via redirection. But apparently[3] the software designers at Microsoft decided that privatization was the right solution. Privatization, in this case, means renaming the COM GUID used for access.

By default, the developers at Microsoft decided to *Isolate* in-process COM objects and *integrate* out-of-process COM. But you can change this.

At the sequencer, these defaults may be overridden selecting the checkbox on the Advanced tab of the Sequence Editor (see image below). This setting changes both types of COM objects to be integrated.



[ ] Allow all named objects to interact with the local system

[✓] Allow all COM objects to interact with the local system

[ ] Allow virtual applications full write permissions to the virtual file system

[3] Microsoft PFE Steven Thomas calls this COM GUID Spoofing here: http://blogs.technet.com/b/gladiatormsft/archive/2013/09/11/app-v-on-com-isolation-and-interaction.aspx

Alternatively, you can also make changes in the DeploymentConfig.xml file to use during deployment. In the xml, you have full control over the isolate/integration setting for each kind of COM independently. (The checkbox inside the sequencer simply sets both types to integrated, both in the internal AppXManifest file and DeploymentConfif.XML file).

2.5 Specialized COM objects

While this paper is focused on generic COM objects, App-V also provides support for many specialized COM objects. These are COM objects that have a specific type of supported interfaces and are secondarily registered for their specific purpose.

Most of the added extension points in 5.0 SP2 are specialized COM.

An example of this is the Browser Helper Object (BHO). A BHO is a COM object used to interact with web browser activity. A BHO is used exclusively today for web browsers, like Internet Explorer. In the past, the desktop shell (windows explorer) also supported BHOs, but this support is no longer present on OSs supported by App-V 5. If you remember the “Active Desktop” in Windows XP, you likely used a BHO in Windows Explorer. The specific interface in a BHO registers itself to be called in response to certain events, such as completion of the reading of a web page or document file. The BHO can handle or alter the information, or take other actions as needed.

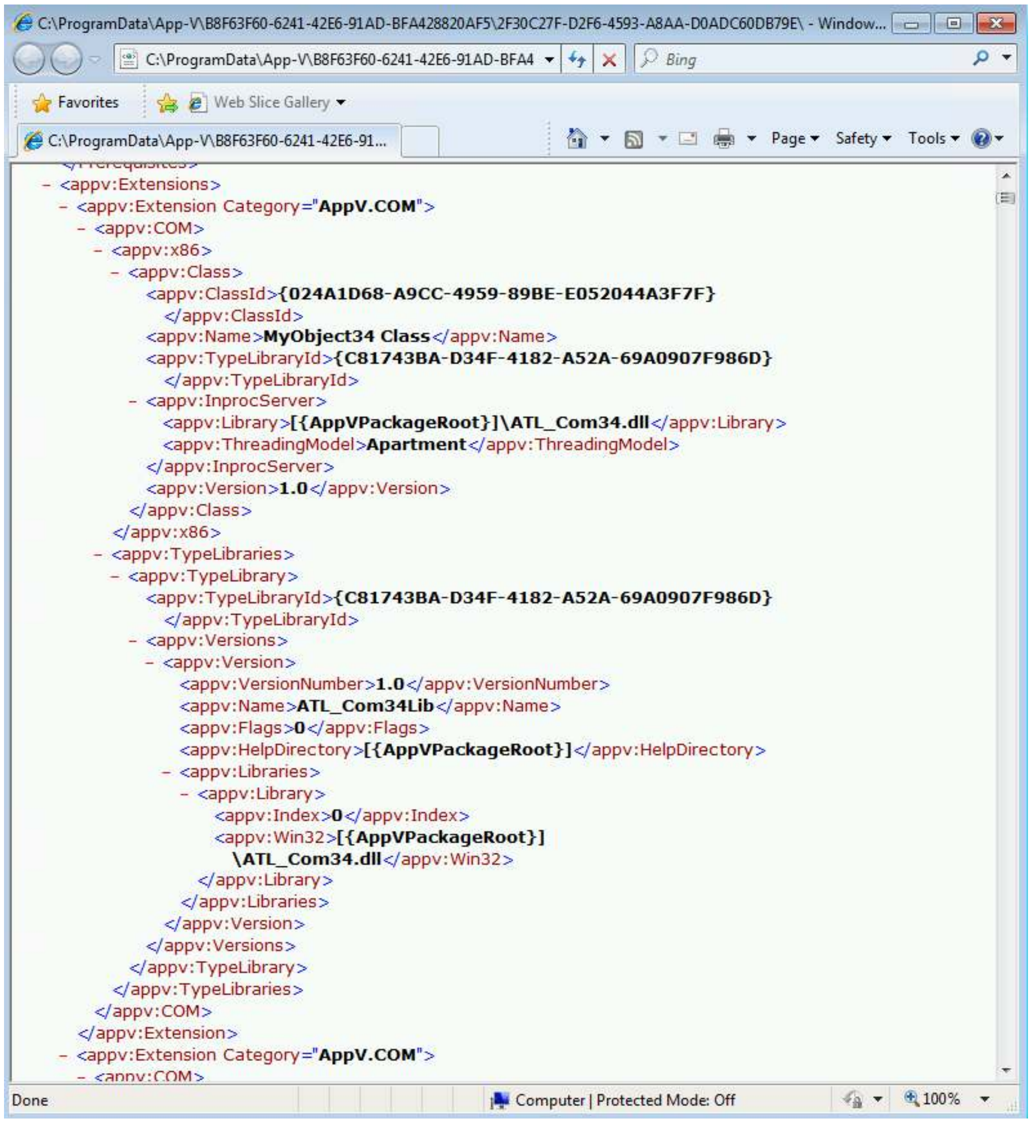
COM objects supported by App-V as extension points have different performance characteristics than the generic COM objects tested in this paper. These objects receive additional attention during deployment and will generally have a greater deployment impact. The greater impact of these objects, however, happens at runtime outside of a given package. Which is a very different kettle of fish to boil.

This paper may be useful in conjunction with other papers to understand the impacts of larger packages that you might choose to deploy over App-V.

2.6 Detecting COM Objects.

The sequencer provides no information regarding COM use inside the package. With experience, you can probably recognize COM object registration in the virtual registry by scanning the CLSID keys under both MACHINE and USER virtual registry.

Detected COM objects are detected by the sequencer and enumerated in the internal AppXManifest file.



C:\ProgramData\App-V\B8F63F60-6241-42E6-91AD-BFA428820AF5\2F30C27F-D2F6-4593-A8AA-D0ADC60DB79E\ - Window...

```
- <appv:Extensions>
  - <appv:Extension Category="AppV.COM">
    - <appv:COM>
      - <appv:x86>
        - <appv:Class>
            <appv:ClassId>{024A1D68-A9CC-4959-89BE-E052044A3F7F}
              </appv:ClassId>
            <appv:Name>MyObject34 Class</appv:Name>
            <appv:TypeLibraryId>{C81743BA-D34F-4182-A52A-69A0907F986D}
              </appv:TypeLibraryId>
          - <appv:InprocServer>
              <appv:Library>[{AppVPackageRoot}]\ATL_Com34.dll</appv:Library>
              <appv:ThreadingModel>Apartment</appv:ThreadingModel>
            </appv:InprocServer>
            <appv:Version>1.0</appv:Version>
          </appv:Class>
        </appv:x86>
      - <appv:TypeLibraries>
        - <appv:TypeLibrary>
            <appv:TypeLibraryId>{C81743BA-D34F-4182-A52A-69A0907F986D}
              </appv:TypeLibraryId>
          - <appv:Versions>
            - <appv:Version>
                <appv:VersionNumber>1.0</appv:VersionNumber>
                <appv:Name>ATL_Com34Lib</appv:Name>
                <appv:Flags>0</appv:Flags>
                <appv:HelpDirectory>[{AppVPackageRoot}]</appv:HelpDirectory>
              - <appv:Libraries>
                - <appv:Library>
                    <appv:Index>0</appv:Index>
                    <appv:Win32>[{AppVPackageRoot}]
                      \ATL_Com34.dll</appv:Win32>
                  </appv:Library>
                </appv:Libraries>
              </appv:Version>
            </appv:Versions>
          </appv:TypeLibrary>
        </appv:TypeLibraries>
      </appv:COM>
    </appv:Extension>
  - <appv:Extension Category="AppV.COM">
    - <appv:COM>
```

Done   Computer | Protected Mode: Off   100%

From this information you can determine if the object is in-process or out-of-process[4]. Which, when facing an application integration issue, might be helpful in determining if enabling in-process COM integration might help.

[4] You can also see the "Apartment Model" used, but this does not appear to be helpful for our purposes.

3 Summary of Where Impacts of Generic COM Are Felt

Generally, the impact of generic COM objects to performance is quite small. Integrated COM affects both Add and Publish steps, while isolated COM affects only the Add step. Some impact on Runtime is also detected, but it is quite small.

Specialized COM objects that are shell extensions were not included in the testing, but would add more significant impacts outside of deployment.

4 Test Strategy Used

This section provides details about how the testing was performed.

4.1 About the Testing Platform

The testing results depicted in this paper are based on:

App-V 5.0 SP2 with HotFix 4 running on a Windows 7 SP1 x86 virtual machine.

The testing was performed in an isolated environment using a Microsoft 2012 R2 server with Hyper-V. The server has 24 processors and 64GB or RAM. To minimize external impacts, this server utilizes local storage and contains a VM with the domain controller. App-V Package sources were located on a share on this host.

The Test VM used had 2GB of RAM and was given 2 virtual CPUs. The App-V Client is configured for Shared Content Store mode (which disables background streaming).

4.2 About Test Packages and “Streaming Configuration”

All Test packages used are specially constructed software packages that I developed. These packages are generally stripped down to a bare minimum, except for an overabundance of the one particular things we want to measure when using this package. In many cases, this means custom software that I developed for the purpose of the test.

Unless specifically noted, each package was sequenced and configured for streaming by *not* launching anything during the streaming training configuration phase of the sequencer. This means that, barring mounting operations, almost everything in the package will fault-stream (stream on demand).

4.3 About the Testing Methods

All tests are automated using significant sleep periods before each portion of the testing to allow all systems to settle down, and warm-up of the external components (hypervisor/fileshare) and within the OS (App-V Client and drivers) are performed. The test process consists of

- A ***Test Cycle*** that consists of a series of Test Passes.
- Each ***Test Pass*** consists of a number of Test Packages.
- Each ***Tested Package*** is tested using a series of actions and measurements.

A *Tested Package*, consists of a series of actions, always preceded by a significant sleep period to allow system background processes to settle down.

A *Test Pass* always starts from a freshly booted snapshot and with a dummy *Test Package* to warm up the App-V Client and Driver sub-systems. The results of this dummy package are not used.

A *Test Cycle* always starts with a *Test Pass* to warm up the external components of the Hypervisor and Windows File Share. Because the packages are relatively small compared to the amount of memory available, the packages are likely retained in memory in the Windows Standby Lists after the initial *Test Cycle.*

These are described as follows, from the bottom up.

4.3.1 Test Package

For a given ***Test Package***, the series of actions includes:

- Waiting
- Add-AppVClientPackage
- Waiting
- Publish-AppVClientPackage
- Waiting
- [Optionally Mount-AppVClientPackage[5]]
- Waiting
- First run (launch “cmd.exe[6] /c time /t” inside the virtual environment).
- Waiting
- Second run (launch “cmd.exe[7] /c time /t” inside the virtual environment).

The time required for each of the actions to complete is recorded.

4.3.2 Test Pass

A ***Test Pass*** consists of testing multiple *Test Packages* as follows:

- Reverting the test VM to a snapshot.
- Waiting for the Hypervisor to settle.
- Booting the VM and logging in.
- Waiting.

[5] With SCS enabled, mounting the package does result in the actual file content being stored in the App-V file cache. I test in SCS mode both with and without mounting to better delineate the cause of performance slowdowns on a package.

[6] This is used rather than a program in the package to produce a comparable time that varies based on special actions that the client must perform during virtual environment startup and shutdown due to the package content.

[7] The client is also known to perform special actions the first time a virtual environment is used, so the second run is used for comparison to the first run.

- A series of actions and measurements on a warm-up package. These results are never used, it is only performed to warm up the client (client service, drivers, and WMI) and to ensure that each subsequent package fairly tested under similar conditions.
- Waiting.
- A series of actions and measurements on the first package.
- Waiting.
- A series of actions and measurements on the second package.
- Etc…
- Recording results

4.3.3 Test Cycle

Finally, A ***Test Cycle*** consists of several consecutive test runs of the same *Test Pass*. The first pass is used to “warm up” external systems and achieve a relatively consistent amount of caching by the server. The results of this pass are not used, but the results of the remaining passes are averaged to produce results. A Test Cycle typically requires a full day to complete.

4.4 About the Test Results Accuracy

As careful as I attempt to be to eliminate variability in the results, there is a fair amount of variability in results between two passes.

Due to the nature of the background interruptions affecting the results, the impact on result accuracy is felt much more on measurements that are shorter in duration than those that are longer. With measurements that are sub-second, this can produce results that typically vary by as much as +/-10% from the average.

Instead, I use an approach to test with a sufficient number of test cycles and select the minimum value seen on any of the tests. The more repetitions that are made, the better this minimum value represents the time it takes for App-V to complete the task without the effects of any extraneous background interference.

5 Test Packages Utilized

This section details the packages used in testing.

5.1.1 Warmup Package

This package is primarily used as the first package in a Test Pass, to warm up the OS and App-V Client components and dependencies[8].

5.1.2 LotsOfNothing (Baseline)

This is a minimal App-V Package.

In developing this package, I discovered that there is an issue with the App-V Client in that there appears to be some sort of undocumented minimal package requirements. If you create a package with no registry entries, no files, and no integrations, the Add-AppVClientPackage cmdlet will error out with error 700002.

Therefore this package consists of one HKLM registry key, one HKCU registry key, one text file in the PVAD folder, and one shortcut (to the text file).

The package was tested to produce a baseline for “absolute minimum” of what the App-V Client can do. These numbers are useful in determining the amount of overhead that the VC Runtimes place on the system.

5.1.3 LotsOfComIsolated

This package consists of a 100 small in-process COM dlls that are registered, but not used. Each COM dll contains a single COM object exposing a single interface which does nothing.

5.1.4 LotsOfComIntegrated

This package consists of a 100 small in-process COM dlls that are registered, but not used. Each COM dll contains a single COM object exposing a single interface which does nothing.

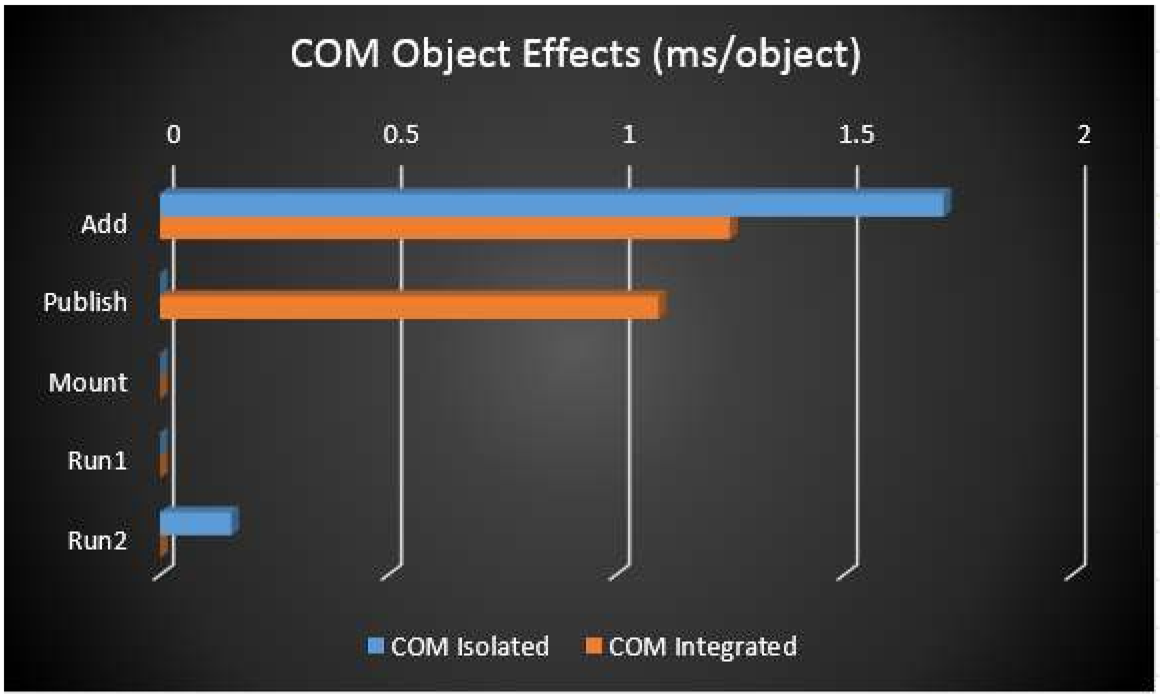
[8] When conducting tests that use mounting, I found it necessary to warm up the system without mounting this package. It appears that the first client activity after boot requires additional time to warm up the client, possibly loading drivers. But I also found that mounting this package causes an odd additional 1 second hit to any subsequently Add-AppVClientPackage commands (even after settling time). This issue only seems to exist with this package, and mounting other packages does not affect subsequent Add cmdlets. The cause of this is unknown.

6 Detail Test Results

Results reported are based on an ideal test environment. Performance impacts identified in this paper will be very different in production environments. Specific numbers are *only* useful in comparison to numbers from other research papers in this series!

6.1 SCS Mode Testing with Mounting

Tests were performed with and without Mounting and with and without SCS mode enabled. The results show that the impact is mostly felt with during the add and publish steps (and a little at runtime), so the results are similar and only the SCS mounting results are shown below.



In situations where deployment performance is crucial, such as VDI scenarios, these results show that generic com objects have little impact.

From the numbers we reach the following conclusions:

Isolated Generic COM items have minimal impact in general, each extra COM file adds about 1.6ms each to the Add step and nothing measurable to the Publish Step.

Integrated Generic COM items have only slightly more impact , each extra COM file adds about 1.25ms each to the Add step and 1ms to the Publish Step.

7 About This Research Paper Series

This research paper is part of a series of papers, released by TMurgent Technologies, that investigate the performance impacts that certain application contents can have in the deployment of Microsoft App-V 5 packages.

Through these papers, we can better understand what areas to focus on when packaging applications for App-V when deployment and end-user experience is important. Additionally, with an understanding of these papers you can better target a specific package that is performing poorly and prioritize your efforts to improve it.

TMurgent Technologies, LLP is based in Canton, MA, USA; just 17 miles south of the offices where Microsoft develops the App-V product. TMurgent's Tim Mangan has a long history with the product, having built the original version at Softricity more than a dozen years ago. TMurgent is well known in the App-V community as a source for the best training classes on App-V as well as an endless supply of tools and information. More information is available at the website, www.tmurgent.com